VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉMY SYNTAKTICKÝCH ANALYZÁTORŮ

PARSER SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAN HRSTKA**
AUTHOR

VEDOUCÍ PRÁCE **Prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Hrstka Jan, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Systémy syntaktických analyzátorů**
Parser Systems
Kategorie: Teoretická informatika

Zadání:

1. Dle instrukcí vedoucího se seznamte s automatovými a gramatickými systémy. Seznamte se s pokročilými verzemi syntaktických analyzátorů.
2. Zaveďte systémy syntaktických analyzátorů analogicky jako gramatické či automatové systémy.
3. Studujte vlastnosti těchto systémů dle instrukcí vedoucího.
4. Dle pokynů vedoucího uvažujte různé části překladačů. Formalizujte je prostřednictvím těchto systémů.
5. Implementujte formalizace navržené v bodě 4.
6. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1
- Meduna, A.: Elements of Compiler Design, New York, Taylor & Francis, 2008
- Levine, J.: flex & bison, O'Reilly Media, 2009, ISBN 978-0-596-15597-1

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a část 3 (alespoň dvě vlastnosti).

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

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Tato práce poskytuje ucelený přehled poznatků z oblasti gramatických systémů. Práce navrhuje, jak paralelně orientované systémy využít v sekvenční syntaktické analýze. Koncept gramatických systémů dále rozšiřuje na úroveň samotných syntaktických analyzátorů, které seskupuje do větších celků a zkoumá vlastnosti těchto uskupení. Cílem práce je obohatit syntaktickou analýzu o přístupy založené na těchto systémech. Vychází z bezkontextových metod syntaktické analýzy, které propojuje a rozšiřuje. Pozornost je především věnována zvýšení generativní kapacity LL a LR syntaktické analýzy. V rámci práce se podařilo sestrojit bezkontextové struktury, které jsou schopny přijímat kontextové jazyky. Práce zároveň poskytuje návod k jejich implementaci. Prezentuje obecný koncept syntaktické analýzy, který zvyšuje generativní kapacitu standardních metod. Využitím uvedeného přístupu je možné rozšířit řadu používaných jazyků o kontextové prvky, především o prvky popírající větu o iteraci.

Abstract

This thesis provides a summary of knowledge of grammar systems. The thesis proposes modifications of parallel oriented grammar systems to be usable in sequential parsing. Concept of grammar systems is extended to level of entire parsers, that are grouped into parsing system. Then the properties of these systems are examined. The aim of thesis is to introduce approaches to syntactic analysis based on grammar systems. Thesis is based on context-free methods of syntactic analysis, extending them and connecting them together. Great attention is dedicated to increase generative capacity of LL and LR parsing. There were created context-free structures within this thesis, which are capable to generate context-sensitive languages. This work also provides a simple recipe for implementation of these structures. We introduced generic concept of parsing, that enlarge generative power of conventional parsing methods. Using presented techniques it is possible to extend many of often used languages with context-sensitive elements, especially elements contradicting with pumping lemma.

Klíčová slova

gramatické systémy, LL syntaktická analýza, LR syntaktická analýza, bezkontextové metody, zpracování kontextových jazyků, zvýšení generativní kapacity

Keywords

grammar systems, LL parsing, LR parsing, context-free approach, parsing context-sensitive languages, enlarge generative power

Citace

HRSTKA, Jan. *Systémy syntaktických analyzátorů*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Systémy syntaktických analyzátorů

Prohlášení

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

.....
Jan Hrstka
16. května 2019

Poděkování

Chtěl bych poděkovat vedoucímu práce, panu prof. RNDr. Alexandru Medunovi CSc., za poskytnutou odbornou pomoc a panu PhDr. Miroslavu Hrstkovi Ph.D. za pravopisnou a typografickou korekci.

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

V klasické teorii formálních jazyků od počátků převládaly centralizované výpočetní prostředky, gramatiky, automaty, u nichž je výpočet řízen jednou centrální autoritou. Jazyky byly generovány pouze jednou gramatikou či přijímány jedním automatem. V 80. letech 20. století se objevila myšlenka distribuovaného výpočtu a začalo se zkoumat, jak se změní vlastnosti gramatik, pokud je seskupíme do větších celků, gramatických systémů. Ukázalo se, že takové seskupení zvyšuje generativní sílu gramatik a zároveň dochází ke snížení složitosti – pomocí relativně jednoduchých gramatických systémů je možné popsat jazyky, jež nespadají do třídy bezkontextových jazyků. Dnes již distribuovaný výpočet hraje na poli teoretické informatiky majoritní roli, tato myšlenka se uchytila především v oblastech počítačových sítí či databázových systémů. K rozšíření přispěla i možnost paralelního zpracování a rozvoj víceprocesorových výpočetních strojů.

Co to tedy je gramatický systém? Jedná se o množinu společně pracujících gramatik generujících jeden jazyk. Tyto gramatiky mezi sebou komunikují pomocí *kooperačního protokolu*. Právě způsob komunikace je zcela klíčový, v některých případech je teorie gramatických systémů spíše chápána jako teorie komunikačních protokolů. Výzkum gramatických systémů se rozštěpil podle způsobu komunikace na větev sekvenční a větev paralelní[7].

Pro sekvenčně orientované gramatické systémy se vžil název *kooperačně distribuované* (CD). V takovém systému všechny gramatiky pracují na jediné větné formě. V systému je vždy právě jedna gramatika *aktivní*. Ta je vybírána na základě kooperačního protokolu. Aktivní gramatika pracuje na větné formě tak dlouho, dokud není splněna *ukončující podmínka*, rovněž stanovená kooperačním protokolem. Následně je činnost aktivní gramatiky ukončena a řízení je předáno další gramatice. Ukončujících podmínek je mnoho, my se v následujících úvahách zaměříme na pět základních: (i) gramatika provede nejvýše k kroků, (ii) právě k kroků, (iii) nejméně k kroků, (iv) nejvyšší možný počet kroků či (v) libovolný počet kroků (krokem je myšlena aplikace přepisovacího pravidla).

Druhou větev tvoří *paralelně komunikující* (PC) gramatické systémy. Každá gramatika v takovém systému má svůj počáteční neterminál a svoji vlastní větnou formu, na které pracuje. Celý systém je synchronizován podle hodin – v každém taktu provedou všechny gramatiky aplikaci některého přepisovacího pravidla. Klíčovou vlastností je komunikace pomocí *žádostí* – pokud některá gramatika potřebuje komunikovat v rámci systému, zveřejní ve své větné formě *dotazovací symbol* Q_i. Po jeho zveřejnění některou z gramatik dochází ke *komunikačnímu kroku*, ve kterém se všechny výskyty dotazovacího symbolu Q_i nahradí větnou formou i-té gramatiky. V PC gramatickém systému se nachází jedna vyvolená komponenta nazývaná *master*. Jazyk generovaný touto komponentou je rovněž jazykem generovaným celým systémem.

Výše zmíněné systémy navazují na dva populární modely řešení problémů. Prvním modelem je *model tabule*[7]. V tomto modelu je celý problém (zbývající otázky k vyřešení i současný stav) formulován na tabuli. Problém řeší několik *autorit* (agentů), které působí jako zdroje znalostí, ale pouze jedna autorita může v rámci svých kompetencí pracovat na tabuli. Po provedení svých zamýšlených kroků přenechá práci další autoritě. Domluva, která další autorita bude na tabuli pracovat, probíhá podle výše zmíněného kooperačního protokolu.

Druhým z modelů je *třídní model*[7]. V tomto modelu máme jednu centrální autoritu – vyučujícího a poté libovolný počet autorit jí podřízených – studentů. Celý problém je rovněž formulován na tabuli, ale jediný, kdo ji může editovat, je právě vyučující. Vyučující rozdělí problém na tabuli do podproblémů, které nechá řešit studenty. Každý student poté pracuje na problému ve svém sešitu. Pokud některý student potřebuje další informace, může se zeptat ostatních studentů či vyučujícího v závislosti na komunikačním protokolu – protokol určuje, zdali studenti mohou klást otázky pouze na výzvu od vyučujícího, nebo kdykoliv během řešení problému. Vyučující se rovněž může dotazovat studentů na jejich vyřešené podproblémy, nebo pouze počká, až mu studenti sami sdělí jejich výsledky.

Oba teoretické modely jsou už známy mnoho let a existují způsoby, jak je interpretovat a implementovat. Před jejich objevením se svět informatiky v této oblasti zaměřoval především na výzkum bezkontextových gramatik. Vzniklo několik prakticky zaměřených metod určených k syntaktické analýze bezkontextových jazyků – od prediktivní syntaktické analýzy s využitím LL tabulky, přes precedenční syntaktickou analýzu až k LR syntaktické analýze, která poskytuje stejnou generativní sílu, jako samotné deterministické zásobníkové automaty. S rozvojem informačních prostředků přestávala síla bezkontextových gramatik stačit. Chceme-li na počítači zpracovávat přirozený jazyk, je nutné se posunout za hranice množiny jazyků generovaných zásobníkovými automaty. Dnes již mnohé programovací jazyky v sobě zahrnují prvky, jež nejsou bezkontextové – příkladem můžeme uvést C++[9] nebo Python[10]. Z těchto důvodů se rozběhl výzkum kontextových gramatik a na ně navázaných metod syntaktické analýzy. Ty ovšem nejsou zdaleka tak jednoduché, jako metody bezkontextové. Samotná pravidla kontextových gramatik jsou špatně čitelná, a proto odhalení jazyku generovaného takovou gramatikou vyžaduje jistou míru úsilí.

V následujících stranách se pokusíme využít bezkontextové metody k analýze kontextových struktur. Několika jednoduchými úpravami docílíme toho, že bude možné analyzovat některé kontextové jazyky pomocí bezkontextových pravidel, což je významný přínos k čitelnosti gramatik a umožňuje to jednodušší zpracování těchto jazyků. Kromě teorie potřebné k pochopení problematiky, si zavedeme několik množin, pomocí kterých provedeme transformaci PC gramatických systému na skupinu rozšířených LL tabulek. Nad těmito tabulkami vystavíme syntaktický analyzátor, který bude simulovat činnost centralizovaného PC gramatického systému. V druhé části práce se v našem smýšlení posuneme o úroveň výše. Nebudeme vytvářet jeden syntaktický analyzátor, ale pokusíme se získat vyšší generativní kapacitu spojením několika jednoduchých bezkontextových analyzátorů do většího celku – systému syntaktických analyzátorů (SA systému). V jazyce C++ implementujeme nástroj k návrhu a testování SA systémů a s jeho využitím vytvoříme syntaktické analyzátory pro některé dobře známé kontextové jazyky.

1.1 Prerekvizity z formálních jazyků

Tato práce je určena čtenářům, kteří již z jiné literatury získali základní znalosti z teorie formálních jazyků. Vhodným průvodcem základy pro čtenáře nevyhýbající se anglické literatuře může být práce Rozenberga a Salomaa [8] či Meduny [6], ovšem lze nalézt i česky psanou literaturu, například [11], skýtající ucelený přehled témat, na které v tato práce navazuje.

Pro lepší orientaci v textu si nyní zavedeme jednotné konvence označování, které jsou v celé práci dodržovány. Nechť Σ je *abeceda* (konečná množina symbolů), poté označení Σ^* reprezentuje množinu všech řetězců nad touto abecedou. Prázdný řetězec budeme značit ε a $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ je množina všech řetězců nad abecedou Σ obsahujících alespoň jeden symbol. Jazykem rozumíme libovolnou podmnožinu Σ^*.

Řetězce budeme označovat malými písmeny z konce abecedy, typicky x, y, z, v, w. Délku řetězce označíme jako $|x|$ a pro $\sigma \subseteq \Sigma$ bude označení $|x|_\sigma$ reprezentovat počet výskytů symbolů z σ v řetězci x (délka řetězce získaného z x odstraněním všech symbolů z $\Sigma - \sigma$). *Chomského gramatikou* myslíme uspořádanou n-tici $G = (N, T, P, S)$, kde N je *abeceda neterminálů*, T je *abeceda terminálů*, S je *počáteční neterminál* (též *axiom*) a P je konečná množina *přepisovacích pravidel* nad $N \cup T$ ve tvaru $u \to v$, kde $u, v \in (N \cup T)^*$, $|u|_N \geq 1$. Pravidla se stejnou levou stranou budeme pro zjednodušení značit $u \to v|w$, kde $w \in (N \cup T)^*$. Terminální symboly gramatik označíme malými písmeny ze začátku abecedy, typicky a, b, c, d. Neterminální symboly budou značeny velkými písmeny. *Derivační krok* délky 1, značeno $\Rightarrow$, je definován

$$x \Rightarrow y, \text{ pokud } x = x_1 u x_2,\ y = x_1 v x_2,\ \text{kde } x_1, x_2 \in (N \cup T)^* \\ \text{pro nějaké pravidlo } p : u \to v \in P.$$

Chceme-li explicitně vyjádřit podle jakého pravidla se derivační krok provedl, píšeme $x \Rightarrow y[p]$. Se znalostí definice derivačního kroku délky 1 můžeme definovat derivační krok délky n, značeno $\Rightarrow^n$, následovně:

$$x_0 \Rightarrow^n x_n,\ \text{pokud } x_0, \ldots, x_n \in (N \cup T)^*,\ n \geq 1,\ x_{i-1} \Rightarrow x_i \text{ podle } p \in P \text{ pro } \forall i : 1 \leq i \leq n.$$

Dále zavedeme derivace libovolné délky, tedy *reflexivní uzávěr* relace $\Rightarrow$, značený $\Rightarrow^+$, respektive *reflexivní a tranzitivní uzávěr* relace $\Rightarrow$, značený $\Rightarrow^*$, jsou definovány:

$$x \Rightarrow^+ y,\ \text{pokud } x \Rightarrow^n y \text{ pro } n \geq 1, \\ x \Rightarrow^* y,\ \text{pokud } x \Rightarrow^n y \text{ pro } n \geq 0.$$

Větnou formou rozumíme libovolný řetězec x_i nad $N \cup T$ odvoditelný z počátečního neterminálu. *Větou* označujeme větnou formu x_i, pro kterou platí $x_i \in T^*$. *Jazyk generovaný gramatikou* je množina $L(G) = \{w \in T^* | S \Rightarrow^* w\}$.

Zásobníkovým automatem (ZA) myslíme uspořádanou n-tici $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde Q je konečná množina *stavů*, Σ je *vstupní abeceda*, Γ je *zásobníková abeceda*, R je konečná množina *pravidel* ve tvaru $Apa \to wq$, kde $A \in \Gamma$, $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ a $w \in \Gamma^*$, $s \in Q$ je *počáteční stav*, $S \in \Gamma$ je *počáteční zásobníkový symbol* a $F \subseteq Q$ je množina *koncových stavů*. Symboly vstupní abecedy Σ označíme malými písmeny ze začátku symboly, typicky a, b, c, d, zatímco stavy označíme malými písmeny z konce abecedy, typicky p, q, r, s. Prvky abecedy Γ budeme značit velkými písmeny. *Výpočetní krok* ZA délky 1, značeno $\vdash$, je definován:

$$x \vdash y, \text{ pokud } x \to y \in R \wedge x = uApav \wedge y = uwqv,$$

$$\text{kde } u, w \in \Gamma^* A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\} \text{ a } v \in \Sigma^*.$$

Řetězce x a y nazýváme *konfigurace* ZA, formálně $x, y \in \Gamma^* Q \Sigma^*$. Chceme-li explicitně vyjádřit podle jakého pravidla se výpočetní krok provedl, píšeme $x \vdash y[p]$. Obdobně jako u gramatiky definujeme výpočetní krok délky n, zapsáno $\vdash^n$:

$$\begin{aligned} x_0 &\vdash^0 x_0, \text{ pokud } n = 0 \wedge x_0 \in \Gamma^* Q \Sigma^*, \\ x_0 &\vdash^n x_n, \text{ pokud } n \geq 1 \wedge x_0, \ldots, x_n \in \Gamma^* Q \Sigma^* \\ &\qquad \wedge \forall i \in \{1, \ldots, n\} : x_{i-1} \vdash x_i[r_i], \text{ kde } r_i \in R. \end{aligned}$$

Reflexivní uzávěr relace $\vdash$, značený $\vdash^+$, respektive *reflexivní a tranzitivní uzávěr* relace $\vdash$, značený $\vdash^*$, jsou definovány:

$$\begin{aligned} x &\vdash^+ y, \text{ pokud } x \vdash^n y \text{ pro } n \geq 1, \\ x &\vdash^* y, \text{ pokud } x \vdash^n y \text{ pro } n \geq 0. \end{aligned}$$

Jazyk přijímaný zásobníkovým automatem M *koncovým stavem*, značeno $L(M)_f$, jazyk přijímaný vyprázdněním zásobníku, značeno $L(M)_\varepsilon$, a jazyk přijímaný *koncovým stavem a vyprázdněním zásobníku*, značeno $L(M)_{f\varepsilon}$, jsou definovány:

$$\begin{aligned} L(M)_f &= \{w \mid w \in \Sigma^*, Ssw \vdash^* zf, z \in \Gamma^*, f \in F\}, \\ L(M)_\varepsilon &= \{w \mid w \in \Sigma^*, Ssw \vdash^* zf, z \in \varepsilon, f \in Q\}, \\ L(M)_{f\varepsilon} &= \{w \mid w \in \Sigma^*, Ssw \vdash^* zf, z \in \varepsilon, f \in F\}. \end{aligned}$$

Dále řekneme, že ZA je *deterministický*, pokud pro každé pravidlo $p = Apa \to wq \in R$ platí, že $R \setminus p$ neobsahuje žádné pravidlo s levou stranou Apa nebo Ap. *Rozšířený zásobníkový automat* je definován stejně jako ZA s tím rozdílem, že pravidla mohou nabývat tvaru $vpa \to wq$, kde $v, w \in \Gamma^*$, $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$.

Pomocí zkratek REG, LIN, CF, CS a RE si označíme rodiny *regulárních*, *lineárních*, *bezkontextových*, *kontextových* a *rekurzivně spočetných* jazyků. Označení MAT reprezentuje rodiny jazyků generovaných *maticovými gramatikami*. U těchto rodin jazyků nejsou využita ε-pravidla. Přítomnost ε-pravidel indikujeme pomocí ε psaného horním indexem (například CF^ε). Obdobně dolním indexem psané ac indikuje povolenou kontrolu výskytů. Takto dostáváme rodiny jazyků MAT_{ac}, MAT^ε, MAT^ε_{ac}. Dále označíme $ET0L$ a $EDT0L$ rodiny jazyků generovaných $ET0L$, respektive $EDT0L$ systémy.

2 Kooperačně distribuované gramatické systémy

Kooperačně distribuované (dále jen *CD*) gramatické systémy navazují na výše zmíněný model tabule. Tyto systémy pracují sekvenčně, jednotlivé gramatiky se střídají v práci na větné formě na základě kooperačního protokolu.

Definice 2.0.1. *Kooperačně distribuovaný gramatický systém*[7] je uspořádaná n-tice

$$\sigma = (N, T, S, P_1, P_2, \ldots, P_n),$$

kde N je abeceda *neterminálů*, T je abeceda *terminálů*, přičemž $N \cap T = \emptyset$. S je *počáteční neterminál* a $P_1, P_2, \ldots, P_n$ jsou konečné množiny *přepisovacích pravidel* nad abecedou $N \cup T$, rovněž nazývané jako *komponenty* systému σ.

Na jednotlivé komponenty se můžeme podívat jako na samostatné gramatiky sdílející stejnou abecedu – každá komponenta má vlastní seznam pravidel, které může aplikovat na větnou formu. Typickým rysem CD gramatického systému je aktivita právě jedné komponenty v každém okamžiku. Na základě kooperačního protokolu je vybrána některá komponenta a ta se stane *aktivní*. Aktivní komponenta pracuje na sdílené (jediné) větné formě dokud není splněna *ukončující podmínka*, poté se na základě kooperačního protokolu vybere další komponenta a ta se stane aktivní. Klíčovou roli hraje ukončující podmínka, po jejímž splnění znovu nastává výběr komponenty. V literatuře lze narazit na mnoho ukončujících podmínek, my se zaměříme na několik základních. U klasického CD gramatického systému sdílí všechny komponenty jednu jedinou ukončující podmínku – o takovém systému můžeme prohlásit, že všechny jeho komponenty pracují ve stejném módu. *Módem* budeme označovat způsob provádění derivačních kroků. Mód práce komponenty určuje, kdy může komponenta provést derivaci a jaká je délka derivace. Mód formálně definujeme jako prvek z množiny $D = \{t, *\} \cup \{=k, \geq k, \leq k \mid k \geq 1\}$. Nyní si formálně definujme derivační kroky prováděné v těchto módech – *ukončující derivaci*, derivaci dlouhou *právě* k kroků, *nejméně* k kroků a *nejvýše* k kroků.

Definice 2.0.2. *Ukončující derivace* i-té komponenty[7], značeno $\Rightarrow^t_{P_i}$, je definována:

$$x \Rightarrow^t_{P_i} y, \text{ pokud } x \Rightarrow^*_{P_i} y \text{ a zároveň neexistuje takové } z \in (N \cup T)^*, \text{ že } y \Rightarrow_{P_i} z.$$

Právě k kroků dlouhá derivace i-té komponenty, značeno $\Rightarrow^{=k}_{P_i}$, je definována:

$$x \Rightarrow^{=k}_{P_i} y, \text{ pokud existují taková } x_1, \ldots, x_{k+1} \in (N \cup T)^*, \text{ že platí } x = x_1, y = x_{k+1}$$
$$\text{a zároveň } x_j \Rightarrow_{P_i} x_{j+1} \text{ pro } \forall j : 1 \leq j \leq k.$$

Nejvýše k kroků dlouhá derivace i-té komponenty, značeno $\Rightarrow_{P_i}^{\leq k}$, je definována:

$$x \Rightarrow_{P_i}^{\leq k} y, \text{ pokud } x \Rightarrow_{P_i}^{=\overline{k}} y \text{ pro nějaké } \overline{k} \leq k.$$

Nejméně k kroků dlouhá derivace i-té komponenty, značeno $\Rightarrow_{P_i}^{\geq k}$, je definována:

$$x \Rightarrow_{P_i}^{\geq k} y, \text{ pokud } x \Rightarrow_{P_i}^{=\overline{k}} y \text{ pro nějaké } \overline{k} \geq k.$$

Neformálně řečeno, v módu $=k$ komponenta P_i provede na větné formě právě k derivačních kroků, kde krokem je myšlena aplikace přepisovacího pravidla. Mód $\leq k$ značí aplikaci nejvýše či právě k přepisovacích pravidel, zatímco mód $\geq k$ značí aplikaci alespoň k přepisovacích pravidel během derivace. Mód t značí užití ukončující derivace, tedy komponenta pracuje, dokud lze aplikovat nějaké přepisovací pravidlo. Mód $*$ označuje tranzitivní a reflexivní uzávěr relace $\Rightarrow$, tedy komponenta provádí derivační kroky tak dlouho, dokud se sama nerozhodne předat aktivitu další komponentě. V módu $*$ může komponenta provádět i derivace nulové délky, což lze využít především z teoretického hlediska.

Definice 2.0.3. *Jazyk generovaný CD gramatickým systémem* σ s n komponentami v módu $f \in D$ je podle [7] definován:

$$L_f(\sigma) = \left\{ w \in T^* | S \Rightarrow_{P_{i_1}}^{f} w_1 \Rightarrow_{P_{i_2}}^{f} \ldots \Rightarrow_{P_{i_m}}^{f} w_m = w,\ m \geq 1,\ 1 \leq i_j \leq n,\ 1 \leq j \leq m \right\}.$$

Do jazyka generovaného gramatických systémem spadá každý řetězec odvozený z počátečního neterminálu s užitím libovolných komponent. Všechny derivace jsou při odvozování prováděny ve stejném módu, jenž je explicitně uveden a není pevnou součástí systému. Z toho plyne, že jeden systém může generovat vícero jazyků v závislosti na zvoleném derivačním módu. Pro lepší porozumění CD gramatickým systémům si jejich funkčnost ověříme na příkladu.

Příklad 2.0.4. Uvažujme jazyk $L_1 = \{a^n b^n c^n | n \geq 1\}$, o němž je dobře známo, že nespadá do třídy bezkontextových jazyků, což se velmi snadno dokáže pomocí věty o iteraci. Pokud se pokusíme nalézt bezkontextovou gramatiku, která tento jazyk generuje, pak zajisté selžeme. Takovou bezkontextovou gramatiku nelze nalézt, na druhou stranu je možné sestrojit CD gramatický systém využívající pouze bezkontextových komponent, jež tento jazyk generuje. Zaměřme se na následující systém:

$$\begin{aligned}
\sigma_1 &= \big(\{S, A, \overline{A}, C, \overline{C}\},\ \{a, b, c\},\ AC,\ P_1,\ P_2\big),\\
P_1 &= \{1|2 : A \to a\overline{A}b \,|\, ab,\ \ 3|4 : C \to c\overline{C} \,|\, c\},\\
P_2 &= \{5|6 : \overline{A} \to aAb \,|\, ab,\ \ 7|8 : \overline{C} \to cC \,|\, c\}.
\end{aligned}$$

Nechť tento CD gramatický systém pracuje v módu $=2$, tedy každá komponenta provede vždy právě dva kroky. Systém provede sekvenci derivačních kroků následovně:

$$AC \Rightarrow_{P_1}^{=2} a\overline{A}bc\overline{C} \Rightarrow_{P_2}^{=2} aaAbbccC \Rightarrow_{P_1}^{=2} aaa\overline{A}bbbccc\overline{C} \Rightarrow_{P_2}^{=2} aaaabbbbcccc.$$

V uvedeném příkladu se komponenty v práci na větné formě pravidelně střídají, což je dáno strukturou ukázkového systému, nikoliv způsobem práce CD gramatických systémů. V tomto příkladu volba komponenty odpadá, protože lze vždy využít pouze jednu. V praxi

zcela jistě může nastat situace, kdy bude možné na větnou formu aplikovat více komponent. V takovém případě určí kooperační protokol, jaká z komponent se použije. Pozorný čtenář si rovněž všimne, že první komponenta využívá pravidla pouze v kombinacích 1-3 a 2-4, ačkoliv je zde $\binom{4}{2} = 6$ možností, jak pravidla zkombinovat. Zatímco kombinace 1-3 zachová počet neterminálů ve větné formě, kombinace 2-4 odstraní všechny neterminály z větné formy, tedy získáme větu. Pokud bychom využili kombinace 1-2, 1-4, 2-3 či 3-4, pak bychom z větné formy během derivace odstranili pouze jeden neterminál. Protože všechny komponenty pracují v módu $=2$, nelze samotný neterminál odstranit – nejméně 2 neterminály jsou nutné pro aplikaci některé z komponent. Jinak řečeno, využitím ostatních kombinací pravidel dostáváme větnou formu, ze které nelze odvodit větu. Naprosto analogické to je s druhou komponentou. S Při tomuto způsobu odvozování se zachovává počet neterminálů ve větné formě a poté opravdu platí $L_{=2}(\sigma_1) = L_1$.

Uvedený příklad nás přivádí k otázce síly CD gramatických systémů. Označme si $CD_n(f)$ rodiny jazyků generované CD gramatickými systémy o nejvýše n komponentách pracující v módu $f \in D$, přičemž neuvažujeme existenci ε-pravidel. Dále označením $CD_\infty(f)$ rozumíme systém s neomezeným počtem komponent. Následující věta shrnuje výsledky prezentované v [7].

Věta 2.0.5. *Generativní kapacitu CD gramatických systémů lze popsat relacemi:*

(i) $CD_\infty(f) = CF$, $\forall f \in \{=1, \geq 1, *\} \cup \{\leq k | k \geq 1\}$,

(ii) $CF = CD_1(f) \subset CD_2(f) \subseteq CD_r(f) \subseteq CD_\infty(f) \subseteq MAT$,
$\forall f \in \{=k, \geq k | k \geq 2\}$, $r \geq 3$,

(iii) $CD_r(=k) \subseteq CD_r(=sk)$, $\forall (k, r, s) \geq 1$,

(iv) $CD_r(\geq k) \subseteq CD_r(\geq k+1)$, $\forall (k, r) \geq 1$,

(v) $CD_\infty(\geq k) \subseteq CD_\infty(=k)$, $k \geq 1$,

(vi) $CF = CD_1(t) = CD_2(t) \subset CD_3(t) = CD_\infty(t) = ET0L$.

Při záměně inkluze $CD_\infty(f) \subseteq MAT$ za $CD^\varepsilon_\infty(f) \subseteq MAT^\varepsilon$, jsou všechny uvedené relace platné i pro systémy s ε-pravidly.

Z prvních dvou bodů věty 2.0.5 plyne několik významných poznatků. Systém o jedné komponentě (nezávisle na módu) je speciálním případem bezkontextové gramatiky a také poskytuje stejnou sílu. Užitím derivací v módech $=1$, ≥ 1 či $*$ generativní kapacitu rovněž nezvětšíme, nezávisle na počtu komponent – stejného chování bychom dosáhli vytvořením běžné bezkontextové gramatiky obsahující pravidla všech komponent. Hodnotné je zjištění, že ačkoliv je mód $\leq k$ zajímavý z teoretického hlediska, nevede ke zvýšení síly bezkontextových gramatik. Máme-li zadáno pouze horní omezení na délku derivace, pak můžeme vždy použít i derivaci délky 1, což degraduje chování takového systému na chování běžné bezkontextové gramatiky. Bod (ii) ukazuje, že jsme schopni se pomocí CD gramatických systémů přiblížit až k síle maticových gramatik. Podle (iii) a (iv) je možné zvýšit generativní kapacitu systému prodloužením derivace. Z bodu (v) vidíme, že jazyky generované systémy pracujícími v módu $=k$ jsou nadmnožinou jazyků generovaných v módu $\geq k$. Podle bodu (vi) lze zvýšit sílu i využitím systémů používající ukončující derivace, jež mají alespoň 3 komponenty. My se ve zbytku práce zaměříme především na derivace v módech $=k$ a t, neboť u nich lze snáze odstranit nedeterminismus a zároveň jsme jejich použitím schopni zvýšit generativní kapacitu gramatického systému.

2.1 Hybridní CD gramatické systémy

Všechny komponenty výše zmíněných systémů pracovaly ve stejném módu, tedy jednalo se homogenní CD gramatické systémy. Odstraníme-li požadavek na homogenitu, dostáváme takzvané *hybridní kooperačně distribuované gramatické systémy* (dále jen *HCD*). Dále se podíváme na to, v čem se liší chování těchto systémů a jaký to má důsledek na jejich generativní kapacitu.

Definice 2.1.1. Hybridní kooperačně distribuovaný gramatický systém[7] je uspořádaná n-tice

$$\sigma = \big(N, T, S, (P_1, f_1), \ldots, (P_n, f_n)\big),\ n \geq 1,$$

kde N je abeceda *neterminálů*, T je abeceda *terminálů*, S je *počáteční neterminál*. (P_1, f_1), $\ldots$, (P_n, f_n) jsou *komponenty* systémů, kde P_i, $1 \leq i \leq n$ jsou konečné množiny přepisovacích pravidel nad $N \cup T$ a $f_i \in D$ je mód, v němž i-tá komponenta systému pracuje.

Struktura HCD systému se od klasického CD systému liší pouze tím, že mód práce je specifikován pro každou komponentu zvlášť – každá komponenta může mít jinou ukončující podmínku. Způsob práce je obdobný jako u CD gramatického systému, ovšem generovaný jazyk nikoliv.

Definice 2.1.2. Jazyk generovaný HCD gramatickým systémem[7] s n komponentami je množina

$$L(\sigma) = \left\{ w \in T^* | S \Rightarrow_{P_{i_1}}^{f_{i_1}} w_1 \Rightarrow_{P_{i_2}}^{f_{i_2}} \ldots \Rightarrow_{P_{i_m}}^{f_{i_m}} w_m = w,\ m \geq 1, 1 \leq i_j \leq n,\ 1 \leq j \leq m \right\}.$$

Na rozdíl od klasického CD gramatického systému generuje HCD právě jeden jazyk, který je definován zvolenými módy jednotlivých komponent. Pro lepší pochopení je vhodné se rovnou podívat na ukázku práce takového systému.

Příklad 2.1.3. Uvažujme jazyk $L_{H1} = \{a^n b a^n b a^n \,|\, n \geq 0\}$, který je bezesporu kontextový. Máme zadán HCD gramatický systém, jenž tento jazyk generuje:

$$\begin{aligned}
\sigma_2 &= \big(\{S, A, \overline{A}\},\ \{a, b\},\ S,\ (P_1, =1),\ (P_2, =3),\ (P_3, =3),\ (P_4, t)\big),\\
P_1 &= \{S \to AAA\},\\
P_2 &= \{A \to a\overline{A}\},\\
P_3 &= \{\overline{A} \to aA\},\\
P_4 &= \{A \to b,\ \overline{A} \to b\}.
\end{aligned}$$

Tento systém nedeterministicky provede posloupnost derivací:

$$S \Rightarrow_{P_1}^{=1} AAA \Rightarrow_{P_2}^{=3} a\overline{A}a\overline{A}a\overline{A} \Rightarrow_{P_3}^{=3} aaAaaAaaA \Rightarrow_{P_2}^{=3} aaa\overline{A}aaa\overline{A}aaa\overline{A} \Rightarrow_{P_4}^{t} aaabaaabaaa.$$

V tomto případě jsou řetězce přijímány pouze poslední komponentou systému, která z větné formy odstraní všechny neterminály. Komponenty P_2 a P_3 se pravidelně střídají a generují nové terminály „a“ při zachování počtu neterminálů ve větné formě. První komponenta není nezbytně nutná, ale jejím vložením jsme se vyhnuli vložení takzvaného čekacího pravidla ve tvaru $A \to A$. Stejného výsledku bychom dosáhli, pokud bychom komponentu P_1 vypustili a rozšířili komponentu P_2 o pravidla $\{S \to S,\ S \to AAA\}$. Poté by bylo možné

odvodit AAA z S s využitím dvou čekacích kroků. Uvedený systému po provedení ukončující derivace již přijal řetězec a nemohl dále pracovat. To ovšem není pravidlem, v jistých případech, jak demonstruje následující příklad, lze po provedení ukončující derivace pokračovat v práci. Komponenta pracující s ukončující derivací dokonce nemusí ani přijímat řetězce, respektive být poslední v pořadí.

Příklad 2.1.4. Rozšíříme-li jazyk L_{H1} z příkladu 2.1.3, dostaneme jazyk $L_{H2} = \{a^n b^m a^{2n} b^m a^n \mid n, m \geq 0\}$, který je bezesporu kontextový. Odpovídají HCD gramatický systém je definován jako:

$$\begin{aligned}
\sigma_3 &= \big(\{S, A, \overline{A}, B, \overline{B}\},\ \{a, b\},\ S,\ (P_1, =2),\ (P_2, =2),\ (P_3, =2),\ (P_4, t)\big),\\
P_1 &= \{S \to S,\ S \to AA,\ B \to \varepsilon,\ \overline{B} \to \varepsilon\}\\
P_2 &= \{A \to a\overline{A}a,\ B \to b\overline{B}\},\\
P_3 &= \{\overline{A} \to aAa,\ \overline{B} \to bB\},\\
P_4 &= \{A \to B,\ \overline{A} \to B\}
\end{aligned}$$

Tento systém nedeterministicky provede posloupnost derivací:

$$\begin{aligned}
S &\Rightarrow^{=2}_{P_1} AA \Rightarrow^{=2}_{P_2} a\overline{A}aa\overline{A}a \Rightarrow^{=2}_{P_3} aaAaaaaAaa \Rightarrow^{=2}_{P_2} aaa\overline{A}aaaaaa\overline{A}aaa\\
&\Rightarrow^{t}_{P_4} aaaBaaaaaaBaaa \Rightarrow^{=2}_{P_2} aaab\overline{B}aaaaaab\overline{B}aaa\\
&\Rightarrow^{=2}_{P_3} aaabbBaaaaaabbBaaa \Rightarrow^{=2}_{P_1} aaabbaaaaaabbaaa.
\end{aligned}$$

Jednoduchým systémem jsme popsali kontextový a ke zpracování relativně složitý jazyk. V systému σ_3 hraje komponenta P_4 roli stavové podmínky, která se provede pouze jednou – ukončí se generování symbolů a a začnou se generovat symboly b. Jedná se o zajímavou alternativu k hlubokým zásobníkovým automatům.

Opět se dostáváme k otázce síly, kterou shrnuje následující věta z [7]:

Věta 2.1.5. *Generativní kapacitu HCD gramatických systémů lze popsat relacemi:*

(i) $CF = HCD_1 \subset HCD_2 \subseteq HCD_3 \subseteq HCD_4 = HCD_\infty \subset MAT_{ac}$,

(ii) $ET0L \subset HCD_4$,

(iii) $CD_\infty(=) \subset HCD_3$,

(iv) $CD_\infty(=) \subset HCD_\infty(fin - t) \subset (HCD_4 \cap MAT)$,

kde HCD_i označuje rodinu jazyků generovaných HCD gramatickými systémy i komponentách, pokud není počet komponent omezen, píšeme HCD_∞. $HCD_\infty(fin - t)$ značí rodiny jazyků generovaných HCD systémy obsahující konečný počet komponent pracujících v módu t.

Z (i) je na první pohled patrné, že počet komponent nevytváří nekonečnou hierarchii rodin jazyků. Uvedené relace ukazují, že HCD mají opravdu velikou generativní sílu. Podle (iii) stačí nám HCD systém o pouze třech komponentách, abychom popsali jazyky generované CD systémy o neomezeném počtu komponent a podle (iv) pouze 4 komponenty pro získání větší síly, než poskytují $ET0L$ systémy. Následující věta konstatuje sílu HCD v porovnání s CF – analogicky jakou u CD systému platí:

Věta 2.1.6. *Pokud HCD gramatický systém bez ε-pravidel σ splňuje následující vlastnosti, pak $L(\sigma) \in CF$:*

(i) σ neobsahuje komponentu pracující v módu $f \in \{=k, \geq k \mid k \geq 2\}$,

(ii) σ obsahuje nejvíce 2 komponenty pracující v módu t.

2.2 Systémy pracující v týmech

Doposud jsme uvažovali systémy, ve kterých pracovaly jednotlivé komponenty nezávisle na sobě. Aktivní komponenta byla vždy vybrána na základě kooperačního protokolu. Nyní komponenty sdružíme do větších celků – týmů, a podíváme se, jak se změnilo jejich chování.

Definice 2.2.1. CD gramatický systém s týmy o proměnlivé délce[7] je konstrukt

$$\sigma = (N, T, S, P_1, \ldots, P_n, R_1, \ldots, R_m),\ n, m \geq 1,$$

kde N je abeceda *neterminálů*, T je abeceda *terminálů* a S je *počáteční neterminál*. P_i, $1 \geq i \geq n$ jsou konečné množiny *přepisovacích pravidel* nad $N \cup T$, též zvané *komponenty* systému. R_j, $1 \geq j \geq n$ jsou podmnožiny $\{P_1, \ldots, P_n\}$ zvané *týmy*.

Jednoduše řečeno doplněním omezení, která určují, kdy lze použít kterou komponentu, do CD gramatického systému, získáváme systémy pracující v týmech (dále jen *PTCD*) poskytující ještě větší generativní sílu. Tyto systémy poskytují další možnosti, jak definovat vazby mezi pravidly. Derivace probíhá mírně odlišně od klasického CD gramatického systému.

Definice 2.2.2. Derivace provedená týmem $R_i = \{P_{j_1}, \ldots, P_{j_s}\}$, značeno $\Rightarrow_{R_j}$, je podle [7] definována:

$$x \Rightarrow_{R_j} y, \text{ pokud } x = x_1, A_1, x_2, A_2, \ldots, x_s, A_s, x_{s+1},\ y = x_1, y_1, x_2, y_2, \ldots, x_s, y_s, x_{s+1},$$
$$x_z, y_r \in (N \cup T)^*,\ 1 \leq z \leq s+1,\ A_r \to y_r \in P_{j_r},\ A_r \in N,\ 1 \leq r \leq s.$$

Z definice 2.2.2 je patrné, že týmová derivace o délce 1 je proveditelná, jestliže některá z komponent týmu obsahuje aplikovatelné pravidlo. Obdobným způsobem můžeme zavést derivace právě k kroků, nejméně k kroků, nejvíce k kroků či libovolný počet kroků dlouhé, značené $\Rightarrow^{=k}_{R_i}$, $\Rightarrow^{\geq k}_{R_i}$, $\Rightarrow^{\leq k}_{R_i}$ a $\Rightarrow^{*}_{R_i}$. Týmové pojetí nám umožňuje zavést několik variant ukončujících derivací.

Definice 2.2.3. Ukončující týmové derivace[7] značeny $\Rightarrow^{t_0}_{R_i}$, $\Rightarrow^{t_1}_{R_i}$ a $\Rightarrow^{t_2}_{R_i}$, jsou definovány:

$x \Rightarrow^{t_0}_{R_i} y$, pokud $x \Rightarrow^{\geq 1}_{R_i} y$ $\wedge$ neexistuje takové z, že $y \Rightarrow_{R_i} z$,
$x \Rightarrow^{t_1}_{R_i} y$, pokud $x \Rightarrow^{\geq 1}_{R_i} y$ $\wedge$ pro žádnou komponentu $P_{jr} \in R_i$ neexistuje z, že $x \Rightarrow_{P_{jr}} z$,
$x \Rightarrow^{t_2}_{R_i} y$, pokud $x \Rightarrow^{\geq 1}_{R_i} y$ $\wedge$ pro některou komponentu $P_{jr} \in R_i$
není derivace $y \Rightarrow_{P_{jr}} z$ proveditelná

Věta 2.2.3 zavádí tři varianty ukončujících derivací, respektive 3 módy práce komponent – t_0, t_1 a t_2. V módu t_0 komponenta ukončí svou činnost, jestliže nemůže tým jako celek

provést další krok. V módu t_1 končí práce na větné formě, až nemůže žádná komponenta systému provést další krok, zatímco v módu t_2 se končí, pokud nejméně jedna komponenta nemůže přepsat žádný symbol ve větné formě.

Jaká je tedy síla PTCD gramatických systémů? V [7] byly publikovány výsledky shrnuté větou 2.2.4.

Věta 2.2.4. *Generativní kapacitu PTCD gramatických systémů lze popsat relacemi:*

(i) $PT_cCD(f) = PT_\infty CD(f) = MAT$, $\forall\big(f \in \{*\} \cup \{\leq k, =k, \geq k \mid k \geq 1\}\big)$,

(ii) $PT_cCD = PT_\infty CD(f) = MAT_{ac}$, $\forall\big(s \geq 2 \wedge f \in t_0, t_1, t_2\big)$,

kde $PT_cCD(f)$ *značí rodiny jazyků generované PTCD systémy bez* ε*-pravidel,* c *udává konstantní počet komponent a* $f \in D^{PT}$, $D^{PT} = (D \setminus \{t\}) \cup \{t_0, t_1, t_2\}$ *mód práce systému. Pokud počet komponent není omezen, píšeme* $PT_\infty CD(f)$*. Stejné relace jsou zachovány i při práci s* ε*-pravidly, pouze dochází k záměně* MAT *za* MAT^ε*, respektive* MAT_{ac} *za* MAT^ε_{ac}*.*

Můžeme tedy konstatovat, že PTCD gramatické systémy jsou velice silné – mají stejnou generativní kapacitu jako samotné maticové gramatiky.

3 Paralelně komunikující gramatické systémy

Dalším typem gramatických systémů jsou systémy složené z paralelně pracujících gramatik (dále jen *PC* gramatické systémy). Tyto systémy navazují na tzv. třídní model zmíněný v kapitole 1. Systém se rovněž skládá z komponent tvořených množinami pravidel, ale na rozdíl od CD gramatických systémů má každá komponenta k dispozici svoji vlastní větnou formu. Všechny komponenty jsou aktivní zároveň a pro zajištění konzistence se komponenty synchronizují podle hodin, tj. každá komponenta provede v jednom taktu hodin nějakou derivaci. Komponenty mezi sebou mohou komunikovat a vyměňovat si informace, respektive větné formy.

Definice 3.0.1. PC gramatický systém[7] je uspořádaná n-tice

$$\sigma = \big(N, T, K, (S_1, P_1), \ldots, (S_n, P_n)\big),\ n \geq 1,$$

kde N je abeceda *neterminálů*, T je abeceda *terminálů*, $K = \{Q_1, Q_2, \ldots, Q_n\}$ je množina *dotazovacích symbolů*, přičemž $N \cap T \cap K = \emptyset$. Prvky $(S_1, P_1), \ldots, (S_n, P_n)$ budeme nazývat *komponenty* systému, kde P_i, $1 \leq i \leq n$ jsou konečné množiny *přepisovacích pravidel* nad $N \cup T \cup K$ a S_i je počáteční neterminál i-té komponenty. Přepisovací pravidla jsou ve tvaru $p : x \to y$, $x \in (N \cup T)^*$, $|x|_N \geq 1$, $y \in (N \cup T \cup K)^*$.

Definice 3.0.2. Konfigurací PC gramatického systému s n komponentami myslíme uspořádanou n-tici $(x_1, x_2, \ldots x_n)$, kde $x_i \in (N \cup T \cup K)^*$, $1 \leq i \leq n$.

Porovnáme-li strukturu PC gramatického systému s klasickou bezkontextovou gramatikou, nalezneme několik zásadních odlišností. Přibyla úplně nová abeceda – abeceda dotazovacích symbolů. Tyto symboly slouží ke komunikaci a umožňují výměnu informací mezi komponentami. Ke sdílení informací dochází poté, co některá komponenta zveřejní ve své větné formě dotazovací symbol. Tyto symboly se mohou objevit pouze na pravé straně pravidel. Konfigurace již není jediný řetězec, ale celá n-tice – každá komponenta má svůj vlastní konfigurační řetězec. Stejně tak má každá komponenta svůj vlastní počáteční neterminál. Abecedy N a T jsou ovšem společné.

Definice 3.0.3. Nechť $\chi_1 = (x_1, x_2, \ldots, x_n)$ a $\chi_2 = (y_1, y_2, \ldots, y_n)$ jsou 2 konfigurace PC gramatického systému σ a $\chi \notin T^*$. Potom řekneme, že PC gramatický systém provede derivační krok[7] z konfigurace χ_1 do konfigurace χ_2, zapsáno $\chi_1 \Rightarrow \chi_2$, pokud

(i) $\forall i : 1 \leq i \leq n$ platí $x_i \in (N \cup T)^*$ a zároveň $\forall i : 1 \leq i \leq n$ lze provést derivační krok $x_i \Rightarrow y_i$ s využitím některého pravidla z P_i nebo $x_i = y_i \in T^*$,

(ii) Existuje takové i, $1 \leq i \leq n$, že platí $|x_i|_K \geq 1$. Pro každé takové i, $x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_t Q_{i_t} z_{t+1}$, $t \geq 1$, kde $z_j \in (N \cup T)^*$, $1 \leq i, j \leq t+1$, $t \in \mathbb{N}$, nalezneme y_i. Pokud x_{i_j}, že $|x_{i_j}|_K = 0$ pro $\forall j : 1 \leq j \leq t$, pak $y_i = z_1 x_{i_1} z_2 1 x_{i_2} \ldots z_t x_{i_t} z_{t+1}$ a $y_{i_j} = S_{i_j}$. Pokud $|x_{i_j}|_K \geq 1$, pak $y_i = x_i$.

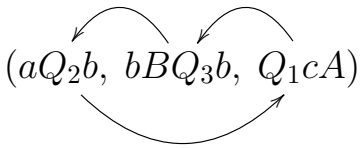
Definice 3.0.3 popisuje způsob odvozování věty PC gramatickým systémem. Jestliže konfigurace $(x_1, x_2, \ldots, x_n)$ neobsahuje žádný dotazovací symbol Q_i (bod (i)), pak systém provádí derivační kroky $x_i \Rightarrow y_i$ každou komponentou systému P_i samostatně – každá komponenta aplikuje na svoji větnou formu právě jedno přepisovací pravidlo. U systému se třemi komponentami by došlo ke třem náhradám, v každém řetězci konfigurace jedna. Vyskytuje-li se v konfiguraci dotazovací symbol, využije se bodu (ii), tedy provede se komunikační krok. Z definice rovněž plyne, že komunikační kroky mají přednost před derivačními. Dokud jsou v některém řetězci přítomny dotazovací symboly, nelze aplikovat další přepisovací pravidla. V komunikačním kroku jsou všechny výskyty dotazovacího symbolu Q_i, jež reprezentuje žádost o komunikaci s i-tou komponentou, nahrazeny současnou větnou formou i-té komponenty (řetězec x_i), vzápětí je větná forma i-té komponenty nahrazena jejím počátečním neterminálem. Poslední část bodu (ii) udává, že pokud řetězec x_i sám obsahuje dotazovací symboly, pak se tato náhrada neuskuteční a je pozdržena od dalšího kroku.

Z tohoto způsobu práce plyne nebezpečí uváznutí systému v mrtvém bodě. To může nastat pokud:

(i) v konfiguraci $(x_1, x_2, \ldots, x_n)$ není přítomen dotazovací symbol, řetězec x_i obsahuje neterminál, avšak žádné pravidlo z P_i nelze aplikovat, nebo

(ii) dojde k zacyklení komunikace.

Příkladem bodu (ii) je situace na obrázku 3.0.1, kdy P_i obsahuje Q_2, P_2 obsahuje Q_3, P_3 obsahuje Q_1. V takovém případě nemůže být proveden žádný komunikační krok a kvůli prioritě komunikace jsou zablokovány i možné derivační kroky.



Obrázek 3.0.1 Zacyklení PC gramatického systému během komunikačního kroku.

PC gramatické systémy můžeme rozčlenit na *centralizované* a *decentralizované*. Pro první jmenované, značené CPC, je typickým rysem, že dotazovací symboly produkuje pouze jedna (centrální) komponenta systému. U decentralizovaných systémů, značených DPC, je přítomno více komponent, jež mohou produkovat dotazovací symboly. Je zřejmé, že problém s možným zacyklením komunikace nastává pouze u decentralizovaných systémů.

PC gramatický systém může provádět derivace ve dvou módech – *vracejícím* (k axiomu) a *nevracejícím*. Ve vracejícím módu se provede komunikační krok přesně podle bodu (ii) definice 3.0.3 – nejprve je provedena náhrada všech výskytů dotazovacích symbolů Q_i řetězcem x_i a následně je samotný řetězec x_i nahrazen axiomem S_i. To znamená, že i-tá komponenta po poskytnutí své větné formy započne práci zcela od začátku. V nevracejícím módu se řetězec x_i axiomem nenahrazuje, tj. i-tá komponenta pokračuje v práci na stejné větné formě. Jinak řečeno, po provedení komunikačním kroku v nevracejícím módu zůstává

řetězec x_i v konfiguraci v nezměněné podobě. Tohoto chování docílíme, jestliže z bodu (ii) definice 3.0.3 vypustíme podmínku $y_{i_j} = S_{i_j}$. Derivace ve vracejícím módu budeme značit $\Rightarrow_r$, derivace v nevracejícím $\Rightarrow_{nr}$. Není překvapivé, že máme-li k dispozici 2 módy práce, je možné, aby jeden PC gramatický systém generoval 3 různé jazyky – s užitím vracejících, nevracejících a kombinovaných derivací.

Definice 3.0.4. Jazyk generovaný PC gramatickým systémem σ, značený $L(\sigma)$, jazyk generovaný PC gramatickým systémem σ ve vracejícím módu, značený $L_r(\sigma)$, a jazyk generovaný PC gramatickým systémem σ v nevracejícím módu, značený $L_{nr}(\sigma)$, jsou definovány[7]:

$$\begin{aligned} L(\sigma) &= \{x \in T^* \mid (S_1, S_2, \ldots, S_n) \Rightarrow^* (x, y_2, \ldots, y_n),\ y_i \in N \cup T \cup K,\ 2 \leq i \leq n\}, \\ L_r(\sigma) &= \{x \in T^* \mid (S_1, S_2, \ldots, S_n) \Rightarrow_r^* (x, y_2, \ldots, y_n),\ y_i \in N \cup T \cup K,\ 2 \leq i \leq n\}, \\ L_{nr}(\sigma) &= \{x \in T^* \mid (S_1, S_2, \ldots, S_n) \Rightarrow_{nr}^* (x, y_2, \ldots, y_n),\ y_i \in N \cup T \cup K,\ 2 \leq i \leq n\}. \end{aligned}$$

Z definice 3.0.4 je patrné, že jazyk generovaný první komponentou PC gramatického systému je ekvivalentní s jazykem generovaným celým systémem. Z toho důvodu se první komponenta označuje jako *master*. Ve chvíli, kdy je řetězec přijat první komponentou, není složení ostatních větných forem podstatné (mohou obsahovat neterminály). Činnost PC gramatického systému ověříme na příkladu.

Příklad 3.0.5. Uvažujme jazyk $L_1 = \{a^{2^n} \mid n \geq 1\}$. Nyní sestrojíme odpovídající PC gramatický systém, který generuje tento bezkontextový jazyk.

$$\begin{aligned} \sigma_4 &= \big(\{S_1, \overline{S_1}, S_2\}, \{a\}, \{Q_1, Q_2\}, (S_1, P_1), (S_2, P_2)\big), \\ P_1 &= \{S_1 \to Q_2Q_2Q_2Q_2\overline{S_1},\ S_1 \to S_1,\ \overline{S_1} \to \overline{S_1},\ \overline{S_1} \to \varepsilon\}, \\ P_2 &= \{S_2 \to S_2,\ S_2 \to Q_1,\ S_1 \to aaaa,\ \overline{S_1} \to \varepsilon\}. \end{aligned}$$

Čtenář si může snadno ověřit, že $L_r(\sigma_4) = L_1$. Systém nedeterministicky provede posloupnost derivací:

$$\begin{aligned} (S_1, S_2) &\Rightarrow_r (S_1, Q_1) \Rightarrow_r (S_1, S_1) \Rightarrow_r (Q_2Q_2Q_2Q_2\overline{S_1}, aaaa) \Rightarrow_r (a^{16}\overline{S_1}, S_2) \\ &\Rightarrow_r (a^{16}\overline{S_1}, Q_1) \Rightarrow_r (S_1, a^{16}\overline{S_1}) \Rightarrow_r (Q_2^4\overline{S_1}, a^{16}) \\ &\Rightarrow_r (a^{64}\overline{S_1}, S_2) \Rightarrow_r (a^{64}\overline{S_1}, Q_1) \Rightarrow_r (S_1, a^{64}\overline{S_1}) \\ &\Rightarrow_r (Q_2^4\overline{S_1}, a^{64}) \Rightarrow_r (a^{256}\overline{S_1}, S_2) \Rightarrow_r (a^{256}, S_2) \end{aligned}$$

Jak demonstruje ukázka při odvozování pomocí PC gramatických systémů hrají významnou roli tzv. čekací pravidla, bez nich by systém σ_4 přijímal pouze prázdný jazyk. Odvozování probíhá zcela nedeterministicky, takže je více způsobů (kombinací pravidel), jak se dobrat ke stejnému řetězci. Systém σ_4 z ukázky je decentralizovaný a pracuje ve vracejícím módu. V dalším příkladu si demonstrujeme práci centralizovaného systému.

Příklad 3.0.6. Uvažujme jazyk $L_2 = \{a^i b^j c^k | i \neq j,\ i \neq k,\ j \neq k,\ i, j, k \geq 3\}$. Tento jazyk, obsahující řetězce s různým počtem výskytů terminálů $\{a, b, c\}$, rovněž není bezkontextový. Odpovídající gramatický systém může být:

$$\begin{aligned} \sigma_5 &= \big(\{S_1, S_2, S_3, S_4, A, B, C\}, a, b, c, Q_2, Q_3, Q_4, (S_1, P_1), (S_2, P_2), (S_3, P_3), (S_4, P_4)\big), \\ P_1 &= \{S_1 \to ABC,\ A \to Q_2,\ B \to Q_3,\ C \to Q_4,\ A \to A,\ B \to B, \\ &\qquad C \to C,\ S_2 \to a,\ S_3 \to b,\ S_4 \to c\}, \\ P_2 &= \{S_2 \to aS_2\}, \end{aligned}$$

$P_3 = \{S_3 \to aS_3\}$,
$P_4 = \{S_2 \to aS_4\}$.

Tento systém nedeterministicky provede posloupnost derivací:

$$\begin{aligned}(S_1, S_2, S_3, S_4) &\Rightarrow_{nr} (ABC, aS_2, bS_3, cS_4) \Rightarrow_{nr} (AQ_3C, aaS_2, bbS_3, ccS_4)\\ &\Rightarrow_{nr} (AbbS_3C, aaS_2, bbS_3, ccS_4) \Rightarrow_{nr} (Q_2bbS_3C, aaaS_2, bbbS_3, cccS_4)\\ &\Rightarrow_{nr} (aaaS_2bbS_3C, aaaS_2, bbbS_3, cccS_4)\\ &\Rightarrow_{nr} (aaaS_2bbS_3C, aaaaS_2, bbbbS_3, ccccS_4)\\ &\Rightarrow_{nr} (aaaS_2bbS_3Q_4, aaaaaS_2, bbbbbS_3, cccccS_4)\\ &\Rightarrow_{nr} (aaaS_2bbS_3cccccS_4, aaaaaS_2, bbbbbS_3, cccccS_4)\\ &\Rightarrow_{nr} (aaaS_2bbS_3c^6, a^6S_2, b^6S_3, c^6S_4) \Rightarrow_{nr} (a^4bbS_3c^6, a^7S_2, b^7S_3, c^7S_4)\\ &\Rightarrow_{nr} (a^4b^3c^6, a^8S_2, b^8S_3, c^8S_4).\end{aligned}$$

Pomocí derivací v nevracejícím módu jsme odvodili řetězec $a^4b^3c^6 \in L_2$, pro názornost zkusíme nyní odvodit stejný řetězec ve vracejícím módu.

$$\begin{aligned}(S_1, S_2, S_3, S_4) &\Rightarrow_{r} (ABC, aS_2, bS_3, cS_4) \Rightarrow_{r} (AQ_3C, aaS_2, bbS_3, ccS_4)\\ &\Rightarrow_{r} (AbbS_3C, aaS_2, S_3, ccS_4) \Rightarrow_{r} (Q_2bbS_3C, aaaS_2, bS_3, cccS_4)\\ &\Rightarrow_{r} (aaaS_2bbS_3C, S_2, bS_3, cccS_4) \Rightarrow_{r} (aaaabbS_3C, aS_2, bbS_3, ccccS_4)\\ &\Rightarrow_{r} (aaaabbS_3Q_4, aaS_2, bbbS_3, ccccS_4) \Rightarrow_{r} (a^4b^2S_3c^5S_4, a^2S_2, b^3S_3, S_4)\\ &\Rightarrow_{r} (a^4b^3c^5S_4, a^3S_2, b^4S_3, cS_4) \Rightarrow_{r} (a^4b^3c^6, a^4S_2, b^5S_3, c^2S_4).\end{aligned}$$

Pozorný čtenář si může všimnout, že každý ze symbolů z K se může objevit ve větné formě pouze jednou při odvozování. Moment zveřejnění tohoto symbolu určuje délku iterace příslušného terminálu. Protože v jednom kroku může být zveřejněn pouze jeden dotazovací symbol, nemůže nastat situace, kdy $|a| = |b|$, $|a| = |c|$ nebo $|b| = |c|$. Zároveň tímto získáváme další zajímavou vlastnost: $L_r(\sigma_5) = L_{nr}(\sigma_5) = L_2$ – tento systém generuje stejný jazyk ve vracejícím i nevracejícím módu.

Nyní se podívejme na otázku síly PC gramatických systémů. Nejprve klasifikujeme PC gramatické systémy do několika skupin. Označením CPC rozumíme centalizované PC gramatické systémy. Přidáme-li písmeno „N“ před uvedené označení, pak tím rozumíme PC gramatické systémy pracující v nevracejícím módu – takto dostáváme systémy typu NPC a NCPC. Pro rodiny jazyků generované PC gramatickými systémy použijeme označení X_nY, kde $X \in \{PC, CPC, NPC, NCPC\}$ a $Y \in \{REG, LIN, CF, CS, CS^\lambda\}$. X značí povolený typ PC systému, Y značí typ použitých pravidel – po řadě regulární, lineární, bezkontextová či kontextová. Zkratka CS^λ připouští pravidla libovolného typu. $n \in \mathbb{N}$ značí stupeň sytému, respektive počet koponent. Pokud není počet komponent omezen, píšeme ∞. Výsledky z [7] shrnuje následující věta.

Věta 3.0.7. *Generativní kapacitu PC gramatických systému lze popsat relacemi:*

(i) $Y_nCS^\lambda = RE, \forall n \wedge Y \in PC, CPC, NPC, NCPC,$

(ii) $NPC_\infty CF \subseteq PC_\infty CF,$

(iii) $MAT \subset PC_\infty CF$,

(iv) $CPC_\infty REG \subset MAT_{fin}$,

(v) $Y_n CS = Y_\infty CS$, $\forall n \geq 1, Y \in \{CPC, NCPC\}$,

(vi) $CS = PC_1 CS = PC_2 CS \subset PC_3 CS = PC_\infty CS = RE$,

(vii) $CS = NPC_1 CS \subset NPC_2 CS = NPC_\infty CS = RE$.

Z věty 3.0.7 je na první pohled patrná velká generativní síla PC gramatických systémů. Za použití pouze bezkontextových pravidel dostáváme nástroj silnější, než jsou maticové gramatiky. Použitím kontextových pravidel můžeme ještě zvýšit sílu těchto systémů. Necentralizované systémy využívající kontextových pravidel o třech komponentách pracující ve vracejícím módu či systémy se dvěma komponentami pracující v nevracejícím módu jsou dostatečně silné k popisu rekurzivně spočetných jazyků.

Významnou vlastností výše zmíněných PC gramatických systémů je synchronizace podle hodin, které měří čas všem komponentám stejně. V každém hodinovém cyklu musí každá komponenta využít právě jedno pravidlo (pokud právě neprobíhá komunikace). Pokud tyto hodiny odstraníme, dostáváme *nesynchronizovaný PC gramatický systém*. To sice vede ke zjednodušení systému, neboť je možné vypustit čekací pravidla tvaru $A \to A$, $A \in N$, avšak v [7] je ukázáno, že desynchronizace zároveň snižuje generativní sílu.

3.1 PC gramatické systémy komunikující příkazem

V doposud zmiňovaných PC gramatických systémech komunikaci vždy inicioval příjemce zprávy tím, že ve své větné formě zavedl dotazovací symbol. Tento způsob komunikace se označuje jako *komunikace na žádost*. Neméně matematicky zajímavý je i obrácený případ, kdy komunikaci iniciuje odesilatel. Takovou komunikaci budeme nazývat *komunikace příkazem*, systémy tuto komunikaci využívající označíme *CCPC*.

Definice 3.1.1. PC gramatický systém komunikující příkazem [7] je uspořádaná n-tice $\sigma = \big(N, T, (S_1, P_1, R_1), \ldots, (S_n, P_n, R_n)\big)$, $n \geq 1$, kde N je abeceda *neterminálů*, T je abeceda *terminálů* a (S_i, P_i, R_i), $1 \leq i \leq n$ jsou *komponenty* systému, kde $S_i \in N$ je *počáteční neterminál*, P_i je množina *přepisovacích pravidel* nad $N \cup T$ a $R_i \subseteq (N \cup T)^*$ je *selektor jazyka* i-té komponenty.

Definice 3.1.2. Konfigurací CCPC gramatického systému[7] myslíme uspořádanou n-tici $(x_1, x_2, \ldots, x_n)$, kde $x_i \in N \cup T$, $1 \leq i \leq n$.

U systémů komunikujících příkazem je vypuštěna abeceda dotazovacích symbolů, roli komunikačního prostředku hraje selektor jazyka, jenž je definován pro každou komponentu zvlášť. Ten funguje jako filtr – určuje, o jaké zprávy má příjemce zájem. Příjemce při výměně informací obdrží pouze takové větné formy (zprávy), které spadají do jazyka specifikovaného jeho selektorem. Způsob odvozování je dosti odlišný od klasického PC gramatického systému, proto si jej rozebereme podrobněji.

Definice 3.1.3. Derivační krok provedený CCPC gramatickým systémem σ s n komponentami[7], značený $\Rightarrow$, je definován:

$$(x_1, \ldots, x_n) \Rightarrow (y_1, \ldots, y_n), \text{ pokud } \forall i : 1 \leq i \leq n \text{ platí } x_i \Rightarrow^* y_i \text{ a zároveň}$$
$$\text{neexistuje takové } z \in (N \cup T)^*, \text{ že } y_i \Rightarrow z_i.$$

Definice 3.1.4. Nechť $\delta(x_i, j)$ je funkce dvou proměnných, definovaná pro $1 \leq i, j \leq n$ jako

$$\delta(x_i, j) = \begin{cases} \varepsilon & \text{pro} x_i \notin R_j \vee i = j, \\ x_i & \text{pro} x_i \in R_j \wedge i \neq j. \end{cases}$$

Dále řekneme, že pro $1 \leq j \leq n$ je sekvence

$$\triangle(j) = \delta(x_1, j)\delta(x_2, j) \ldots \delta(x_n, j)$$

celková zpráva přijímaná j-tou komponentou. Rovněž pro $1 \leq j \leq n$ zavedeme *celkovou zprávu odesílanou* j-tou komponentou jako

$$\delta(j) = \delta(x_j, 1)\delta(x_j, 2) \ldots \delta(x_j, n).$$

Komunikační krok, značený $(x_1, \ldots, x_n) \vdash (y_1, \ldots, y_n)$, je pro $1 \leq i \leq n$ definován

$$y_i = \begin{cases} \triangle(i) & \text{pro } \triangle(i) \neq \varepsilon, \\ x_i & \text{pro } \triangle(i) = \varepsilon \wedge \delta(i) = \varepsilon, \\ S_i & \text{pro } \triangle(i) = \varepsilon \wedge \delta(i) \neq \varepsilon. \end{cases}$$

Derivační krok CCPC gramatického systému je již pro nás povědomý – každá komponenta provede na své vlastní větné formě ukončující derivaci. Tím je na rozdíl od systémů komunikujících na žádost vyloučeno, že systém se zasekne během derivačního kroku, neboť je vždy možné provést derivaci délky 0. Výměna informací probíhá decentralizovaně v komunikačním kroku. Každá komponenta může posílat zprávy všem ostatním (nikoliv sama sobě), respektive může obdržet zprávy od všech ostatních komponent. Pro každou komponentu systému se sestaví celková zpráva. Větné formy všech ostatních komponent (nikoliv větná forma příjemce) jsou porovnány se selektorem jazyka (filtrem) příjemce, a pokud splňují zadaný tvar, jsou přidány do celkové zprávy, jež se po prověření všech komponent odešle příjemci. Jinak řečeno, příjemce obdrží celkovou zprávu vytvořenou konkatenací všech větných forem ostatních komponent, které prošly jeho filtrem. Filtr může být tvořen například regulárním výrazem. Příjemce po obdržení neprázdné celkové zprávy nahradí svoji větnou formu obsahem zprávy. Pokud je komponenta odesilatelem zprávy, avšak žádnou zprávu sama nepřijímá, je její větná forma v komunikačním kroku nahrazena počátečním neterminálem. V posledním případě, kdy není komponenta ani příjemcem ani odesilatelem, zůstává její větná forma nezměněna. Podívejme se nyní, jak se kombinují komunikační a derivační kroky během odvozování.

Definice 3.1.5.

$$L(\sigma) = \Big\{ w \in T^* \mid (S_1, \ldots, S_n) \Rightarrow (x_1^{(1)}, \ldots, x_n^{(1)}) \vdash (y_1^{(1)}, \ldots, y_n^{(1)}) \Rightarrow (x_1^{(2)}, \ldots, x_n^{(2)}) \vdash (y_1^{(2)}, \ldots, y_n^{(2)}) \Rightarrow \ldots \Rightarrow (y_1^{(s)}, \ldots, y_n^{(s)}),\ s \geq 1, w = x_1^{(s)} \Big\}.$$

Jazyk generovaný CCPC gramatickým systémem je podle definice 3.1.5 sekvence derivačních a komunikačních kroků, které se pravidelně střídají. Pozornému čtenáři jistě neunikne, že systém se může zacyklit v případě, že se v komunikačním kroku nepošle ani jedna zpráva – to lze ovšem snadno detekovat a o zkoumané větné formě lze prohlásit, že nespadá do generovaného jazyka. Za povšimnutí stojí i fakt, že sekvence končí derivačním nikoli komunikačním krokem. Stejně jako u klasických PC systémů platí, že jazyk generovaný první komponentou (*master*) je rovněž jazykem generovaným celým systémem. Pro lepší porozumění, jak CCPC gramatický systém pracuje, si na příkladu ukážeme činnost takového systému.

Příklad 3.1.6. Uvažujme jazyk $L_{cc} = \{a^n b^m c^n d^m \mid n, m \geq 1\}$, který je kontextový. Odpovídající CCPC gramatický systém je definován jako

$$\sigma_6 = \big(\{S_1, \overline{S_1}, S_2, S_3, X, Y, M, T, U, V, W\}, \{a, b, c, d\}, (S_1, P_1, R_1), (S_2, P_2, R_2), (S_3, P_3, R_3)\big),$$
$$P_1 = \{S_1 \to S_1,\ S_1 \to NM,\ S_1 \to N\overline{S_1}M,\ \overline{S_1} \to X\overline{S_1},\ \overline{S_1} \to \overline{S_1}Y,$$
$$\overline{S_1} \to XY | X | Y,\ T \to a,\ U \to b,\ V \to c,\ W \to d\},$$
$$P_2 = \{S_2 \to S_2,\ X \to a,\ Y \to b,\ N \to T,\ M \to U\},$$
$$P_3 = \{S_3 \to S_3,\ X \to c,\ Y \to d,\ N \to V,\ M \to W\},$$
$$R_1 = \{Ta^*b^*U,\ Vc^*d^*W\},$$
$$R_2 = \{NX^*Y^*M\},$$
$$R_3 = \{NX^*Y^*M\}.$$

Systém nedeterministicky provede posloupnost derivací

$$(S_1, S_2, S_3) \Rightarrow (NXYYM, S_2, S_3) \vdash (S_1, NXYYM, NXYYM) \Rightarrow (S_1, TabbU, VcddW)$$
$$\vdash (TabbUVcddW, S_2, S_3) \Rightarrow (aabbbccddd, S_2, S_3).$$

Dostáváme se k otázce síly CCPC gramatických systémů. Zavedeme si označení $CCPC_nX$ pro rodiny jazyků generované CCPC gramatickými systémy o nejvýše n komponentách, kde $X \in \{REG, CF\}$ udává typ použitých pravidel. Pro CF uvažujme pouze použití bezkontextových pravidel neobsahujících ε. Pokud počet komponent není omezen, píšeme $CCPC_\infty X$. V [7] byly publikovány výsledky shrnuté větou 3.1.7.

Věta 3.1.7. *Generativní kapacitu PC gramatických systémů lze popsat relacemi:*

(i) $REG = CCPC_1REG = CCPC_2REG \subset CCPC_3REG = CCPC_\infty REG = CS,$

(ii) $CF = CCPC_1CF \subset CCPC_2CF = CCPC_\infty CF = CS.$

Získaná hierarchie rodin jazyků je pouze dvouúrovňová, překvapivě stejnou sílu dostaneme užitím regulárních či bezkontextových pravidel. K popisu kontextových jazyků nám stačí systémy o třech komponentách využívající pravě lineárních pravidel či systémy o dvou komponentách využívající bezkontextová pravidla neobsahující ε. Pokud povolíme ε-pravidla, dostáváme další zajímavé výsledky:

(i) $CS^\varepsilon \subseteq CCPC_3REG^\varepsilon$,

(ii) $CS^\varepsilon \subseteq CCPC_2CF^\varepsilon$.

Tedy přidáním ε-pravidel do CCPC gramatických systémů dostáváme nástroj silnější než samotné kontextové jazyky. Důkazy inkluzí jsou k nahlédnutí v [5].

4 Zvýšení generativní síly užitím PC gramatických systémů

Velikým cílem stojícím na počátku výzkumu gramatických systémů bylo zvýšení generativní síly bezkontextových gramatik, respektive možnost zpracovat kontextové jazyky za pomoci bezkontextových pravidel. Pro analýzu kontextových jazyků lze samozřejmě použít kontextové gramatiky, avšak této možnosti se dále věnovat nebudeme. Naopak se pokusíme využít toho, že na zpracování bezkontextových jazyků bylo navrženo mnoho relativně jednoduchých metod (např. LL prediktivní syntaktická analýza, LR syntaktická analýza, precedenční syntaktická analýza, metody obecné syntaktické analýzy…), ze kterých můžeme vycházet a pokusíme se zvýšit jejich generativní sílu. Pro detailní seznámení s těmito metodami může čtenář využít [1] nebo česky psanou alternativu [6].

4.1 Týmové PC gramatické systémy

Pro začátek vyjdeme z definice PC gramatických systémů ze sekce 3, kterou zavedením několika omezení přizpůsobíme pro LL prediktivní syntaktickou analýzu. První omezení budeme klást na komunikační symboly – dále se zaměříme jen na centralizované PC gramatické systémy. Z CD gramatických systémů převezmeme koncept rozdělení komponent do týmů a definujeme CPC gramatický systém s týmy (dále jen TCPC).

Definice 4.1.1. PC gramatický systém s týmy je uspořádaná n-tice

$$\sigma = \big(N, T, K, (S_1, P_1), \ldots, (S_n, P_n), R\big),\ n \geq 1,$$

kde N je abeceda *neterminálů*, T je abeceda *terminálů*, $K = \{Q_1, Q_2, \ldots, Q_n\}$ je množina *dotazovacích symbolů*, přičemž $N \cap T \cap K = \emptyset$. Prvky $(S_1, P_1), \ldots, (S_n, P_n)$ jsou *komponenty* systému, kde P_i, $1 \leq i \leq n$ jsou konečné množiny *přepisovacích pravidel* nad $N \cup T \cup K$ a S_i je počáteční neterminál i-té komponenty. R je množina týmů, přičemž platí $R \subseteq 2^P$, kde 2^P je potenční množina komponent definovaná pro $P = \{P_1, \ldots, P_n\}$.

Na rozdíl od klasického PC gramatického systému je tu navíc množina týmů, obsahující libovolně seskupené komponenty. První komponentu každého týmu budeme nazývat *leader* a pro přehlednost ji budeme značit $\triangleright P_i$ pro $1 \leq i \leq n$. Pozorný čtenář si zajisté povšimne, že PC gramatický systém je ekvivalentní s TCPC gramatickým systémem, jehož množina týmů R zahrnuje pouze jeden tým, který ovšem pokrývá všechny komponenty, tj. $R = \{\triangleright P_1, P_2, \ldots P_n\}$.

Obecný tvar přepisovacích pravidel je $p : x \to y$, $x \in (N \cup T)^*$, $|x|_N \geq 1$, $y \in (N \cup T \cup K)^*$. Pro efektivní práci s týmy se omezíme pouze na 4 typy pravidel:

1. *Komunikační pravidla*, která nabývají výše zmíněného tvaru, ale zároveň splňují podmínku $|x|_K \geq 1$, tj. obsahují alespoň jeden komunikační symbol. Tato pravidla může mít pouze centrální komponenta.

2. *Čekací pravidla*, tj, pravidla $A \to A$ pro každé $A \in N$.

3. *ε-pravidla*, tj. pravidla $A \to \varepsilon$ pro libovolné $A \in N$. Tato pravidla může mýt pouze centrální komponenta.

4. *Derivační pravidla* realizující odvozování jednotlivých komponent. Tato pravidla budou pouze v Graibachové normální formě, tj. nabývají tvaru $A \to xB$, kde $x \in T$ a $B \in N^\star$. Pro začátek budeme uvažovat pouze komponenty s jedním derivačním pravidlem.

Toto omezení můžeme zavést bez újmy na obecnosti, neboť libovolnou bezkontextovou gramatiku lze převést do Graibachové normální formy. Čekací pravidla jsme zavedli, aby byla možná práce v týmech. Budeme požadovat, aby zároveň pracovaly na větné formě všechny komponenty z *aktivního týmu* (každá pracuje na své vlastní větné formě). Ostatní komponenty, které nepatří do zvoleného aktivního týmu, jsou tzv. *neaktivní*. Definice PC gramatického systému ovšem vyžaduje, aby každá komponenta provedla v každém kroku právě jednu derivaci, čehož docílíme aplikací čekacích pravidel. Zavedeme omezení, že komponenty aktivního týmu nesmí využít čekací pravidlo, zatímco neaktivní komponenty vždy aplikují čekací pravidlo, a to konkrétně na nejlevější neterminál. Pro úplnost definujeme týmovou derivaci PC gramatického systému formálně:

Definice 4.1.2. Nechť $\chi_1 = (x_1, x_2, \ldots, x_n)$ a $\chi_2 = (y_1, y_2, \ldots, y_n)$ jsou 2 konfigurace TCPC gramatického systému σ. Řekneme, že konfigurace χ_1 derivuje konfiguraci χ_2 týmem $R_j \in R$, zapsáno $\chi_1 \Rightarrow^{R_j} \chi_2$, pokud platí:

$$(\forall P_i \in R_j \exists A \in N : x_i = uAv \wedge y_i = uzv \wedge A \to z \in P_i \wedge A \neq z) \wedge$$
$$(\forall P_i \in \{P_1, \ldots, P_n\} \setminus R_j : x_i = y_i \wedge \exists A \in N : A \to A \in P_i),$$

kde $u, v \in (N \cup T)^*$, $A \in N$, $z \in (N \cup T \cup K)^*$ a $1 \leq i \leq n$.

V tomto bodě je nutné zmínit, že zavedením týmů se nemění generativní kapacita PC gramatických systémů, ale zato se změní míra nedeterminismu a složitost gramatického systému. Týmy předem určují, které komponenty pracují souběžně a pouze eliminují značnou část větných forem, ze kterých nelze odvodit větu. Tuto situaci si ukážeme na příkladu. Uvažujme jazyky $L_8 = \{a^n b^m c^n d^m | n, m \geq 1\}$ a $L_9 = \{a^n b^m a^n b^m | n, m \geq 1\}$ a gramatické systémy:

$\sigma_8 = (\{S, A, B, C, D\}, \{a, b, c, d\}, \{Q_1, Q_2, Q_3, Q_4, Q_5\},$
$(S, P_1), (A, P_2), (B, P_3), (C, P_4), (D, P_5), \{R_1 = \{\triangleright P_1\}, R_2 = \{\triangleright P_2, P_4\}, R_3 = \{\triangleright P_3, P_5\}\})$,
$P_1 = \{S \to Q_2 Q_3 Q_4 Q_5 \,|\, S\}$,
$P_2 = \{A \to aA \,|\, A\}$,
$P_3 = \{B \to bB \,|\, B\}$,
$P_4 = \{C \to cC \,|\, C\}$,
$P_5 = \{D \to dD \,|\, D\}$,

$$\sigma_9 = (\{S, A, B\}, \{a, b\}, \{Q_1, Q_2, Q_3\}, (S, P_1), (A, P_2), (B, P_3)),$$
$$P_1 = \{S \to Q_2Q_3Q_2Q_3 \mid S, A \to a, B \to b\},$$
$$P_2 = \{A \to aA \mid A\},$$
$$P_3 = \{B \to bB \mid B\}.$$

Gramatický systém σ_9 generující jazyk L_9 nedeterministicky provede posloupnost derivací:

$$\begin{aligned}(S, A, B) &\Rightarrow_r (S, aA, bB) \Rightarrow_r (S, aaA, bB) \Rightarrow_r (S, aaaA, bB) \Rightarrow_r (Q_2Q_3Q_2Q_3, aaaaA, bbB)\\ &\vdash (aaaaAbbBaaaaAbbB, A, B) \Rightarrow_r (aaaaabbBaaaaAbbB, A, B)\\ &\Rightarrow_r (aaaaabbbaaaaAbbB, A, B) \Rightarrow_r (aaaaabbbaaaaabbB, A, B)\\ &\Rightarrow_r (aaaaabbbaaaaabbb, A, B).\end{aligned}$$

Posloupnost aplikace pravidel zde není nijak omezena a není zaručeno pořadí použití pravidel. Zaručeno je pouze, že bude dodržen počet terminálů n a m, ale k větě se lze dostat nekonečně mnoha způsoby (posloupnost lze vždy prodloužit použitím čekacích pravidel), což je velice nepříjemná vlastnost pro strojové zpracování. Ještě více by se situace zkomplikovala, pokud bychom zkoušeli generovat L_8 pomocí běžného PC gramatického systému. TCPC systémy v tomto směru generování zjednodušují. Pro σ_8 dostaneme následující posloupnost derivačních kroků:

$$\begin{aligned}(S, A, B, C, D) &\Rightarrow_r^{R_2} (S, aA, B, cCD) \Rightarrow_r^{R_2} (S, aaA, B, ccCD) \Rightarrow_r^{R_2} (S, aaaA, B, cccCD)\\ &\Rightarrow_r^{R_3} (S, aaaA, bB, cccCdD) \Rightarrow_r^{R_3} (S, aaaA, bbB, cccCddD)\\ &\Rightarrow_r^{R_1} (Q_2Q_3Q_4Q_5, aaaA, bbB, cccC, ddD) \vdash (aaaAbbBcccCddD, A, B, C, D)\\ &\Rightarrow_r^{R_1} (aaaabbBcccCddD, A, B, C, D) \Rightarrow_r^{R_1} (aaaabbbcccCddD, A, B, C, D)\\ &\Rightarrow_r^{R_1} (aaaabbbccccddD, A, B, C, D) \Rightarrow_r^{R_1} (aaaabbbccccddd, A, B, C, D).\end{aligned}$$

Zatímco v předchozím případě mohla komponenta P_3 používat pravidla zcela nezávisle na P_2, zde to už to neplatí (například P_2 derivovala a P_3 čekala). Je-li aktivní tým R_2, žádný člen tohoto týmů nesmí využít čekacího pravidla, a proto je nucena komponenta P_2 použít pravidlo $A \to aA$ a komponenta P_3 pravidlo $B \to bB$. Naopak neaktivní komponenty jsou nuceny použít čekací pravidla.

4.2 Syntaktická analýza založená na TCPC gramatických systémech

Způsob práce CPC, respektive TCPC, gramatických systémů je poměrně odlišný od práce klasické LL gramatiky, a proto budeme muset syntaktický analyzátor výrazně modifikovat, aby byl schopný pracovat s těmito systémy.

Nejprve zavedeme pro každou komponentu samostatnou LL tabulku a samostatný zásobník. Derivace bude přirozeně probíhat podle pravidla z tabulky na zásobníku příslušné aktivní komponenty. Pravidlo je nalezeno na základě vstupní lexémy a horního neterminálu na zásobníku. Aplikace pravidla je provedena běžným způsobem, avšak porovnání již klasicky dělat nelze. Vyskytne-li se na vrcholu zásobníku terminál, klasický analyzátor jej porovná se vstupní lexémou a v případě shody jej ze zásobníku odstraní. Toto chování

by ovšem v kombinaci s TCPC vyústilo ve ztrátu informace, což demonstruje následující příklad.

Příklad 4.2.1. Uvažujme CPC systém σ_{10} nad abecedou $T = \{a, \odot\}$ o dvou komponentách $P_1 = \{S \to S \,|\, Q_2 \odot Q_2 \odot Q_2, A \to \varepsilon\}$, $P_2 = \{A \to aA\}$ (CPC systém bez týmů je ekvivalentní s TCPC systémem, kde $R_1 = \{\triangleright P_1, P_2\}$). Jedna z možných derivací by mohla být:

$$\begin{aligned}(S, A) &\Rightarrow_r (S, aA) \Rightarrow_r (Q_2 \odot Q_2 \odot Q_2, aaA) \vdash (aaA \odot aaA \odot aaA, A) \\ &\Rightarrow_r (aa \odot aaA \odot aaA, aA) \Rightarrow_r (aa \odot aa \odot aaA, aaA) \Rightarrow_r (aa \odot aa \odot aa, aaaA).\end{aligned}$$

Pokud bychom po provedení derivace komponentou P_2 porovnaly vstup s vrcholem zásobníku (terminál a) a vrchol zásobníku odstranily, již nebudeme mít k dipozici informaci o zpracovaném vstupu, které je nutná pro komunikaci. Po dvou derivacích a porovnáních pomocí P_2 bychom komunikovali pouze část věty:

$$\begin{aligned}(S, A) &\Rightarrow_r (S, aA) \simeq (S, A) \Rightarrow_r (Q_2 \odot Q_2 \odot Q_2, aA) \simeq (Q_2 \odot Q_2 \odot Q_2, A) \vdash (A \odot A \odot A, A) \\ &\Rightarrow_r (\odot A \odot A, aA) \Rightarrow_r (\odot \odot A, aA) \Rightarrow_r (\odot\odot, aA),\end{aligned}$$

kde $\simeq$ značí porovnání prováděné syntaktickým analyzátorem a odstranění vrchního terminálu ze zásobníku. Ve skutečnosti by byl přijat pouze řetězec $aa \odot \odot$, který určitě nespadá do jazyka $\{a^n \odot a^n \odot a^n \,|\, n \geq 1\}$ generovaného σ_{10}.

Z výše zmíněných důvodů zavedeme omezení, že terminály ze zásobníku smí odstraňovat pouze centrální komponenta, což zamezí ztrátě informace. Zároveň je nutné i při práci ostatních komponent odebírat lexémy ze vstupu a porovnávat je s odpovídajícími terminály na zásobníku. Tuto situaci lze vyřešit pomocí značek terminálů. *Označený terminál*, značeno $\underline{a}$ pro $a \in T$, je při práci necentrální komponenty ignorován, tj. komponenta pracuje stejně, jako kdyby na zásobníku nebyl. Nechť P je komponenta, která není centrální. Pak P pracuje následujícím způsobem:

- P provede derivaci podle pravidla r z LL tabulky.
- Je-li P leader týmu, pak P bude porovnávat.
- Je-li na vrcholu zásobníku neoznačený terminál a a vstupní lexéma je a, pak P odstraní a ze vstupu (požádá o další token) a zároveň označí a na zásobníku jako $\underline{a}$. Při této operaci P ignoruje veškeré označené terminály na svém zásobníku.

Centrální komponenta P_c pracuje se značkami odlišně odlišně:

- P_c provede derivaci podle pravidla r z LL tabulky.
- Je-li na zásobníku označený terminál $\underline{a}$, P_c jej odstraní ze svého zásobníku.
- Je-li na zásobníku neoznačený terminál a, pak jej P_c porovná ze vstupem a v případě shody P_c odstraní a ze vstupu i ze svého zásobníku.
- Komunikuje-li P_c větnou formu pomocí jednoho komunikačního symbolu Q_k, pak:
  - první výskyt Q_k je nahrazen větnou formou k-té komponenty bez úpravy značení,
  - každý další výskyt Q_k je nahrazen větnou formou k-té komponenty ve které jsou všechny označené terminály nahrazeny za neoznačené terminály.

4.2.1 Výběr aktivního týmu

Před tím, než přejdeme k samotným LL tabulkám, je nutné ještě zavést funkci *výběru týmu*.

Definice 4.2.2. Funkce výběru týmu je relace $\tau : R \cup \{\emptyset, \geq\} \times T \cup \{\$\} \to R$, kde R je tým TCPC gramatického systému a T je abeceda terminálů.

τ je funkce dvou proměnných – na vstupu očekává poslední aktivní tým R či některý ze symbolů $\emptyset$, $\geq$. Symbol $\geq$ udává, který tým se má vybrat v prvním kroku odvozování (žádná komponenta nebyla doposud aktivní), zatímco symbol $\emptyset$ udává, že nezáleží na tom, který tým byl aktivní v předchozím kroku. Druhým vstupem funkce je aktuální vstupní lexéma. Výstupem je vždy tým, který má být použit k derivování. Funkci τ lze jednoduše vizualizovat tabulkou (tabulka 4.2.1). Například pro řádek 4 (R_1), sloupec 4 (c), je séman-

$R \diagdown T$	**a**	**b**	**c**	**d**	**$**
$\geq$	R_2				
$\emptyset$					
R_1			R_1	R_1	R_1
R_2	R_2	R_3			
R_3		R_3	R_1		

Tabulka 4.2.1 Příklad funkce výběru týmu ve formě tabulky pro jazyk $L_8 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$

tika tabulky následující: byl-li naposledy aktivní tým R_1 a je na vstupu lexéma c, pak bude v dalším kroku aktivní tým R_1, matematicky zapsáno: $\tau(R_1, c) = R_1$. Tento způsob popisu sice přesně vystihuje očekávané chování gramatického systému, ale pro zpracování jednodušších jazyků (včetně L_8) je zbytečně komplikovaný. Mnohdy se nepotřebujeme vůbec dívat na předchozí aktivní týmy a v takovém případě si můžeme dovolit ponechat v tabulce pouze řádek pro symbol $\emptyset$. Tím se funkce zjednoduší pouze na jeden parametr – vstupní lexému, tedy dostaneme relaci $T \cup \{\$\} \to R$. Zjednodušenou funkci zobrazuje tabulka 4.2.2. Například pro sloupec 4 (c) se sémantika pozměnila následovně: je-li na vstupu terminál c, další derivace bude provedena týmem R_1.

$R \diagdown T$	**a**	**b**	**c**	**d**	**$**
$\emptyset$	R_2	R_3	R_1	R_1	R_1

Tabulka 4.2.2 Příklad zjednodušené funkce výběru týmu ve formě tabulky pro jazyk $L_8 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$.

4.2.2 Konstrukce LL tabulek

Nyní se podívejme na samotnou strukturu LL tabulek pro jednotlivé komponenty. Je zřejmé, že pro úspěšnou konstrukci LL tabulky musí být každá komponenta systému sama o sobě LL gramatikou, tj. každá komponenta systému P_i musí splňovat podmínku:

$$A \to x \in P_i \wedge A \to y \in P_i \wedge x \neq y \Longrightarrow (x) \cap First(y) = \emptyset.$$

Takové komponenty budeme dále označovat jako *LL komponenty*. Situace, kdy lze z jednoho neterminálu odvodit dvě rozdílné větné formy začínající stejným terminálem je u LL komponent vyloučena. Buňky tabulky mohou pro každou komponentu nabývat hodnot:

- Pravidlo r – komponenta aplikuje r na svoji větnou formu.
- Identita id – komponenta aplikuje některé čekací pravidlo, tj. přepíše se nejlevější neterminál sebou samým. Z pohledu implementace není nutné činit žádnou akci.
- Prázdné pole – syntaktická chyba, řetězec není přijat komponentou a tedy ani gramatickým systémem.

Protože se tato upravená verze tabulky liší od klasické LL tabulky, budeme ji nadále nazývat jako **LLI** (*Left-most Left-to-right with Identity*)

Definice 4.2.3. Nechť $\sigma = (N, T, K, (S_1, P_1), \ldots (S_n, P_n), R)$ je TCPC gramatický systém. Pak LLI tabulka komponenty P_i, značená $T_{LLI}^{P_i}$, je definována jako

$$T_{LLI}^{P_i} = \{\alpha_{P_i}(A, a)\},$$

kde $\alpha_{P_i}(A, a) = p$, kde $a \in T \cup \{\$\}$, $A \in N$, $(p \in P_i \vee p = id)$ pro $1 \leq i \leq n$.

Před uvedením samotného algoritmu pro tvorbu LL tabulek si nejprve zavedeme několik predikátů:

$$\begin{aligned}
P_i \text{ is leader} &\iff \exists r \in R : r = \{\triangleright P_i, P_j, \ldots\} \text{ pro nějaké } 1 \leq i, j \leq n, \\
P_i \text{ is central} &\iff \exists r \in P_i : r = A \to x \wedge A \in N \wedge x \in (N \cup T \cup K)^* \wedge |r|_K \geq 1, \\
r \text{ is der-rule} &\iff r = A \to aX \wedge A \in N \wedge a \in T \wedge X \in N^*, \\
r \text{ is } \varepsilon\text{-rule} &\iff r = A \to \varepsilon \wedge A \in N, \\
r \text{ is com-rule} &\iff r = A \to x \wedge |x|_{Q_k} \geq 1 \text{ pro libovolné } Q_k \in K, \\
r \text{ is wait-rule} &\iff r = A \to A \wedge A \in N.
\end{aligned}$$

Tyto predikáty pouze matematicky vystihují to, co jsme již výše zavedli slovně. Pomocí predikátu „is leader“ můžeme ověřit, zda je komponenta vedoucím týmu, pomocí predikátu „is central“ ověříme, zdali se jedná o centrální komponentu, a pomocí predikátů „is der-rule“, „is ε-rule“, „is com-rule“ a „is wait-rule“ zjistíme, o který typ pravidla se jedná – po řadě derivační pravidlo, ε-pravidlo, komunikační pravidlo či čekací pravidlo.

Dále si pro každý terminál spočteme množinu $Team(a)$, která v sobě zahrnuje všechny týmy $r \in R$, které mohou být aktivní, je-li na vstupu terminál $a \in T$.

Definice 4.2.4. Nechť τ je funkce výběru týmu gramatického systému σ. Pak pro každé $a \in T \cup \{\$\}$ je definována množina $Team(a)$ jako:

$$\forall r \in R \cup \{\emptyset, \geq\} : r \in Team(a) \iff \tau(r, a) = \overline{r} \wedge \overline{r} \in R.$$

Pozorný čtenář si zajisté povšimne, že se jedná o množinu týmů uvedených v tabulce dané funkcí výběru týmu τ ve sloupci označeném terminálem a. Například pro výše uvedenou tabulku 4.2.1 platí $Team(a) = \{R_2\}$, $Team(b) = \{R_3\}$, $Team(c) = Team(d) = Team(\$) = \{R_1\}$. Pro výpočet množiny $Team$ využijeme prosté iterace přes τ, což realizuje algoritmus 4.2.1.

Algoritmus 4.2.1: Množina $Team$

Vstup: PC gramatický systém $\sigma = (N, T, K, (S_1, P_1), \ldots (S_n, P_n), R)$,
Výstup: Množina $Team(a)$ pro každé $a \in T \cup \{\$\}$

```
1: foreach a ∈ T ∪ {$} do
2:     Team(a) = ∅
3:     foreach r ∈ R ∪ {∅, ≥} do
4:         Team(a) := Team(a) ∪ τ(r, a)
5:     end foreach
6: end foreach
```

Další potřebnou prerekvizitou je množina $Utilize_N(P_i)$, což je množina všech neterminálů, které může využít komponenta P_i, tj. jsou na levé straně některého jejího pravidla. Formálně:

Definice 4.2.5. Nechť $\sigma = (N, T, K, (S_1, P_1), \ldots (S_n, P_n), R)$ je gramatický systém. Pak pro každou komponentu P_i je definována množina $Utilize_N(P_i)$ jako:

$$A \in Utilize_N(P_i) \iff A \in N \wedge A \to x \in P_i \text{ pro nějaké } x \in (N \cup T)^*.$$

Než se podíváme na konstrukci LLI tabulek pro jednotlivé komponenty, je nutné definovat množinu $First(x)$. Tato množina je zcela jistě většině čtenářů známá, ale pro ujasnění pojmu uvádíme definici:

Definice 4.2.6. Nechť $G = (N, T, S, P)$ je BKG. Pro každé $x \in (N \cup T)^*$ je definováno $First(x)$ [6] jako:

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

Jednoduše řečeno, množina $First(x)$ obsahuje všechny takové terminály, které se nacházejí na prvním místě ve větných formách odvozených z x pomocí gramatiky G. Obecnou definici množiny $First$ lze velice jednoduše upravit pro TCPC gramatické systémy – pravidla všech komponent sloučíme do jedné množiny pravidel P a pro tuto množinu poté sestavíme množinu $First$. Na komunikační symboly se díváme stejně, jako na neterminály, tedy platí $x \in (N \cup T \cup K)^*$.

Množinu $Follow$ zavádíme odlišně, než je běžné – zavádíme ji pouze pro centrální komponentu:

Definice 4.2.7. Nechť $\sigma = (N, T, K, (S_1, P_1), \ldots, (S_n, P_n), R)$ je TCPC gramatický systém. Pak $Follow_K^{P_i}$ definujeme jako:

$$\begin{aligned} Follow(A)_K^{P_i} \iff \big\{ & a \in T \,|\, S \Rightarrow_r^{P_i *} XAQ_{k_1}, \ldots, Q_{k_j} Z \\ & \wedge X, Z \in (N \cup T \cup K)^* \wedge A \in N \wedge Q_{k_1}, \ldots, Q_{k_j} \in K \\ & \wedge P_{k_1} \text{ is leader} \wedge \ldots \wedge P_{k_j} \text{ is leader} \\ & \wedge a \in First(Z)\big\} \text{ pro libovolné } j \geq 0. \end{aligned}$$

Lepší představu o množině $Follow_K^{P_i}$ dostaneme po neformálním výkladu. Představme si případ, kdy má komponenta P_1 komunikační pravidlo $S \to acMMQ_2Q_3Q_4cMQ_5a$. Víme, že komponenty P_2 a P_3 vystupují v některém týmu jako leader, přitom P_2 používá A a P_3

používá neterminál B. Nyní můžeme dosadit A za Q_2 do pravé strany pravidla a odstraníme všechny komunikační symboly následující bezprostředně za A, patří-li tyto symboly komponentě, která může vystupovat jako leader – dostáváme větnou formu $acMMAQ_4cMQ_5a$. Nyní položíme $Follow_K^{P_i}(A) = First(Q_4cMQ_5a)$. Poté postup opakujeme pro B – dosadíme B za Q_3, po odstranění komunikačních symbolů leaderů dostáváme $acMMQ_2BQ_4cMQ_5a$, a tedy $Follow_K^{P_i}(B) = First(Q_4cMQ_5a)$. Tento postup přesně odpovídá dekompozici, kterou nám udává definice $Follow$.

Pro konstrukci LLI tabulek musíme mít k dispozici množinu $Follow_K^{P_c}(A)$ pro každé $A \in Utilize_N(P_c)$, kde P_c je centrální komponenta gramatického systému. Konečně můžeme přejít k samotnému algoritmu pro vytvoření LLI tabulky (algoritmus 4.2.2). Vstupní řetězec lze rozdělit na úseky a s těmi lze pracovat samostatně, přičemž z každého úseku získáme samostatnou LLI tabulku. Toto rozdělení nám samo o sobě zajišťuje funkce τ, proto budeme iterovat přes všechny možné dvojice $(\text{tým}, \text{terminál})$. Jiným způsobem než k ostatním členům týmu se musíme chovat ke komponentě leader. Zatímco leader se snaží svými derivacemi získat symbol shodující se se vstupní lexémou, u ostatních komponent toto platit nemusí. Leader vždy aplikuje takové pravidlo $r : A \to x$, pro které platí, že $First(x)$ se shoduje se vstupní lexémou, tedy chování leadera je naprosto shodné s chováním klasické LL gramatiky. Další členové týmu nemohou využít stejný princip, neboť musí aplikovat nějaké pravidlo souběžně s leaderem týmu, ale terminál, který se komponenta snaží odvodit, není v momentě aplikace pravidla na vstupu. Pro lepší pochopení uvádíme příklad 4.2.8.

Příklad 4.2.8. Uvažujme jazyk $L = \{a^n b^n a^n \mid n \geq 1\}$ a komponenty $P_1 = \{S \to S \mid Q_2Q_3Q_2, A \to \varepsilon, B \to \varepsilon\}$, $P_2 = \{A \to aA\}$ a $P_3 = \{B \to bB\}$. Máme 2 týmy: $R = \{\{P_1\}, \{P_2, P_3\}\}$. Je-li na vstupu lexéma a, bude komponenta P_2 aplikovat pravidlo $A \to aA$, takže skutečně platí $First(aA) = a$. Aby byl zachován stejný počet terminálů a a b, musí zároveň pracovat i komponenta P_3 (je ve stejném týmu jako P_2). Komponenta P_3 ovšem musí aplikovat pravidlo $B \to bB$, neboť čekací pravidlo nesmí použít a jiné pravilo nemá k dispozici. Komponenta P_3 tedy na základě vstupní lexémy a odvodila terminál b.

Algoritmus 4.2.2: LLI Tabulka

Vstup: CD gramatický systém $\sigma = (N, T, K, (S_1, P_1), \dots (S_n, P_n), R)$,
množina $First(x)$ pro každé $x \in (N \cup T)$,
množina $Follow(X)$ pro každé $X \in N$
Výstup: LLI tabulka ve tvaru $T_{LLI}^{P_i} = \{\alpha(A \in N, a \in T)\}$
pro každou komponentu P_i, $1 \leq i \leq n$.

1: **foreach** $a \in T \cup \{\$\}$ **do**
2: **foreach** $team \in Team(a)$ **as** t **do**
3: **foreach** $member \in t$ **as** m **do**
4: **if** m **is** leader **then**
5: **foreach** $A \to x \in \triangleright m$ **as** r **do**
6: **if** r **is** der-rule **and** $First(x) = a$ **then**
7: $\alpha_{\triangleright m}(A, a) := r$ ▷ Derivace – současný vstup
8: **end if**
9: **end foreach**
10: **else** ▷ Paralelní derivace – očekávaný vstup
11: assignRules(m, T)
12: **end if**
13: **if** m **is** cental **then**
14: **foreach** $rule : A \to x \in m$ **as** r **do**
15: **if** r **is** com-rule **then**
16: $\overline{x} = Pass(r)$
17: $\alpha_m(A, First(\overline{x})) = r$ ▷ Přiřazení komunikačních pravidel
18: **else if** r **is** ε-rule **then**
19: **foreach** $b \in Follow_K^m(A)$ **do**
20: $\alpha_m(A, b) = r$ ▷ Přiřazení ε-pravidel
21: **end foreach**
22: **end if**
23: **end foreach**
24: **end if**
25: **end foreach**
26: **foreach** $nonmember \in P \setminus t$ **as** $\neg m$ **where** $P = \{P_1, \dots, P_n\}$ **do**
27: **foreach** $A \in Utilize_N(\neg m)$ **do**
28: $\alpha_{\neg m}(A, a) := id$ ▷ Přiřazení čekacích pravidel
29: **end foreach**
30: **end foreach**
31: **end foreach**
32: **end foreach**

Výběr správného pravidla pro komponenty, které nejsou leader, je komplexní problém. Tento výběr je v algoritmu 4.2.2 zahrnut jako procedura „assignRules“. Nejjednodušší způsob, který není zdaleka optimální, je tuto proceduru implementovat jako klasický algoritmus tvorby LL tabulky (nezávisle na týmech). V takovém případě projdeme všechna pravidla a všechny terminály přičemž pravidla přiřadíme podle ekvivalence $\alpha_{P_i}(A, a) = r \iff A \to x \in r \wedge a \in First(x)$, tedy pole tabulky pro kombinaci A a a bude obsahovat pravidlo právě pokud z A můžeme odvodit a. Tento způsob vyplňování tabulek podřízených komponent povede k mnohonásobnému vyplňování tabulky, ale nebude žádné pravidlo opomenuto. Zá-

vislosti mezi komponentami jsou zde zanedbány, ale tuto vazbu bude implementovat přímo algoritmus TCPC syntaktické analýzy.

Ve chvíli, kdy jsme přiřadili odpovídající pravidla všem členům týmu, je nutné ještě všem komponentám, které nejsou v týmu, přiřadit čekací pravidla. Čekací pravidla má ovšem smysl uvádět pouze pro neterminály, které komponenta může nějakým způsobem odvodit, tj. vyskytují se na levé straně některého pravidla. Takovéto neterminály udává množina $Utilize_N$.

Tím jsme pokryli pouze derivační a čekací pravila, ještě je nutné se vypořádat s ε-pravidly a komunikačními pravidly. Pro práci s komunikačními pravidly zavedeme operátor $Pass(r)$ (pro každé $r \in P_i$), který funguje následovně:

- pro komunikační pravidlo $r = A \to x$ pracuje $Pass(r)$ nad pravou stranou pravidla x, kde $x \in (T \cup N \cup K)^*$,
- s pomocí τ sestavíme množinu týmů, které byly aktivní před komponentou vlastnící komunikační pravidlo (centrální komponenta),
- z množiny týmů odstraníme centrální komponentu,
- z každého týmu vezmeme leadera $\triangleright P_i$,
- pro každého nalezeného leadera $\triangleright P_i$ projdeme x a odstraníme první nalezený výskyt komunikačního symbolu Q_i,
- vrátíme x.

Jednoduše řečeno, operátor $Pass$ z komunikačního pravidla odstraní veškeré symboly, které již odvodila některá z jiných komponent. Vyskytuje-li se v pravidle komunikační symbol komponenty leader vícekrát, pak je odstraněn pouze jeho první výskyt – další výskyty doposud nebyly zpracovány a budou zpracovány centrální komponentou. Toto chování koresponduje se způsobem značení terminálů analyzátorem – označené terminály jsou odstraněny po provedení komunikace, tj. nehrají v odvozování centrální komponentou žádnou roli.

Příklad 4.2.9. Uvažujme jazyk $L = \{a^n \imath^n a^n \imath^n \mid n \geq 1\}$ a korespondující TCPC gramatický systém:

$$\begin{aligned}
\sigma &= (\{S, A, I\}, \{a, \imath\}, \{Q_1, Q_2, Q_3\}, (S, P_1), (A, P_2), (I, P_3), R), \\
R &= \{R_1 = \{\triangleright P_1\}, R_2 = \{\triangleright P_2, P_3\}\}, \\
P_1 &= \{S \to S \mid Q_2 Q_3 Q_2 Q_3, A \to \varepsilon, I \to \varepsilon\}, \\
P_2 &= \{A \to A \mid aA\}, \\
P_3 &= \{I \to I \mid \imath I\}.
\end{aligned}$$

Tento systém proveden na základě vstupu $aa\imath\imath aa\imath\imath$ provede derivace

$$(S, A, I) \Rightarrow_r^{R_2} (S, \underline{a}A, \imath I) \Rightarrow_r^{R_2} (Q_2 Q_3 Q_2 Q_3, \underline{aa}A, \imath\imath I)$$

Nyní je na řadě komunikace. Komponenta P_2 je leader, takže můžeme spoléhat na to, že část vstupu aa je již zpracovaná. Na základě lexémy $\imath$ na vstupu bychom měli aplikovat

komunikační pravidlo centrální komponenty. K tomu je přesně určen operátor $Pass$, neboť právě ten odstraní z pravidla již zpracovanou část vstupu, tedy:

$$Pass(S \to Q_2Q_3Q_2Q_3) = Q_3Q_2Q_3,$$

protože $\triangleright P_2$ je leader, ale P_3 není.

Po komunikaci dostaneme větnou formu $(\underline{aa}A\imath\imath IaaA\imath\imath I, A, I)$, ze které odstraněním označených terminálů vznikne $(A\imath\imath IaaA\imath\imath I, A, I)$. Po aplikaci ε-pravidel pomocí R_1 získáme zadaný vstupní řetězec.

Poslední částí LLI tabulky jsou ε-pravidla, která může vlastnit pouze centrální komponenta. ε-pravidlo využijeme ve všech případech, kdy má centrální komponenta na zásobníku neterminál A a na vstupu je lexéma a, pro které platí $Follow(A) = a$. Při hlubším zkoumání vyšlo najevo, že tento postup selže pro nedeterministické jazyky, což ovšem z praktického hlediska nečiní veliký problém – lze upravit gramatický systém (přidáme nějaký terminál v roli separátoru).

4.2.3 Ukázka konstrukce LLI bulek

Uvažujme jazyk $L_8 = \{a^n b^m c^n d^m | n, m \geq 1\}$ a TCPC gramatický systém σ_8, který tento jazyk přijímá:

$$\begin{aligned}
&\sigma_8 = (\{S, A, B, C, D\}, \{a, b, c, d\}, \{Q_1, Q_2, Q_3, Q_4, Q_5\},\\
&\quad (S, P_1), (A, P_2), (B, P_3), (C, P_4), (D, P_5), \{R_1 = \{\triangleright P_1\}, R_2 = \{\triangleright P_2, P_4\}, R_3 = \{\triangleright P_3, P_5\}\})\\
&P_1 = \{1, 2 : S \to Q_2Q_3Q_4Q_5 \,|\, S, 3 : A \to \varepsilon, 4 : B \to \varepsilon, 5 : C \to \varepsilon, 6 : D \to \varepsilon\},\\
&P_2 = \{7, 8 : A \to aA \,|\, A\},\\
&P_3 = \{9, 10 : B \to bB \,|\, B\},\\
&P_4 = \{11, 12 : C \to cC \,|\, C\},\\
&P_5 = \{13, 14 : D \to dD \,|\, D\}.
\end{aligned}$$

Funkce výběru týmů je definována podle tabulky 4.2.2, tedy:

$$\begin{aligned}
\tau(\emptyset, a) &= R_2,\\
\tau(\emptyset, b) &= R_3,\\
\tau(\emptyset, c) &= \tau(\emptyset, d) = \tau(\emptyset, \$) = R_1.
\end{aligned}$$

Aplikujeme algoritmus 4.2.2 – začneme s terminálem a, respektive s dvojicí (a, R_2):

i.	$T \cup \{\$\} = \{a, b, c, d, \$\}$
ii.	$term = a$
iii.i	$teams = Team(a) = \{R_2\} = \{\{\triangleright P_2, P_3\}\}$
iii.ii	$team = R_2 = \{\triangleright P_2, P_3\}$
iii.iii.i	$member = \triangleright P_2 = \{A \to aA\}$
iii.iii.ii	$\triangleright P_2$ is leader
iii.iii.iii	$r = A \to aA$
iii.iii.iv	$First(aA) = a : \alpha_{\triangleright P_2}(A, a) = r[7]$
iii.iii.v	$member = P_4 = \{C \to cC\}$
iii.iii.vi	not P_4 is leader
iii.iii.vii	assignRule$(P_4, T) : \alpha_{P_4}(C, T) = r[11]$
iii.iv	$nonmembers = \{P_1, P_3, P_5\}$
iii.v.i	$nm = P_1$
iii.v.ii	$Utilize_N(P_1) = \{S, A, B, C, D\}$
iii.v.iii	$\alpha_{P_1}(S, a) = id$
iii.v.iv	$\alpha_{P_1}(A, a) = id$
iii.v.v	$\alpha_{P_1}(B, a) = id$
iii.v.vi	$\alpha_{P_1}(C, a) = id$
iii.v.vii	$\alpha_{P_1}(D, a) = id$
iii.v.viii	$nm = P_3$
iii.v.ix	$Utilize_N(P_3) = \{B\}$
iii.v.x	$\alpha_{P_3}(B, a) = id$
iii.v.xi	$nm = P_5$
iii.v.xii	$Utilize_N(P_5) = \{D\}$
iii.v.xiii	$\alpha_{P_5}(D, a) = id$

Pro terminál b, respektive dvojici (b, R_3) je postup stejný, přičemž získáme následující pravidla:

$$\begin{array}{lll} \alpha_{\triangleright P_3}(B, b) = 9 & \alpha_{P_5}(D, b) = 13 & \alpha_{P_1}(S, b) = id \\ \alpha_{P_1}(A, b) = id & \alpha_{P_1}(B, b) = id & \alpha_{P_1}(C, b) = id \\ \alpha_{P_1}(D, b) = id & \alpha_{P_2}(A, b) = id & \alpha_{P_4}(C, b) = id \end{array}$$

Jinak se bude algoritmus chovat pro komponentu P_1 (dvojice (P_1, c)), která je sama v týmu a zároveň je centrální komponentou.

iv.	$term = c$
v.i	$teams = Team(c) = \{R_1\} = \{\{\triangleright P_1\}\}$
v.ii	$team = R_1 = \{\triangleright P_1\}$
v.iii.i	$member = \triangleright P_1 = \{S \to Q_2Q_3Q_4Q_5, A \to \varepsilon, B \to \varepsilon, C \to \varepsilon, D \to \varepsilon\}$
v.iii.ii	$\triangleright P_1$ is leader
v.iii.iii	v $\triangleright P_1$ není derivační pravidlo
v.iii.iv	$\triangleright P_1$ is central
v.iii.v	$r = S \to Q_2Q_3Q_4Q_5$
v.iii.vi	r is com-rule
v.iii.vii	$\overline{x} = Pass(S \to Q_2Q_3Q_4Q_5) = Q_4Q_5$
v.iii.viii	$\alpha_{\triangleright P_1}(S, First(\overline{x})) = \alpha_{\triangleright P_1}(S, c) = r[1]$
v.iii.ix	$r = A \to \varepsilon$
v.iii.x	r is ε-rule
v.iii.xi	$Follow_K^{\triangleright P_1}(A) = \{c\}$
v.iii.xii	$\alpha_{\triangleright P_1}(A, c) = r[3]$
v.iii.xiii	$r = B \to \varepsilon$
v.iii.xiv	r is ε-rule
v.iii.xv	$Follow_K^{\triangleright P_1}(B) = \{c\}$
v.iii.xvi	$\alpha_{\triangleright P_1}(B, c) = r[4]$
v.iii.xvii	$r = C \to \varepsilon$
v.iii.xviii	r is ε-rule
v.iii.xix	$Follow_K^{\triangleright P_1}(C) = \{d\}$
v.iii.xx	$\alpha_{\triangleright P_1}(C, d) = r[5]$
v.iii.xxi	$r = D \to \varepsilon$
v.iii.xxii	r is ε-rule
v.iii.xxiii	$Follow_K^{\triangleright P_1}(D) = \{\$\}$
v.iii.xxiv	$\alpha_{\triangleright P_1}(D, \$) = r[6]$

A samozřejmě i pro komponentu P_1 získáváme následující pravidla pro identitu:

$$\begin{array}{llll} \alpha_{P_2}(A,c) = id & \alpha_{P_3}(B,c) = id & \alpha_{P_4}(C,c) = id & \alpha_{P_5}(D,c) = id \\ \alpha_{P_2}(A,d) = id & \alpha_{P_3}(B,d) = id & \alpha_{P_4}(C,d) = id & \alpha_{P_5}(D,d) = id \\ \alpha_{P_2}(A,\$) = id & \alpha_{P_3}(B,\$) = id & \alpha_{P_4}(C,\$) = id & \alpha_{P_5}(D,\$) = id \end{array}$$

Další dvojice (P_1, d) a $(P_1, \$)$ už nám nová pravidla nepřinesou (pouze znovu vyprodukují stejná pravidla, jako dvojice (P_1, c), kterou jsme již zpracovali výše), tedy konečná podoba LLI tabulek je dána tabulkou 4.2.3.

P_1	**a**	**b**	**c**	**d**	**$**
S	id	id	1		
A	id	id	3		
B	id	id	4		
C	id	id		5	
D	id	id			6

P_2	**a**	**b**	**c**	**d**	**$**
A	7	id	id	id	id

P_4	**a**	**b**	**c**	**d**	**$**
B	11	id	id	id	id

P_3	**a**	**b**	**c**	**d**	**$**
C	id	9	id	id	id

P_5	**a**	**b**	**c**	**d**	**$**
D	id	13	id	id	id

Tabulka 4.2.3 LLI tabulky pro gramatický systém σ_8 generující jazyk $L_8 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$.

4.3 TCPC syntaktický analyzátor

V této chvíli máme vše potřebné k vysvětlení činnosti syntaktického analyzátoru založeného na TTCP gramatickém systému. Z algoritmu vyčleníme funkci výběru týmu, kterou implementujeme samostatně v podobě algoritmu 4.3.3. Výběr následníka je jednoduchý – je-li

Algoritmus 4.3.3: Výběr týmu: procedura *selectTeam*

Vstup: TCPC gramatický systém $\sigma = (N, T, K, (S_1, P_1), \ldots, (S_n, P_n), R)$, poslední aktivní tým R_{Last} a aktuální vstupní lexéma a.
Výstup: Další tým v pořadí, který má být využit k odvozování, nebo prázdné pole.

```
1: if τ(∅, a) ∈ R then
2:     return τ(∅, a)                ▷ Na posledním aktivním týmu nezáleží
3: else if R_Last =≥ then
4:     return τ(≥, a)                ▷ Výběr týmu pro první derivaci
5: else
6:     return τ(R_Last, a)           ▷ Tým R_Last ∈ R je použit k výběru
7: end if
```

pro terminál a zadáno, že na týmu předchozím týmu nezáleží, pak se přednostně uplatní $\tau(\emptyset, a)$. Nebyl-li doposud využit žádný tým, tj. jedná se o první derivaci, pak musíme nutně uplatnit $\tau(\geq, a)$. V jakémkoliv jiném případě spoléháme na výběr následujícího týmu na základě zadaného předchozího. Je důležité se pozastavit nad tím, že funkce může vrátit prázdné pole tabulky výběru týmů, respektive žádnou hodnotu, proto je nutné ještě dodatečně otestovat, zdali jsme vůbec nějaký tým získali. Nevrátí-li nám funkce žádný tým, tj. $\tau(R_{Last}, a) \notin R$, pak se jedná o syntaktickou chybu, neboť výběr probíhá na základě očekávaného vstupu.

Dále zavedeme proceduru *simulateStep* popsanou algoritmem 4.3.4, která provede jeden krok necentrální komponenty, jež ani není leader (krok podřízené komponenty). Tato funkce vychází z klasického schématu syntaktického analyzátoru – výběr pravidla, aplikace pravidla, porovnání a načtení dalšího vstupu. Tento jednoduchý algoritmus později použijeme k simulaci funkce podřízených komponent (komponenty nefigurující jako leader). Komponenta P získá ze svého zásobníku vrchní neterminál a současnou vstupní lexému, pomocí nichž ve své vlastní LL tabulce T^P_{LLI} dohledá odpovídající pravidlo. Nebylo-li pravidlo nalezeno, jedná se zřejmě o syntaktickou chybu. V případě nalezení identity můžeme prohlásit krok za úspěšný, v opačném případě nalezené pravidlo aplikujeme. Poté opakovaně porovnáváme vstupní lexému $Token$ s vrcholem zásobníku. V případě shody označíme terminál na zásobníku (nemůžeme jej odstranit kvůli potenciální budoucí komunikaci) a načteme další část vstupní věty. Podřízené komponenty musí uchovávat zpracovanou část řetězce pro potřeby komunikace, a tedy pouze označí část svého zásobníku jako zpracovanou. Stejně jako u běžné LL syntaktické analýzy může nastat i situace, kdy na vrcholu zásobníku narazíme na jiný terminál, než je na vstupu, pak se zcela jistě jedná o syntaktickou chybu. Je nutné doplnit, že použitá operace *top* je definovaná odlišně, než bývá zvykem – vrcholem zásobníku nemyslíme vrchní symbol, nýbrž nejhornější neoznačený symbol (nemusí se tedy jednat s skutečný vrchol zásobníku, nad porovnávaným symbolem může být libovolný počet označených symbolů). Pro úplnost zavádíme operaci *top* formálně:

$$Stack.\text{top}() = X \iff Stack = \underline{a_1 a_2} \ldots \underline{a_k} X Y^* \\ \wedge\, a_1, a_2, \ldots a_k \in T \wedge X \in (T \cup N \cup K)$$

Algoritmus 4.3.4: Simulace kroku komp.: procedura $simulateStep(P, Stack_P, *Token)$

Vstup: Komponenta P pracující na zásobníku $Stack_P$,
a vstupní lexéma $Token$ předávaná odkazem,
kde w je vstupní věta a P není leader ani centrální komponenta.

Výstup: Provede jeden derivační krok pomocí P a načte další vstupní lexému. V případě úspěchu vrací **true**, jinak vrací **false**.

```
1: TopNonterminal := najdi vrchní neterminál na zásobníku Stack_P
2: Rule := α_P(TopNonterminal, Token)                  ▷ získej pravidlo z T^P_LLI
3: if not Rule then
4:     return false                          ▷ Žádné pravidlo → syntaktická chyba
5: else if Rule = id then
6:     return true                                    ▷ Čekací krok bez přepisu
7: end if

8: if levá strana Rule je na zásobníku Stack_P then
9:     nahraď na zásobníku Stack_P levou stranu pravidla Rule pravou stranou
10: else
11:     return false                       ▷ Chybí neterminál → syntaktická chyba
12: end if

13: while Term = Token where Term = Stack_P.top() do                ▷ Porovnání
14:     označ Term jako Term (podtržené)
15:     Token := přečti další symbol z w
16: end while
17: if Stack_P.top() ∈ T then
18:     return false                   ▷ Neočekávaný token → syntaktická chyba
19: end if
20: return true                                  ▷ Derivační krok proveden úspěšně
```

$$\wedge\, Y \in (T \cup N \cup K)^* \text{ pro libovolné } k \geq 0.$$

V tomto bodě je vhodné zavést i operaci *pop*, která také funguje odlišně, než je zvykem. Tuto operaci budeme dále potřebovat pro centrální komponentu. Samotná operace *pop* skýtá stejné úskalí – neodstraňujeme ze zásobníku jeho vrchol, ale nejvrchnější neoznačený symbol, formálně:

$$\begin{aligned} Stack.\text{pop}() = LP \iff Stack = \underline{a_1 a_2} \ldots \underline{a_k} X Y^* \\ \wedge\, L = \underline{a_1 a_2} \ldots \underline{a_k} \wedge P = Y^* \\ \wedge\, a_1, a_2, \ldots a_k \in T \wedge X \in (T \cup N \cup K) \\ \wedge\, Y \in (T \cup N \cup K)^* \text{ pro libovolné } k \geq 0. \end{aligned}$$

Před uvedením popisu a samotného algoritmu nejprve nastíníme jeho hlavní myšlenku. Jednotlivé týmy TCPC gramatického systému se střídají v práci. V rámci týmu je nutná synchronizace – zároveň pracují na svých větných formách všechny komponenty týmu. Zatímco leader týmu může derivovat na základě vstupní lexémy, podřízené komponenty tuto možnost nemají. Podřízené komponenty totiž odvozují část věty, která bude na vstupu někdy v budoucnu. Pro vynucení derivace podřízených komponent bychom se potřebovali

dívat na teoreticky neomezený počet vstupních lexém, což by bylo implementačně náročné, ne-li nesplnitelné. Místo nahlížení na budoucí vstupy využijeme pro vynucení synchronizace čítač. Derivace týmu potom spočívá v tom, že necháme provést derivaci pouze leadera a všem komponentám z týmu zvýšíme hodnotu čítače. Derivaci podřízených komponent přesuneme na některý budoucí okamžik. Podřízená komponenta provede svoji derivaci až ve chvíli, kdy narazíme u centrální komponenty na komunikační symbol. Můžeme se na to tento způsob dívat jako na líné vyhodnocení (*lazy evaluation* – derivujeme až ve chvíli, kdy je to nezbytné). V okamžiku objevení komunikačního symbolu Q_i je na vstupu část věty příslušící komponentě P_i a můžeme dohledat pravidlo v LLI tabulce. Jediné, co musíme zajistit, je délka derivace. Za tímto účelem jsme zavedli čítač – podřízená komponenta musí provést právě k kroků (komponenta pracuje v módu $= k$), kde k je hodnota čítače. Aby tento princip fungoval, musí být centrální komponenta v některém týmu leader (například existuje tým, ve kterém je pouze centrální komponenta).

Nyní už uvádíme algoritmus 4.3.5, který formálně definuje činnost TCPC syntaktického analyzátoru. Nejprve provedeme inicializaci – na zásobník každé komponenty vložíme symbol \$ a její počáteční neterminál. Pro správný výběr týmu realizujícího první derivaci využijeme symbol $\geq$. Pomocí něj a výše zmíněné procedury „selectTeam“ určíme první (další) tým, který má derivovat (aktivní tým). Z tohoto týmu získáme leadera, na jeho zásobníku vyhledáme horní neterminál a v jeho LLI tabulce $T_{LLI}^{\triangleright Comp}$ najdeme pravidlo, které máme použít. V případě prázdného pole se jedná o syntaktickou chybu, v případě nalezení identity se jedná o chybu v návrhu TCPC gramatického systému. Při detekci *id* musíme také algoritmus ukončit, neboť nepřepíše-li leader žádný neterminál, pak si ani nemůže vyžádat další token. Výpočet by se neposunul kupředu a uvázl by v nekonečném cyklu, čemuž musíme zamezit. Jakékoliv aplikovatelné pravidlo využijeme k provedení derivace. Zároveň musíme vynutit synchronizaci podřízených komponent z týmu – všem inkrementujeme čítač.

Po provedení derivace přichází na řadu porovnání, přičemž nejhornější neoznačený symbol na zásobníku může být:

1. terminál shodný se vstupem – v tomto případě došlo k úspěšnému odvození aktuální vstupní lexémy. Centrální komponenta se může svého vrcholu zásobníku zbavit pomocí operce *pop*. Všechny jiné komponenty musí obsah svého zásobníku zachovat pro budoucí komunikaci, proto pouze označí vrchní terminál. Pokud byl zpracovaný terminál \$ (přijatý centrální komponentou), byla věta w úspěšně přijata.

2. terminál rozdílný od vstupu – na vstupu se objevil neočekávaný token a věta w je odmítnuta.

3. komunikační symbol Q_i – nyní je na vstupu část věty, kterou generuje komponenta P_i. Nejprve synchronizujeme komponentu s leaderem jejího týmu (případě s více leadery, je-li P_i ve více týmech – operace je asociativní). Vynutíme provedení $Steps_i$ kroků dlouhé derivace. Poté první výskyt komunikačního symbolu Q_i na zásobníku centrální komponenty nahradíme obsahem zásobníku komponenty P_i. Libovolný další symbol Q_i nahradíme obsahem zásobníku komponenty P_i, přičemž provedeme záměnu všech označených symbolů za neoznačené. Část vstupu odpovídající dalším výskytům Q_i ještě nebyla zpracován a bude zpracována centrální komponentou. Ze zásobníku centrální komponenty můžeme nyní odstranit všechny označené symboly. Na závěr komunikace je potřeba zajistit návrat k axiomu.

Algoritmus 4.3.5: LLI syntaktický analyzátor

Vstup: TCPC gramatický systém $\sigma = (N, T, K, (S_1, P_1), \ldots, (S_n, P_n), R)$,
LLI tabulka v podobě $T_{LLI}^{P_i}$ pro každou komponentu P_i,
funkce výběru aktivního týmu τ a větu w.

Výstup: Odpověď na otázku $w \in L(\sigma)$.

1: $Team := \geq$ ▷ Žádný tým nebyl aktivní
2: $Token :=$ přečti další symbol z w ▷ Aktuální vstupní lexéma
3: **foreach** $P_i \in P$ **where** $P = \{P_1, \ldots, P_n\}$ **do**
4: $Stack_{P_i} := \$S_i$ ▷ Inicializace zásobníku komp. P_i
5: $Steps_{P_i} := 0$ ▷ Inicializace čítače komp. P_i
6: **end foreach**

7: **while true do**
8: $Team :=$ selectTeam($Team$, $Token$) ▷ Výběr týmu
9: **if not** $Team$ **then**
10: **return false** ▷ Není tým → syntaktická chyba
11: **end if**

12: $\triangleright Comp := \triangleright P$ **where** $\triangleright P$ je leader týmu $Team$ ▷ Získáme leadera týmu
13: $TopNonterminal :=$ najdi vrchní neterminál na zásobníku $\triangleright Comp$
14: $Rule := \alpha_{\triangleright Comp}(TopNonterminal, Token)$ ▷ získej pravidlo z $T_{LLI}^{\triangleright Comp}$
15: **if not** $Rule$ **or** $Rule = id$ **then**
16: **return false** ▷ Žádné pravidlo → syntaktická chyba
17: **end if**

18: **if** levá strana $Rule$ je na zásobníku $Stack_{\triangleright Comp}$ **then**
19: nahraď na zásobníku $Stack_{\triangleright Comp}$ levou stranu pravidla $Rule$ pravou stranou
20: **else**
21: **return false** ▷ Chybí neterminál → syntaktická chyba
22: **end if**
23: **foreach** $p \in Team$ **do**
24: $Steps_p := Steps_p + 1$ ▷ Vynucení synchronizace v týmu
25: **end foreach**

4. neterminál – v současném kroku již nelze pracovat žádný další symbol.

Na závěr provedeme kontrolu validity stavů všech komponent – projdeme postupně všechny komponenty a zkontrolujeme, zdali komponenta může pokračovat, tj. má ve své LL tabulce pro vstupní lexému uvedené pravidlo či *id*.

```
26:     while true do                                                    ▷ Porovnání
27:         Term := Stack_▷Comp.top()
28:         if Term = Token then
29:             if ▷Comp is central then
30:                 if Token = $ then
31:                     return true                                      ▷ Věta přijata
32:                 end if
33:                 Stack_▷Comp.pop()
34:             else
35:                 označ Term jako Term
36:             end if
37:             Token := přečti další symbol z w
38:         else if Term ∈ T then
39:             return false                ▷ Neočekávaný token → syntaktická chyba

                                            ▷ Komunikace – pokud centrální komp. obsahuje symbol z K
40:         else if Term = Q_j where Q_j ∈ (K \ {Q_▷Comp}) then
41:             while Steps_j > 0 do                ▷ Simulace derivací podřízené komp.
42:                 if not simulateStep(P_j, Stack_j, Token) then
43:                     return false            ▷ Simulace selhala → syntakická chyba
44:                 end if
45:                 Steps_j := Steps_j − 1
46:             end while
47:             nahraď první výskyt Q_j ve Stack_▷Comp obsahem zásobníku Stack_j
48:             if Stack_▷Comp obsahuje symbol Q_j then
49:                 odstraň značení pro všechny označené symboly na zásobníku Stack_j
50:                 nahraď všechny výskyty Q_j ve Stack_▷Comp obsahem zásobníku Stack_j
51:             end if
52:             Stack_j := $S_j                                          ▷ Návrat k axiomu
53:             odstraň z Stack_▷Comp všechny označené symboly   ▷ Zprac. část vstupu
54:         else
55:             break
56:         end if
57:     end while                                                        ▷ Porovnávací cyklus

                                                ▷ Kontrola id, mohou všechny komp. pokračovat?
58:     foreach P_i ∈ P where P = {P_1, ..., P_n} do
59:         TopNonterminal := najdi vrchní neterminál na zásobníku
60:         if α_{P_i}(TopNonterminal, Token) je prázdné then
61:             return false            ▷ Neexistuje pravidlo → syntaktická chyba
62:         end if
63:     end foreach
64: end while                                                            ▷ Hlavní cyklus – while true
```

4.4 Generativní kapacita TCPC syntaktické analýzy

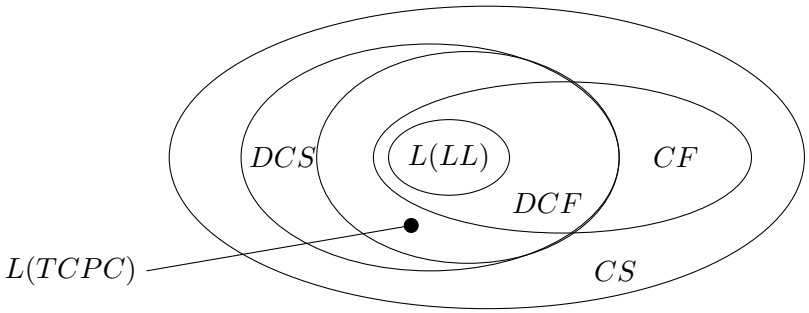
TCPC syntaktická analýza je založená na LL syntaktické analýze, respektive na deterministických zásobníkových automatech pracujících shora dolů. Není žádným překvapením, že tedy poskytuje vyší generativní kapacitu než LL syntaktická analýza – pro důkaz lze použít TCPC gramatický systém $\sigma = (N, T, \{Q_1\}, (S_1, P_1), \{R_1 = \{P_1\}\})$. Je zřejmé, že pokud by některé pravidlo obsahovalo Q_1, pak vznikne cyklus a nelze přijmout, proto se omezíme na pravidla $A \to x$, kde $|x|_{Q_1} = 0$, tj. pravidla neobsahující komunikační symbol. Máme-li pouze jeden tým, můžeme zavést funkci výběru týmu ve tvaru $\forall a \in T : \tau(\emptyset, a) = R_1$. Takto zavedený tým je ekvivalentní s LL gramatikou ve tvaru $G = (N, T, P, S)$, tedy můžeme přijímat jakýkoliv jazyk z $L(LL)$. Pro důkaz ostré inkluze mezi rodinami $L(LL)$ a $L(TCPC)$ musíme najít alespoň jeden jazyk, který není LL, ale lze ho přijímat pomocí TCPC gramatického systému. Takovým jazykem je například $L_1 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$ zmíněný výše v kapitole 4.2.3. Tento jazyk je kontextový a zároveň deterministický, což ovšem neznamená, že lze přijímat všechny deterministické kontextové jazyky. Uvažujme jiný kontextový jazyk $L_2 = \{a^{2^n} \mid n \geq 0\}$. Pro jeho přijetí bychom museli využít decentralizovaný systém. Zřejmě tedy platí následující vztah:

$$L(LL) \subset L(TCPC) \subset DCS,$$

kde $L(TCPC)$ je rodina jazyků přijímaných TCPC syntaktickými analyzátory. Zároveň není překvapivé, že existují nedeterministické jazyky z CF, které nelze pomocí TCPC systému přijmout, příkladem takového jazyka je $L = \{ww^r \mid w \in \{a, b\}^*\}$, tedy platí:

$$CF \not\subset L(TCPC)$$

Otevřeným problémem zůstává, jestli lze pomocí TCPC gramatického systému přijmout jakýkoliv jazyk z DCF. Na příkladu alespoň ukážeme, že existují jazyky z $L(LR)$, respektive z DCF, které nepatří do $L(LL)$, ale patří do $L(TCPC)$. Uvažujme jazyk $L_3 = \{a^i b^j \mid i >= j\}$ a TCPC gramatický systém $\sigma_3 = (\{S_1, S_2, B\}, \{a, b\}, \{Q_1, Q_2\}, (S_1, P_1), (S_2, P_2), \{R_1 = \{\triangleright P_1\}, R_2 = \{\triangleright P_2\}\})$, kde $P_1 = \{S_1 \to S_1 \mid Q_2, B \to b \mid \varepsilon, S_2 \to \varepsilon\}$ a $P_2 = \{S_2 \to aS_2B\}$. Nyní definujeme funkci výběru týmu jako $\tau(\emptyset, a) = R_2$, $\tau(\emptyset, b) = R_1$. Jednoduše lze nahlédnout, že počet neterminálů B a tedy i terminálů b nikdy nebude větší než počet terminálů a. Výše uvedené vztahy jsou shrnuty na obrázku 4.4.1.



Obrázek 4.4.1 Hierarchie rodin jazyků CS, DCS, CF, DCF, $L(LL)$ a oblast užití TCPC syntaktických analyzátorů $L(TCPC)$.

5 Systémy syntaktických analyzátorů

V této kapitole se zkusíme podívat na gramatické systémy z jiného úhlu pohledu, nebudeme se snažit vytvořit jeden syntaktický analyzátor pro jeden gramatický systém, nýbrž použijeme klasické bezkontextové metody k získání jednoho syntaktického analyzátoru pro každou komponentu zvlášť. Tyto analyzátory poté spojíme do většího celku – systému syntaktických analyzátorů (dále jen SA systém), aby emulovaly chování gramatického systému. Zavedeme nový teoretický model založený na řetězení syntaktických analyzátorů, který nám poskytne aparát k formálnímu popisu a implementaci metod překladu. Cílem bude stejně jako v předchozích kapitolách zvýšení generativní kapacity bezkontextových metod.

5.1 Systémy syntaktických analyzátorů

Definice 5.1.1. Nechť $G_1, \ldots, G_n$ jsou BKG ve tvaru $G_i = (N_i, T_i, P_i, S_i)$ pro $1 \leq i \leq n$, $n \in \mathbb{N}$. Pak SA systémem rozumíme uspořádanou n-tici $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$, kde

$$
\begin{aligned}
&N = \bigcup_{i=1}^{n} N_i, \qquad T = \bigcup_{i=1}^{n} T_i, \\
&\mathcal{G} \subseteq \{G_1, \ldots, G_n\}, \\
&\mathcal{P} \subseteq \{P(G_1), \ldots, P(G_n)\}, P(G_i) = G_i^A, \text{ kde } A \in \eta, \eta = \{LL, LR, CG, C\}, \\
&\circ \subseteq \mathcal{P} \times \mathcal{P}, \\
&\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P} \times \{s, \neg s, r, \succ r, c(\phi), \succ c(\phi), t, \neg t, m(\mu), f(g), \succ f(g)\},
\end{aligned}
$$

kde N je konečná množina neterminálů, T je konečná množina terminálů, $\mathcal{G}$ je konečná množina gramatik a $\mathcal{P}$ je konečná množina syntaktických analyzátorů. Prvky množiny $\mathcal{G}$ nazýváme *modely* a prvky $\mathcal{P}$ nazýváme *komponenty*, přičemž pro $1 \leq i \leq n$ platí $P(G_i) \in \mathcal{P} \Rightarrow G_i \in \mathcal{G}$, tedy pro každou komponentu musí existovat model. $A \in \eta$ je typ komponenty, $\circ$ je relace skládání komponent (*Composition relation*) a $\mathcal{R}$ je množina omezení (*Restrictions*) mezi komponentami.

Komponentou SA systému může být libovolný syntaktický analyzátor vystavěný nad některým modelem z $\mathcal{G}$. Bavíme-li se o bezkontextových metodách syntaktické analýzy, budou tyto analyzátory realizovány deterministickými zásobníkovými automaty. Tento koncept ovšem lze použít i pro kontextové syntaktické analyzátory (za předpokladu, že je i model kontextový). Jedná se o velice dobře rozšiřitelný teoretický koncept – my pro jednoduchost zavedeme pouze 4 základní analyzátory, ale pozorný čtenář může množinu $\mathcal{P}$ rozšířit o další prvky:

- G_i^{LL} značí klasický LL syntaktický analyzátor vystavěný nad gramatikou G_i,
- G_i^{LR} značí klasický LR syntaktický analyzátor vystavěný nad gramatikou G_i,
- G_i^{CG} značí syntaktický analyzátor vystavěný nad kontrolovanou gramatikou G_i. Tento analyzátor očekává na vstupu kromě větné formy i sekvenci pravidel, pomocí kterých má větnou formu odvodit. Přijme právě tehdy, pokud se mu pomocí aplikace zadaných pravidel podaří odvodit větnou formu, jež je shodná se vstupní větou. Formálně:

$$u \in L\left(G_i^{CG}\right) \Longleftrightarrow S_i s_{i_0} u \vdash^* f_i[r_1, \ldots, r_k] \wedge [r_1, \ldots, r_k] = R_c$$
$$\text{pro } u \in \Sigma_i^*, f_i \in F_i \text{ a } 1 \leq i \leq n, \text{ kde } r_j \in R_i, 1 \leq j \leq k, k \in \mathbb{N}$$
$$\text{a } R_c \text{ je zadaná posloupnost pravidel.}$$

- G_i^C je komparátor. Jedná se o jednoduchý zásobníkový automat přijímající vyprázdněním zásobníku, který má pouze pravidla tvaru $as_0a \ \rightarrow s_0$ pro každé $a \in T_i$, kde $s_0 \in F$ je počáteční stav. Nejprve se vloží na zásobník zadaná posloupnost symbolů, a poté se analyzátor spustí. Dokáže-li automat vyprázdnit zásobník a je-li přečten celý vstup, pak je řetězec přijat. Formálně:

$$u \in L\left(G_i^C\right) \Longleftrightarrow v s_{i_0} u \vdash^* s_{i_0} \text{ pro } u, v \in \Sigma_i^*, s_{i_0} \in F_i \text{ a } 1 \leq i \leq n,$$
$$\text{kde } v \text{ je zadaná posloupnost symbolů.}$$

Jednoduchým zobecněním můžeme místo G_i^{LL} dosadit libovolný syntaktický analyzátor pracující shora dolů a místo G_i^{LR} libovolný syntaktický analyzátor pracující zdola nahoru.

Je vhodné zmínit, že více syntaktických analyzátorů v SA systému může sdílet stejný model. Z toho důvodu zavádíme konvenci značení $G_{i|j}^A$, kde A značí typ použitého analyzátoru, i značí model z $\mathcal{G}$ a j je označení instance analyzátoru.

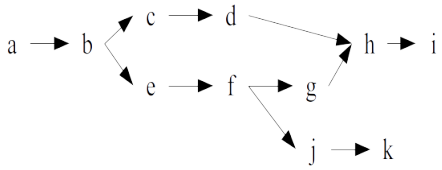
Nyní se blíže podívejme na relaci kompozice $\circ$. Tato relace nám udává vnitřní strukturu systému – specifikuje propojení mezi komponentami. Na relaci $\circ$ se můžeme dívat jako na operaci skládání komponent. Výraz $G_2^A \circ G_1^A$ budeme číst „G_2^A po G_1^A“, přičemž myslíme, že nejprve pracuje komponenta G_1^A a až dokončí svoji činnost, přijde na řadu komponenta G_2^A. Sémantika toho zápisu je pro dva syntaktické analyzátory následující: nejprve pracuje komponenta G_1^A, odmítne-li, pak je řetězec odmítnut ($w \notin L(G_2^A \circ G_1^A)$). Přijme-li G_1^A, pak začne pracovat komponenta G_2^A, pokud i ta přijme, pak je věta přijata, v opačném případě je zamítnuta.

Definice 5.1.2. Nechť $g_i, g_j \in \mathcal{P}$ jsou komponenty. Řekneme, že g_j *následuje za* g_i, zapsáno $g_j > g_i$, pokud $g_j \circ g_{m_k} \circ \ldots \circ g_{m_1} \circ g_i$, pro nějaké $g_{m_z} \in \mathcal{P}$, $1 \leq z \leq m$, $m \in \mathbb{N}$. Dále pro každé $g \in \mathcal{P}$ definujeme množinu následníků g^+ jako $g^+ = \{g_j \in \mathcal{P} \,|\, g_j > g\}$. Množina předchůdců g^- je definována obdobně jako $g^- = \{g_j \in mcP \,|\, g > g_j\}$. Dále zavádíme množinu největších prvků relace $sup(\circ) = \{g \in \mathcal{P} \,|\, g^+ = \emptyset\}$ a množinu nejmenších prvků relace $inf(\circ) = \{g \in \mathcal{P} \,|\, g^- = \emptyset\}$.

Příklad 5.1.3. Uvažujme relaci uvedenou na obrázku 5.1.1, pak platí:

- $f^+ = \{a, b, c\}$,
- $f^- = \{g, h, i, j, k\}$,
- $sup(\circ) = \{i, k\}$, tj. i a k jsou největší prvky,

- $inf(\circ) = \{a\}$, tj. a je nejmenší prvek.



Obrázek 5.1.1 Příklad relace kompozice $\circ$ pro prvky $a, \ldots, k \in \mathcal{P}$, kde $\circ = \{(a,b),(b,c),(b,e),(c,d),(d,h),(e,f),(f,g),(f,j),(g,h),(h,i),(j,k)\}$.

Na relaci $\circ$ budeme klást omezující požadavky, relace musí být:

1. *ireflexivní* $(\forall g \in \mathcal{P} : \neg(g \circ g))$
2. *asymetrická* $(\forall g_i, g_j \in \mathcal{P} : g_j \circ g_i \Rightarrow \neg(g_i \circ g_j))$,
3. *acyklická* $(\forall g_i, g_j \in \mathcal{P} : g_j \in g_i^+ \Rightarrow \neg(g_i \in g_j^+))$,

Je zřejmé, že reflexivní komponentu nemůžeme připustit, protože takový analyzátor by potřeboval ke své činnosti svůj předchozí výstup, a tedy nemohl by být nikdy spuštěn. Stejný problém nastává u symetrie – nelze spustit ani jeden z analyzátorů $g_i, g_j \in \mathcal{P}$, protože g_i by potřeboval výstup z g_j a naopak. Zároveň by nešlo určit pořadí vyhodnocování komponent, což je zcela fundamentální při interpretaci SA systému. Relace musí být acyklická, abychom mohli nalézt komponenty, u nichž vyhodnocení SA systému začne $(inf(\circ))$. K uváznutí (deadlock) sice dojít nemůže, protože vstupní program nebude nekonečný, ale cykly by do SA systémů zanesly velikou míru nedeterminismu. Na druhou stranu může být relace $\circ$ v obecné podobně vícecestná, jak je zobrazeno na obrázku 5.1.1. Pro správnou interpretaci je ovšem potřeba definovat chování komponent, u kterých se tok vstupního programu rozbíhá $(\forall a,b,c \in \mathcal{P} : (a,b) \in \circ \wedge (a,c) \in \circ)$ nebo spojuje $(\forall a,b,c \in \mathcal{P} : (a,b) \in \circ \wedge (c,b) \in \circ)$. Tuto problematiku odložíme na později a vrátíme se k ní až po rozboru množiny $\mathcal{R}$.

Příklad 5.1.4. Pozornému čtenáři jistě neuniklo, že v relaci $\circ$ může být více instancí komponent stejného typu $A \in \eta$ nad stejným modelem G_i, ale ireflexivita, asymetrie a acykličnost musí být dodržena pro každou z těchto instancí. Situace $G^A_{i|2} \circ G^A_{i|1}$ je zcela korektní, zatímco $G^A_{i|1} \circ G^A_{i|1}$ porušuje reflexivitu. Obdobně $\circ = \left\{ \left(G^A_{i|1}, G^A_{j|1}\right), \left(G^A_{j|1}, G^A_{i|2}\right) \right\}$ je v pořádku, zatímco $\circ = \left\{ \left(G^A_{i|1}, G^A_{j|1}\right), \left(G^A_{j|1}, G^A_{i|1}\right) \right\}$ porušuje antisymetrii.

Definice 5.1.5. *Řetězcem relace* $\circ$ budeme nazývat libovolnou množinu $cas(\circ) \subseteq \mathcal{P}$ takovou, že:

$$\forall g_i \in (cas(\circ) \setminus sup(cas(\circ)))\ \exists g_j \in cas(\circ) : g_j \circ g_i \wedge$$
$$\forall g_i \in (cas(\circ) \setminus inf(cas(\circ)))\ \exists g_j \in cas(\circ) : g_i \circ g_j \wedge$$
$$\forall g_i, g_j, g_k \in cas(\circ) : (g_i \circ g_j \wedge g_i \circ g_k \Longrightarrow g_j = g_k) \wedge (g_j \circ g_i \wedge g_k \circ g_i \Longrightarrow g_j = g_k).$$

Řetězec relace $\circ$ budeme též označovat jako *kaskádu komponent* či *kaskádu analyzátorů* nad $\circ$. Dva řetězce $cas_1(\circ)$, $cas_2(\circ)$ označujeme jako nezávislé, pokud platí $cas_1(\circ) \cap cas_2(\circ) = \emptyset$. Řetězec označujeme jako maximální, pokud $sup(cas(\circ)) \in sup(\circ) \wedge inf(cas(\circ)) \in inf(\circ)$, zapsáno $cas(\circ)^{\uparrow}$. Dále zavedeme *rozklad na relaci* $\circ$ jako množinu χ obsahující všechny

maximální řetězce, tj. $\chi = \{c \in 2^{\mathcal{P}} \mid cas(c)^{\uparrow}\}$. *Mohutností relace* $\circ$, značeno $|\chi|$ nebo také $|\circ|$, poté myslíme počet prvků množiny χ. Relaci $\circ$ budeme nazývat *vícecestnou* nebo též *větvenou*, pokud $\exists c_1, c_2 \in \chi, c_1 \neq c_2 : c_1 \cap c_2 \neq \emptyset$.

Řetězcem komponent je tedy libovolná posloupnost komponent, které na sebe navazují. Každá komponenta kromě poslední (sup) musí mít právě jednoho následníka a každá komponenta kromě první (inf) musí mít právě jednoho předchůdce. Dva řetězce jsou nezávislé, pokud neobsahují společnou komponentu. Maximální řetězec nad $\circ$ obsahuje na sebe navazující prvky relace od nejmenšího (inf) po největší (sup). Rozklad $\circ$ potom obsahuje všechny řetězce maximální délky. Relace $\circ$ je větvená právě když některé řetězce rozkladu obsahují stejnou komponentu.

Definice 5.1.6. *Řezem relace* $\circ$ budeme nazývat množinu $co(\circ) \subseteq \mathcal{P}$ takovou, že

$$(\forall g_i \in co(\circ) : \nexists g_j \in co(\circ) : g_j \in g_i^+) \wedge (\forall g_i \in co(\circ) : \nexists g_j \in co(\circ) : g_j \in g_i^-).$$

Řez relace $\circ$ je libovolná množina obsahující nesrovnatelné prvky, tedy pro každé dvě komponenty g_i a g_j platí, že g_j ani nepředchází ani nenásleduje g_i. Komponenty v $co(\circ)$ jsou zcela nezávislé, mohou zpracovávat stejný vstup a mohou být vyhodnocovány v libovolném pořadí.

Příklad 5.1.7. Relace uvedená na obrázku 5.1.1 obsahuje řetězce $\{a\}$, $\{a, b\}$, $\{a, b, c\}$, $\{a, b, c, d\}$ a mnoho dalších. Řez $cas(\circ) = \{a, b, c, d, h, i\}$ je maximální, protože obsahuje prvek ze $sup(\circ)$, konkrétně i, a zároveň prvek z $inf(\circ)$, konkrétně a. Rozklad relace $\circ$ je množina $\chi = \{\{a, b, c, d, h, i\}, \{a, b, e, f, g, h, i\}, \{a, b, e, f, j, k\}\}$, mohutnost $\circ = 3$. Relace $\circ$ je větvená, neboť všechny prvky rozkladu obsahují prvky a, b. Daná relace obsahuje mimo jiné řezy $\{a\}$, $\{e, c\}$, $\{c, g, k\}$ či $\{i, k\}$.

Relace $\circ$ udává pořadí vyhodnocování komponent SA systému, ale nelze pomocí ní nijak zvýšit generativní kapacitu. Ke zvýšení síly slouží množina omezení $\mathcal{R}$. Omezení ve tvaru (g_i, g_j, π) můžeme klást na libovolné dvě komponenty z $g_i, g_j \in \mathcal{P}$, pro které platí $g_j \in g_i^+$, kde $\pi \in \{s, \neg s, r, \succ r, c(\phi), \succ c(\phi), t, \neg t, m(\mu), f(g), \succ f(g)\}$ je funkce udávající typ a význam omezení. Význam tohoto zápisu je následující: komponenta g_j, která je vyhodnocena někdy po komponentě g_i, musí splňovat omezení π. Přijme-li komponenta g_j tak, že není omezení π splněno, pak v rámci SA systému g_j zamítne. Pouze pokud přijme g_j a jsou splněna omezení na tuto komponentu kladená, pak je věta přijata. V tomto směru lze přirovnat SA systém k regulovaným gramatikám, ale regulace zde není uvnitř jedné komponenty, nýbrž mezi různými komponentami. Je zřejmé, že na dva libovolné analyzátory z $\mathcal{P}$ může být kladeno více omezení zároveň.

Nyní se postupně podíváme na jednotlivá omezení, popíšeme si jejich význam a definujeme je formálně. Nechť $g_i = (Q_i, \Sigma_i, \Gamma_i, R_i, s_{0_i}, S_i, F_i)$ a $g_j = (Q_j, \Sigma_j, \Gamma_j, R_j, s_{0_j}, S_j, F_j)$, $g_i, g_j \in \mathcal{P}$, pro $1 \leq i, j \leq n$, jsou syntaktické analyzátory nad libovolnými modely $G_i, G_j \in \mathcal{G}$ takové, že platí $g_j \in g_i^+$. Mezi g_j a g_i se tedy může vyskytovat libovolný počet dalších komponent $g_{m_1}, \ldots, g_{m_z}$, $1 \leq m \leq n$, $z \in \mathbb{N}$, které jsou v relaci $\circ$. Tyto komponenty tvoří kaskádu analyzátorů $g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i$. Na vstupu tohoto uskupení je řetězec $x \in T$, pro který platí $x = uwv$, $u \in T_i$, $w \in T$, $v \in T_j$, respektive $u \in \Sigma_i$, $w \in \Sigma$, $v \in \Sigma_j$. Jednoduše řečeno, g_i bude zpracovávat podřetězec u, g_j zpracuje v a mezi nimi může být libovolný počet komponent, které dohromady zpracují w. Je zřejmé, že pro $g_j \circ g_i$ $(z = 0)$ platí $w = \varepsilon$. Uvedenou konfiguraci použijeme k formální definici omezení:

- (g_i, g_j, s) značí *pozitivní synchronizaci* mezi komponentami. Pro $(g_j, g_i) \in \circ$ je sémantika následující: nejprve pracuje g_i, přijme-li řetězec na vstupu v k krocích, pak začne pracovat g_j, který musí také přijmout v k krocích, jinak odmítne. Pro libovolné $g_j \in g_i^+$ rovněž platí, že pro přijetí musí g_j vykonat stejný počet kroků jako g_i s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^k f_i \wedge S_j s_{0_j} v \vdash^k f_j$$
$$\wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \text{ pro } f_i \in F_i, f_j \in F_j \text{ a } k \geq 0.$$

- $(g_i, g_j, \neg s)$ značí *negativní synchronizaci* mezi komponentami. Pro $(g_j, g_i) \in \circ$ je sémantika následující: nejprve pracuje g_i, přijme-li řetězec na vstupu v k krocích, pak začne pracovat g_j, který musí přijmout v $\overline{k}$ krocích, kde $k \neq \overline{k}$, jinak odmítne. g_j tedy musí provést jiný počet kroků než g_i. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^k f_i \wedge S_j s_{0_j} v \vdash^{\overline{k}} f_j \wedge k \neq \overline{k}$$
$$\wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \text{ pro } f_i \in F_i, f_j \in F_j \text{ a } k, \overline{k} \geq 0.$$

- (g_i, g_j, r) značí *replikaci* mezi komponentami. Pro $(g_j, g_i) \in \circ$ je sémantika následující: nejprve pracuje g_i, přijme-li řetězec na vstupu, pak se tento řetězec vloží na zásobník g_j místo axiomu a následně se g_j spustí. Přijme-li g_j, je řetězec přijat. Jinak řečeno g_j využívá vstup přijatý g_i k validaci svého vlastního vstupu. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i \wedge u s_{0_j} v \vdash^* f_j$$
$$\wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \text{ pro } f_i \in F_i \text{ a } f_j \in F_j.$$

- $(g_i, g_j, \succ r)$ značí *slabou replikaci* mezi komponentami. Její sémantika je shodná se sémantikou replikace, ale v tomto případě není axiom g_j nahrazen replikovaným řetězcem, ale řetězec je vložen na zásobník za axiom. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i \wedge S_j u s_{0_j} v \vdash^* f_j$$
$$\wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \text{ pro } f_i \in F_i \text{ a } f_j \in F_j.$$

- $(g_i, g_j, c(\psi))$ značí *konverzi* (*conversion*) mezi analyzátory. Sémantika pro $(g_j, g_i) \in \circ$ je následující: nejprve pracuje g_i, přijme-li řetězec na vstupu w, pak se pro tento řetězec spočte $\overline{w}$, jež se vloží na zásobník g_j místo axiomu a následně se g_j spustí. Výpočet $\overline{w}$ spočívá v prosté textové náhradě jednoho symbolu za jiný, přičemž množinu možných náhrad udává množina ψ. Příjme-li poté i g_j, je řetězec přijat. Jinak řečeno, g_j k validaci svého vlastního vstupu využívá přijatý konvertovaný vstup komponenty g_i. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i \wedge \overline{u} s_{0_j} v \vdash^* f_j$$
$$\wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1})$$
$$\text{pro } u = X_1, \ldots, X_t,\ \overline{u} = c(X_1, \phi), \ldots, c(X_t, \phi), t \geq 0, f_i \in F_i \text{ a } f_j \in F_j,$$

kde $c(X, \psi) = Y \iff (X, Y) \in \phi$ pro $X \in \Gamma_i, Y \in \Gamma_j$.

Nutno doplnit, že ψ je zobrazení, tedy platí $(x, y_1) \in \psi \wedge (x, y_2) \in \psi \implies y_1 = y_2$. Tím je vyloučena nejednoznačnost při konverzi symbolů.

- $(g_i, g_j, \succ c(\phi))$ značí *slabou konverzi* mezi komponentami. Její sémantika je shodná se sémantikou konverze, ale v tomto případě není axiom g_j nahrazen konvertovaným řetězcem, ale řetězec je vložen na zásobník za axiom. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i \wedge S_j \overline{u} s_{0_j} v \vdash^* f_j \\ \wedge w \in L(g_{m_z} \circ \ldots \circ g_{m_1}).$$

- (g_i, g_j, t) značí *pozitivní sledování* (*tracking*) mezi komponentami. Sémantika pro $(g_j, g_i) \in \circ$ je následující: nejprve pracuje g_i, přijme-li řetězec na vstupu postupnou aplikací pravidel $r_1, \ldots, r_k$, pak začne pracovat g_j. Pokud i g_j přijalo pomocí pravidel $r_1, \ldots, r_k$, pak je řetězec přijat, jinak je zamítnut. V tomto případě se tedy komponenty synchronizují pomocí použitých pravidel a pro přijetí řetězce musí použít obě komponenty shodná pravidla. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i[r_1, \ldots, r_k] \wedge S_j s_{0_j} v \vdash^* f_j[r_1, \ldots, r_k] \\ \wedge w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \\ \text{pro } f_i \in F_i, f_j \in F_j, r_p \in R_i \cap R_j, 1 \leq p \leq k \text{ a } k \geq 0.$$

- $(g_i, g_j, \neg t)$ značí *negativní sledování* (*negative tracking*) mezi komponentami. Sémantika pro $(g_j, g_i) \in \circ$ je následující: nejprve pracuje g_i, přijme-li řetězec na vstupu postupnou aplikací pravidel $r_1, \ldots, r_k$, pak začne pracovat g_j. Pokud g_j přijme pomocí libovolné posloupnosti pravidel kromě posloupnosti $r_1, \ldots, r_k$, pak je řetězec přijat, jinak je zamítnut. Jednoduše řečeno, komponenta g_j má předem stanovenou posloupnost pravidel, kterou nesmí použít, ale může přijmout libovolným jiným způsobem. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i[r_{i_1}, \ldots, r_{i_k}] \wedge S_j s_{0_j} v \vdash^* f_j[r_{j_1}, \ldots, r_{j_o}] \\ \wedge [r_{i_1}, \ldots, r_{i_k}] \neq [r_{j_1}, \ldots, r_{j_o}] \wedge w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \\ \text{pro } f_i \in F_i, f_j \in F_j, r_{i_p} \in R_i, r_{j_q} \in R_j, 1 \leq p \leq k, 1 \leq q \leq o \text{ a } k, o \geq 0.$$

- $(g_i, g_j, m(\mu))$ značí *mapování* pravidel mezi analyzátorem a gramatikou prostřednictvím množiny μ. Sémantika pro $(g_j, g_i) \in \circ$ je následující: nejprve pracuje g_i, přijme-li řetězec na vstupu postupnou aplikací pravidel $r_1, \ldots, r_k$, pak začne pracovat g_j. Pokud g_j přijme pomocí posloupnosti pravidel $m(r_1), \ldots, m(r_k)$, pak je řetězec přijat, jinak je zamítnut. Mapování pravidel je formě dvojic $(r_1, \overline{r_1}), \ldots, (r_k, \overline{r_k})$ je dané množinou μ. Jinak řečeno, pro každé pravidlo použité komponentou g_i je zadáno jiné pravidlo, které g_j musí použít. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Formálně:

$$uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \iff S_i s_{0_i} u \vdash^* f_i[r_1, \ldots, r_k] \wedge S_j s_{0_j} \vdash^* f_j[\overline{r_1}, \ldots, \overline{r_k}] \\ \wedge (r_z, \overline{r_z}) \in \mu \wedge w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \\ \text{pro } f_i \in F_i, f_j \in F_j, 1 \leq z \leq k \text{ a } k \geq 0.$$

- $(g_i, g_j, f(g))$ značí *aplikaci funkce* při předávání řetězce mezi analyzátory. Sémantika pro $(g_j, g_i) \in \circ$ je následující: nejprve pracuje g_i, přijme-li řetězec w na vstupu, pak se spočte $\overline{w} = g(w)$, $\overline{w}$ se vloží na zásobník g_j místo axiomu a g_j se spustí. Přijme-li g_j, pak je celý vstup přijat, jinak je odmítnut. Totéž platí pro libovolné $g_j \in g_i^+$ s tím rozdílem, že mezi g_i a g_j může být libovolný počet dalších komponent. Funkce g může být libovolná Turingovsky vyčíslitelná funkce. Formálně:

$$\begin{aligned} uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \Longleftrightarrow S_i s_{i_0} u \vdash^* f_i \wedge \overline{u} s_{0_j} v \vdash^* f_j \\ \wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}) \\ \text{pro } \overline{u} = g(u), \overline{u} \in \Gamma_j^*, f_i \in F_i \text{ a } f_j \in F_j, \\ \text{kde } g : \Sigma_i^* \longrightarrow \Gamma_j^*. \end{aligned}$$

- $(g_i, g_j, \succ f(g))$ značí *slabou aplikaci funkce* při předávání řetězce mezi analyzátory. Její sémantika je shodná se sémantikou aplikace funkce, ale v tomto případě není axiom g_j nahrazen vypočteným řetězcem, ale řetězec je vložen na zásobník za axiom. Formálně:

$$\begin{aligned} uwv \in L(g_j \circ g_{m_z} \circ \ldots \circ g_{m_1} \circ g_i) \Longleftrightarrow S_i s_{i_0} u \vdash^* f_i \wedge S_j \overline{u} s_{0_j} v \vdash^* f_j \\ \wedge\, w \in L(g_{m_z} \circ \ldots \circ g_{m_1}). \end{aligned}$$

Pozorný čtenář si zajisté povšiml, že ne všechna omezení lze použít na libovolnou komponentu typu $A \in \eta$. Například omezení $\neg t$ nedává ve smysl s komponentou typu CG, naopak omezení t je pro CG nutné zadat vždy (pokud není zadáno t, nebo $\neg t = t$, pak přijímá CG prázdný jazyk). Slabá omezení $\succ r, \succ c(\phi), \succ f(g)$ lze použít pouze pro komponentu typu LL (obecně libovolný analyzátor pracující shora dolů), ale nelze je využít pro LR (analyzátor pracující zdola nahoru). Pro komponentu typu C tato omezení vůbec nedávají smysl. Smysluplné použití omezení ponecháme na čtenáři, ale zaměříme se na výlučné použití některých omezení, které plyne přímo z definice.

- Každá komponenta g_i může mít nejvýše jedno omezení typu $\overline{\phi} = \{r, \succ r, c(\phi), \succ c(\phi), f(g), \succ f(g)\}$, tj. $\forall g_j, g_i, g_k \in \mathcal{P}, \forall \eta_1, \eta_2 \in \overline{\phi} : (g_j, g_i, \eta_1) \in \mathcal{R} \wedge (g_k, g_i, \eta_2) \in \mathcal{R} \Longrightarrow g_j = g_k$. Důvod je zcela zřejmý, je-li na zásobníku nahrazen axiom komunikovaným řetězcem, pak již nelze aplikovat jiné omezení nahrazující axiom. U slabých omezení se sice nenahrazuje axiom, ale nelze přesně stanovit pořadí vkládání řetězců na zásobník. To závisí na interpretaci SA systému a jiné pořadí by změnilo jazyk přijímaný komponentou – vícenásobná aplikace těchto omezení by byla zdrojem nedeterminismu.

- Každá komponenta g_i může mít nejvýše jedno omezení typu t. Tato restrikce vychází z unifikovaného přístupu k interpretaci sledování – pro komponentu typu CG musí být z principu její činnosti vždy zadáno právě jedno omezení typu t. Pro komponenty LL ani LR nelze interpretovat n-tici omezení typu t bez přidané sémantiky (bylo by potřeba zavést nějakou logickou operaci nad touto n-ticí), a proto nepovolíme použití mnohonásobného sledování. Formálně: $\forall g_j, g_i, g_k \in \mathcal{P} : (g_j, g_i, t) \in \mathcal{R} \wedge (g_k, g_i, t) \in \mathcal{R} \Longrightarrow g_j = g_k$.

Na množinu $\mathcal{R}$ se lze také dívat jako na relaci $\mathcal{R}^{\succeq}$ – každý prvek $(g_i, g_j, \eta_{ij}) \in \mathcal{R}$ si vyjádříme jako dvojici (g_i, g_j) s atributem η_{ij}. S relací $\mathcal{R}^{\succeq}$ pak můžeme pracovat podobně jako s $\circ$. Především musíme zajistit, aby bylo možné SA systém jednoznačně interpretovat

(vyhodnotit), a proto na $\mathcal{R}^{\succeq}$ budeme klást stejné požadavky, jako na $\circ$. $\mathcal{R}^{\succeq}$ musí nutně být ireflexivní, asymetrická a acyklická.

V tomto bodě je formální definice SA systémů kompletní a můžeme se začít zabývat přijímaným jazykem. Pozorný čtenář již zcela jistě intuitivně chápe pojem *jazyk přijímaný SA systémem*, ale nyní je na čase tento intuitivní pojem formalizovat pomocí definice 5.1.8.

Definice 5.1.8. Nechť $\sigma = SA$ je SA systém, $w \in T^*$ je věta a χ je rozklad relace $\circ$. Jazyk přijímaný kaskádou komponent $g_k \circ \ldots \circ \ldots \circ g_1$ SA systému σ, kde $\{g_1, \ldots, g_k\} \in \chi$, zapsáno $L(g_k \circ \ldots \circ \ldots \circ g_1)$, je pro libovolné $k \geq 0$ definován jako:

$$L(g_k \circ \ldots \circ \ldots \circ g_1) = \{w \in T^* \mid w = x_1 x_2 \ldots x_k \wedge \forall i \in \{1, \ldots k\} : x_i \in L(g_i)\}.$$

Komponenta g_k v této kaskádě se nazývá *koncová komponenta*. Dále platí:

$$w \in L(g_k \circ \ldots \circ \ldots \circ g_1) \iff g_k \text{ přijala } w \iff x_k \in L(g_k).$$

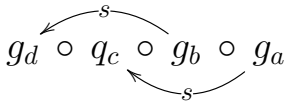
Jazyk přijímaný koncovými komponentami $g_1, \ldots, g_m$ v SA systému σ, zapsáno $L(\sigma \mid g_1, \ldots, g_m)$, je pro $m \geq 0$ definován jako:

$$L(\sigma \mid g_1, \ldots, g_m) = \{w \in T^* \mid w \in L(g_1) \wedge \ldots \wedge w \in L(g_m)\},$$

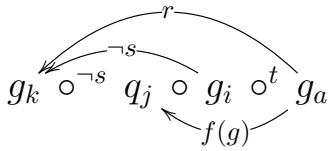
kde $\{g_1, \ldots, g_m\} \subseteq sup(\circ)$. Jazyk SA systému je poté definován jako $L(\sigma) = L(\sigma \mid sup(\circ))$.

Základní myšlenka spočívá v tom, že můžeme pro každou kaskádu (řetězec relace $\circ$) rozkladu relace $\circ$ určit vlastní jazyk. Věta je přijata kaskádou právě tehdy, když ji lze dekomponovat na podřetězce a každá komponenta v kaskádě přijme jeden podřetězec. Při této dekompozici musí být samozřejmě zachováno pořadí dané relací $\circ$. Je zřejmé, že pokud přijme i poslední komponenta, pak je věta přijata, a naopak pokud nepřijme libovolná komponenta z kaskády, věta je odmítnuta. Nepřijme-li g_1, je naprosto zbytečné ověřovat další komponenty v kaskádě $(g_2, \ldots, g_k)$. Poslední komponentu kaskády budeme nazývat jako koncovou, protože ona rozhoduje o přijetí věty. Z toho důvodu trochu nepřesně říkáme, že koncová komponenta g_k přijala či odmítla větu w. Protože celá kaskáda patří do rozkladu χ, je zřejmé, že koncová komponenta musí být vždy v $sup(\circ)$ (horní uzávěr relace $\circ$) a první komponenta g_1 musí být v $inf(\circ)$ (plyne z definice rozkladu 5.1.5). Na rozdíl od klasických gramatických systémů zde máme při specifikaci přijímaného jazyka volbu – můžeme se omezit pouze na některé koncové komponenty (pouze některé kaskády). Výčtem můžeme stanovit koncové komponenty, které musí přijmout, aby byla věta w přijata celým systémem (ostatní komponenty přijmout nemusí). V takovém případě je přijímaný jazyk konjunkcí výsledku syntaktické analýzy všech uvedených komponent. Pokud tento výčet neuvedeme, pak se předpokládá, že celá množina $sup(\circ)$, tedy všechny koncové komponenty kaskád, musí přijmout větu w, aby byla přijata celým systémem.

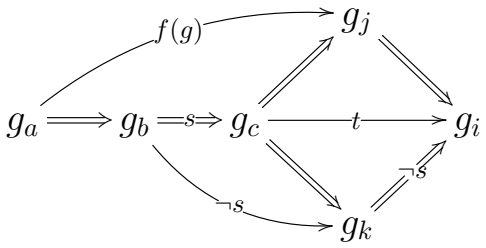
Před uvedením příkladů si ještě zavedeme konvence značení omezení. Jsou-li omezení kladena na prvky jdoucí v relaci $\circ$ za sebou, tj. $g_j \circ g_i$, pak toto omezení můžeme indikovat horním indexem nad symbolem $\circ$. Například pro synchronizaci mezi g_j a g_i použijeme zápis $g_j \circ^s g_i$. Klademe-li více omezení na stejné prvky, pak tato omezení oddělujeme čárkou, například $g_j \circ^{s,r} g_i$. Pro $g_j \in g_i^+$ můžeme použít pro zaznačení omezení šipky, jak ukazuje následující příklad pro systém zpracovávající jazyk $\{a^n b^m c^n d^m | n, m > 0\}$:



Tyto konvence lze kombinovat, například:



Pokud je relace $\circ$ větvená či komplikovanější, je vhodné zvolit alternativní reprezentaci pomocí takzvaného grafu toku. V tomto grafu jsou prvky přímo ležící v relaci $\circ$ znázorněny pomocí dvojité šipky $\Rightarrow$. Omezení mezi nepřímo sousedícími prvky (následníky) jsou značena pomocí jednoduché šipky $\rightarrow$. Tato reprezentace má jednu velikou výhodu – jednoduše lze odhalit cyklus, který by znemožnil interpretaci systému.



Nyní si je vhodné uvést si pro objasnění problematiky několik příkladů.

Příklad 5.1.9. Uvažujme jazyky:

$$
\begin{aligned}
L_{21} &= \{a^n b^n c^n \mid n \geq 1\},\\
L_{22} &= \{a^n b^m a^n b^m \mid n \geq 1\},\\
L_{23} &= \{a^n b^m c^n d^m \mid n \geq 1\},\\
L_{24} &= \{a^n b^{2^n} \mid n \geq 1\},\\
L_{25} &= \{a^n b^m a^{2n} b^{3m} \mid n \geq 1\},\\
L_{26} &= \{ww \mid w \in \{a,b\}^*\},
\end{aligned}
$$

Jsou dány gramatiky:

$$
\begin{aligned}
G_1 &= (S_1, T, \{S_1 \rightarrow aS_1 \mid a\}, S_1)\\
G_2 &= (S_2, T, \{S_2 \rightarrow bS_1c \mid bc\}, S_2)\\
G_3 &= (S_3, T, \{S_3 \rightarrow aS_3 | bS_3 | a | b\})\\
G_4 &= (S_4, T, \{S_4 \rightarrow aaS_4 | bbbS_4 | aa | bbb\})
\end{aligned}
$$

Tyto gramatiky tvoří SA systém:

$$\sigma_{21} = (\{S_1, S_2, S_3, S_4\}, \{a, b, c, d\}, \{G_1, G_2, G_3, G_4\}, \{G_1^{LR}, G_2^{LR}, G_3^{LR}, G_3^{C}, G_3^{CG}, G_4^{CG}\}, \circ, \mathcal{R})$$

Pro systém σ_{21} platí následující relace:

$$
\begin{aligned}
L_{21} &= L(G_2^{LR} \circ^s G_1^{LR}),\\
L_{22} &= L(G_3^{C} \circ^r G_3^{LR}),
\end{aligned}
$$

$$L_{23} = L(G_3^C \circ^{c(\phi)} G_3^{LR}) \text{ kde } \phi = \{(a,c),(b,d)\},$$
$$L_{24} = L(G_3^C \circ^{f(g)} G_1^{LR}), \text{ kde } g(x) = b^{2^z}, z = |x|_a$$
$$L_{25} = L(G_4^{CG} \circ^{m(\mu)} G_3^{LR}), \text{ kde } \mu = \{(S_3 \to aS_3, S_4 \to aaS_4),$$
$$(S_3 \to bS_3, S_4 \to bbbS_4), (S_3 \to a, S_4 \to aa), (S_3 \to b, S_4 \to bbb)\}$$
$$L_{26} = L(G_3^{CG} \circ^t G_3^{LR}).$$

5.1.1 Boolova algebra nad SA systémy

V naprosté většině případů bude relace lineární, ale již jsme výše naznačili, že lze komponenty SA systému propojovat do složitějších uskupení. K tomu ovšem budeme muset zavést logické operace nad $\circ$. Vyjadřovací síla SA systémů se sice nezvýší, ale zavedením logických operací v některých případech docílíme snížení složitosti SA systémů. Následně budeme moci využívat systémy s větvenou relací $\circ$.

Definice 5.1.10. Nechť $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$ je SA systém, pak zobrazení $\Lambda(\circ) : \mathcal{P} \longmapsto \lambda$ je boolova algebra nad relací $\circ$, kde $\lambda = \{\bot, \wedge, \wedge^L, \wedge^R, \vee, \vee^L, \vee^R, \neg, \}$ značí množinu operací nad relací $\circ$. SA systém je logicky platný, pokud je $\Lambda(\circ)$ injektivní zobrazení, tedy platí:

$$\forall g \in \mathcal{P}\left((\exists \lambda_1 \in \lambda : (g, \lambda_1) \in \Lambda(\circ)) \wedge (\forall \lambda_1, \lambda_2 \in \lambda, \lambda_1 \neq \lambda_2 : (g, \lambda_1) \in \Lambda(\circ) \Longrightarrow (g, \lambda_2) \notin \Lambda(\circ))\right).$$

Zavádíme 4 základní logické operace – logické *a* ($\wedge$, *and*), logické *nebo* ($\vee$, *and*), negaci ($\neg$, *not*) a identitu ($\bot$, *bottom*), kterou budeme chápat jako prázdnou operaci. Operaci $\bot$ zavádíme, aby bylo možné popsat systém pomocí injektivního zobrazení. Budeme totiž požadovat, aby každá komponenta z $\mathcal{P}$ měla přiřazena právě jednu operaci z λ. Způsob přiřazení operací a význam operací na komponentami si nyní zavedeme. Nechť $g \in \mathcal{P}$ je zpracovávaná komponenta, této komponentě přiřadíme operaci:

- spojuje-li komponenta g relaci $\circ$, tj. $\exists g_i, g_j \in \mathcal{P}, g_i \neq g_j : (g, g_i) \in \circ \wedge (g, g_j) \in \circ$, pak $\lambda \in \{\wedge, \vee\}$, respektive $(g, \wedge) \in \Lambda(\circ)$ nebo $(g, \vee) \in \Lambda(\circ)$. Zřejmě tedy všechny komponenty, které mají více než jednoho přímého předchůdce, budou mít přiřazenou jednu z operací $\wedge$ či $\vee$. Horním indexem L či R označujeme levou či pravou prioritu při vyhodnocování. Je zřejmé, že komponenta g bude pracovat poté, co svoji práci dokončily komponenty g_i i g_j. Logická operace nad g udává, jak bude komponenta g zacházet s výstupem komponent g_i a g_j.
  - $\wedge$ – komponenta g bude pracovat pouze, pokud g_i i g_j přijaly a zároveň řetězec w_i přijatý g_i se shoduje s řetězcem w_j přijatým g_j.
  - $\wedge^L$ – komponenta g bude pracovat pouze, pokud g_i i g_j přijaly, komponenta g bude poté pokračovat v práci na větné formě v místě, kde skončila g_i.
  - $\wedge^R$ – komponenta g bude pracovat pouze, pokud g_i i g_j přijaly, komponenta g bude poté pokračovat v práci na větné formě v místě, kde skončila g_j.
  - $\vee$ – komponenta g bude pracovat pouze, pokud některá z komponent g_i či g_j přijala. V případě, že přijaly obě tyto komponenty, g_i přijala větu w_i, g_j přijala větu w_j, pak bude g pokračovat pouze pokud $w_i = w_j$.
  - $\vee^L$ – komponenta g bude pracovat pouze, pokud některá z komponent g_i či g_j přijala. V případě, že přijaly obě tyto komponenty, bude komponenta g pokračovat v práci na větné formě tam, kde skončila komponenta g_i.

* $\vee^R$ – komponenta g bude pracovat pouze, pokud některá z komponent g_i či g_j přijala. V případě, že přijaly obě tyto komponenty, bude komponenta g pokračovat v práci na větné formě tam, kde skončila komponenta g_j.

- Nemá-li komponenta g předchůdce, tj. $g \in inf(\circ)$, pak $\lambda = \bot$, respektive $(g, \bot) \in \Lambda(\circ)$. Komponenty z dolního uzávěru nemají žádné předchůdce a tím pádem se s jejich vstupy nemusíme zaobírat.

- Má-li komponenta g právě jednoho přímého předchůdce g_i, tj. $\exists g_i \in \mathcal{P} : (g, g_i) \in \circ$, pak $\lambda \in \{\neg, \bot\}$, respektive $(g, \bot) \in \Lambda(\circ)$, nebo $(g, \neg) \in \Lambda(\circ)$.

  * $\bot$ – komponenta g bude pracovat pouze pokud g_i přijme a bude pokračovat od symbolu větné formy, kde g_i skončila. Vskutku se jedná o prázdnou operaci, neboť naprosto stejně interpretujeme SA systémy bez rozšíření v podobě logických operací.

  * $\neg$ – komponenta g bude pracovat, pouze pokud g_i odmítne a bude pokračovat od prvního symbolu zpracovávaného komponentou g_i. Negaci lze tedy chápat jako další pokus o zpracování větné formy počínaje stejným symbolem. Touto operací jsme zkompletovali Booleovu algebru nad SA systémy.

5.2 Návrh bezkontextového SA systému

Nyní se zaměříme na způsob, jak pomocí SA systému popsat zpracovávaný jazyk. Samozřejmě tento proces nelze zcela zautomatizovat, nastíníme tedy pouze základní myšlenku. Chceme-li pomocí SA systému založeného na deterministických konečných automatech zpracovat jazyk, jenž není bezkontextový, musíme nutně provést dekompozici tohoto jazyka na několik bezkontextových jazyků. Tuto dekompozici provedeme tak, abychom závislosti mezi jednotlivými bezkontextovými jazyky mohli vyjádřit pomocí omezení $\mathcal{R}$ nebo přímo relace $\circ$. K dekompozici lze využít:

1. rozklad na podsekvence – jazyk $L = \{ab\}$ rozdělíme na $L_1 = \{a\}$, $L_2 = \{b\}$, sestavíme gramatiky G_{L_1}, G_{L_2} a analyzátory $G^A_{L_1}$, $G^A_{L_2}$, které přidáme do relace $\circ$. Potom platí, že $L(G^A_{L_2} \circ G^A_{L_1}) = L$.

2. vytýkání stejného vzoru – opakuje-li se v libovolném jazyce stejný vzor ∂, pokusíme se jej vytknout (budeme k němu přistupovat jako k proměnné), tj. jeho první výskyt ponecháme beze změny a libovolný další výskyt označíme $\bullet$. Proměnnou ∂ jazyka L_i značíme pomocí notace $\partial \parallel L_i$. Dále využijeme rozklad na podsekvence tak, aby platilo:

   (a) každá podsekvence obsahuje nejvýše jeden originální vzor ∂,

   (b) lze-li do podsekvence obsahující ∂ zahrnout i některý další výskyt $\bullet\partial$, učiníme tak. Je-li v podsekvenci originální vzor ∂ i odkazovaný $\bullet\partial$, pak řekneme, že podsekvence obsahuje *interně odkazovaný vzor*. Obsahuje-li podsekvence pouze $\bullet\partial$ a nikoliv originální vzor, pak se jedná o *externě odkazovaný vzor*.

   (c) každá podsekvence obsahuje nejvýše jeden externě odkazovaný vzor $\bullet\partial$,

   (d) nejsložitější využitelná podsekvence bude tedy obsahovat originální vzor, externě odkazovaný vzor (libovolný počet výskytů) a interně odkazovaný vzor (libovolný

počet výskytů). Například $n, \bullet m \parallel \{a^n c^{\bullet n} b^{\bullet m} c^{\bullet m}\}$ je naprosto v pořádku, ale $\bullet n, \bullet m \parallel \{a^{\bullet n} c^{\bullet n} b^{\bullet m} c^{\bullet m}\}$ je špatně zvolená podsekvence, neboť obsahuje dva externí vzory (každý má 2 výskyty). Zároveň je nutno podotknout, že správně zvolená podsekvence nemusí být nutně bezkontextová, proto je vhodné mnohdy volit rozklad na více jednodušších podsekvencí.

Poté navrhneme pro každou podsekvenci takovou gramatiku, která ji generuje a sestavíme kaskádu analyzátorů, jež přijímá původní sekvenci. První analyzátor v řadě pokryje vždy originální vzor ∂, pro každý další analyzátor pokrývající externě odkazovaný vzor $\bullet\partial$ zavedeme omezení (od analyzátoru pokrývající originální vzor ∂). Formálně: $\partial \parallel L_1 \wedge \bullet\partial \parallel L_2 \iff (G_{L_2}, G_{L_1}, A) \in \mathcal{R}$, kde $G_{L_1}, G_{L_2} \in \mathcal{P}$, $L_1 = L(G_{L_1})$ a $L_2 = L(G_{L_2})$.

3. nahrazení (substituce) – máme-li jazyk L_1 nad Σ_1 a jazyk L_2 nad Σ_2, pro něž lze nalézt izomorfismus $\Psi : \Sigma_1 \longleftrightarrow \Sigma_2$, aby platilo $L_1 = \Psi(L_2)$, pak můžeme provést substituci. Jinak řečeno, sestrojíme množinu náhrad $\Psi \subseteq \Sigma_1 \times \Sigma_2$ takovým způsobem, abychom záměnou všech symbolů ze Σ_1 za jejich obraz ve větě patřící do L_1 získali větu z L_2. Po nalezení takového morfismu lze zavést omezení ve formě konverze nebo aplikace funkce.

Příklad 5.2.1. Uvedenou techniku si ukážeme na jazyku $L = \{a^n b^m c^n d^m \mid n, m > 0\}$. Provedeme vytýkání pro n a m, dostáváme tedy:

$$n, m \parallel \{a^n b^m c^{\bullet n} d^{\bullet m}\}.$$

Máme vytknuté 2 proměnné, pro každou z nich samostatně sestavíme gramatiku pomocí rozkladu na podsekvence. Dostáváme:

$$\begin{aligned} n \parallel L_1 &= \{a^n \mid n > 0\}, \\ m \parallel L_2 &= \{b^m \mid m > 0\} \text{ a} \\ \bullet n, \bullet m \parallel L_3 &= \{c^{\bullet n}, d^{\bullet m} \mid \bullet\, n, \bullet m > 0\}. \end{aligned}$$

Jazyk L_3 nyní obsahuje dvě odkazované proměnné, proto jej dále rozdělíme na:

$$\begin{aligned} \bullet n \parallel L_3 &= \{c^{\bullet n} \mid \bullet\, n > 0\} \text{ a} \\ \bullet m \parallel L_4 &= \{d^{\bullet m} \mid \bullet\, m > 0\}. \end{aligned}$$

Nyní pro každé L_i, $i \in \{1, 2, 3, 4\}$ sestrojíme gramatiku a analyzátor:

$$\begin{aligned} G_1 &= (\{S\}, \{a\}, \{S \to aS|\varepsilon\}, S), \\ G_2 &= (\{S\}, \{b\}, \{S \to bS|\varepsilon\}, S), \\ G_3 &= (\{S\}, \{c\}, \{S \to cS|\varepsilon\}, S), \\ G_4 &= (\{S\}, \{d\}, \{S \to dS|\varepsilon\}, S), \\ \mathcal{P} &= \left\{G_1^{LL}, G_2^{LL}, G_2^{LL}, G_2^{LL}\right\}. \end{aligned}$$

Pro každý odkaz na proměnnou $(\bullet n, \bullet m)$ zavedeme omezení mezi analyzátorem, kde se tento odkaz vyskytuje, a analyzátorem, kde se vyskytuje originální vzor, tj:

$$\mathcal{R} = \left\{\left(G_1^{LL}, G_3^{LL}, s\right), \left(G_2^{LL}, G_4^{LL}, s\right)\right\}.$$

Dostáváme tedy systém s následujícím propojením komponent:

$$G_3^{LL} \overset{s}{\circ} G_3^{LL} \circ G_2^{LL} \overset{s}{\circ} G_1^{LL}$$

V tomto bodě je vhodné zmínit, že existuje celá řada správných výsledků a mnoho způsobů, jak se k nim dobrat. Například stejný jazyk přijímá i systém, kde místo synchronizace použijeme omezení typu t (sledování). Místo komponent G_2^{LL} a G_4^{LL} lze využít například G_2^{CG} a G_4^{CG}. Další možností je použít místo všech LL analyzátorů komponenty typu LR. Naprosto odlišný výsledek dostaneme, pokud jako první krok zvolíme rozklad na podsekvence místo vytýkání, pak:

$$L_1 = \{a^n b^m \,|\, n, m > 0\},$$
$$L_2 = \{c^n d^m \,|\, n, m > 0\}$$

Poté můžeme jednoduše zavést substituci Ψ:

$$L_2 = \Psi(L_1), \Psi = \{(a, c), (b, d)\}.$$

Poté sestrojíme gramatiky a analyzátory:

$$G_1 = \{\{S, A, B\}, \{a, b\}, \{S \to AB, A \to aA \,|\, \varepsilon, B \to bB \to \varepsilon\}, S\},$$
$$G_2 = \{\{S, C, D\}, \{c, d\}, \{S \to CD, C \to cC \,|\, \varepsilon, D \to dD \to \varepsilon\}, S\},$$
$$\mathcal{P} = \left\{G_1^{LL}, G_2^{C}\right\}$$

a zavedeme omezení:

$$\mathcal{R} = \left\{\left(G_1^{LL}, G_2^{C}, c(\Psi)\right)\right\}.$$

Výsledný systém tedy je $G_2^C \circ^{c(\Psi)} G_1^{LL}$. I v tomto případě lze dojít k jinému výsledku, místo konverze můžeme využít sledování a komponentu typu CG.

5.3 Konstrukce a analýza bezkontextového SA systému

V kapitole 5.2 jsme si objasnili návrh SA systému, nyní se přesuneme k jeho realizaci – ukážeme si, jak správně sestavit komponenty systému, a analyzujeme chování systému z hlediska determinismu. Tato analýza je podstatná pro správnou činnost SA systému. Bezkontextové SA systémy mohou bez problému zpracovat kontextové jazyky, ovšem realizujeme-li tyto systémy pomocí deterministických zásobníkových automatů, pak lze přijímat pouze jazyky z rodiny DCS. V některých případech lze rozšířit sílu na celou rodinu CS za cenu značného zesložitění systému.

5.3.1 Množiny $Empty$, $First$, $Last$, $Next_{SA}$ a $Follow_{SA}$

Ke konstrukci klasických bezkontextových analyzátorů typu LL (*left-to-right, leftmost derivation*) či LR (*left-to-right, rightmost derivation*) se zpravidla využívají množiny $First$, $Empty$ a $Follow$, které zajisté není třeba pozornému čtenáři představovat. Jsou-li ovšem tyto analyzátory konstruovány v rámci SA systému, budou se tyto množiny v některých drobných nuancích lišit, a proto jako součást této práce uvádíme jejich formální definici

a algoritmy, pomocí nichž lze tyto množiny vypočíst. Dále zavedeme úplně nové množiny $Last$ a $Next$.

Uvedené množiny se počítají pro každý model zvlášť, potažmo pro každou komponentu zvlášť. V případě, že nezáleží na propojení mezi komponentami (topologii systému, respektive relaci $\circ$), můžeme množinu spočítat pro model a dále ji využijeme k sestrojení všech komponent nad tímto modelem (libovolný počet instancí LL či LR) – to je případ množin $Empty$, $First$, $Last$. Množinu $x \in \{First, Empty, Last\}$ nad modelem $G_i \in \mathcal{G}$ budeme značit x^{G_i} (čímž myslíme, že je tato množina platí pouze pro model G_i). U množin $Next$ a $Follow$ záleží i na propojení komponent relací $\circ$ v rámci SA systému (topologii SA systému), což budeme indikovat pomocí označení SA zapsaného dolním indexem. Tyto množiny se tedy mohou lišit mezi různými instancemi komponent nad jedním modelem. Pro naznačení, že množina $x \in \{First, Last, Empty, Next_{SA}, Follow_{SA}\}$ je spočtena pro komponentu $g_{i|j} \in \mathcal{P}$, budeme zapisovat $x^{g_{i|j}}$. Z praktického hlediska může být lepší spočítat všechny uvedené množiny pro každou komponentu a nezabývat se tím, zdali komponenty sdílí či nesdílí model.

Množina $First$, zavedená v sekci 4.2.6, a množina $Empty$, zavedená níže, jsou definovány i využity zcela běžným způsobem.

Definice 5.3.1. Nechť $G = (N, T, S, P)$ je BKG a $X_1X_2 \dots X_n \in (N \cup T)^*$ je řetězec. $Empty(X_1X_2 \dots X_n)$[6] definujeme následovně:

$$Empty(X_1X_2 \dots X_n) = \{\varepsilon\} \textbf{ if } X_1X_2 \dots X_n \Rightarrow^* \varepsilon \textbf{ else } \emptyset.$$

Pro připomenutí uvádíme význam těchto množin – $First^{G_i}(x)$ obsahuje všechny symboly $a \in T_i$, které se mohou vyskytovat na nejlevější pozici v x (x je generováno modelem G_i). $Empty^{G_i}(x)$ poté udává, zdali je možné řetězec x s použitím modelu G_i vymazat pomocí ε-pravidel. Pro výpočet $Empty^{G_i}(x)$ lze využít algoritmus 5.3.1 uvedený v [6].

Algoritmus 5.3.1: Množina $Empty$

Vstup: BKG (model) $G_i = (N_i, T_i, S_i, P_i)$, $1 \leq i \leq |\mathcal{G}|$
Výstup: množina $Empty^{G_i}(X)$ pro každé $X \in N_i \cup T_i$

1: $\forall a \in T_i$: $Empty^{G_i}(a) := \emptyset$
2: $\forall A \in N_i$: $Empty^{G_i}(A) := \{\varepsilon\}$ **if** $A \to \varepsilon \in P_i$ **else** $\emptyset$
3: **while** některá množina $Empty^{G_i}(X)$ změněna **do**
4: **if** $A \to X_1X_2 \dots X_m \in P_i$ **and** $\forall j \in \{1, \dots, m\} : Empty^{G_i}(X_j) = \{\varepsilon\}$ **then**
5: $Empty^{G_i}(A) := \{\varepsilon\}$
6: **end if**
7: **end while**

$First^{G_i}(x)$ je možné vyčíslit pomocí algoritmu 5.3.2 uvedeného v [6]:

Algoritmus 5.3.2: Množina $First$

Vstup: BKG (model) $G_i = (N_i, T_i, S_i, P_i)$, $1 \leq i \leq |\mathcal{G}|$
Výstup: množina $First^{G_i}(A)$ pro každé $X \in N_i \cup T_i$

1: $\forall a \in T_i$: $First^{G_i}(a) := \{a\}$
2: $\forall A \in N_i$: $First^{G_i}(A) := \emptyset$
3: **while** některá množina $First^{G_i}(A)$ změněna **do**
4: **if** $A \to X_1 X_2 \ldots X_{k-1} X_k \ldots X_n \in P_i$ **then**
5: $First^{G_i}(A) := First^{G_i}(A) \cup First^{G_i}(X_1)$
6: **if** $Empty^{G_i}(X_1 X_2 \ldots X_{k-1}) = \{\varepsilon\}$ **then**
7: $First^{G_i}(A) := First^{G_i}(A) \cup First^{G_i}(X_k)$
8: **end if**
9: **end if**
10: **end while**

Zatímco množina $First$ nám udává nejlevější (počáteční) symboly řetězce, pro analýzu determinismu budeme potřebovat i nejpravější (poslední) symboly. Z toho důvodu zavádíme množinu $Last$. Na rozdíl od $First$ je zde využita nejpravější derivace.

Definice 5.3.2. Nechť $G = (N, T, S, P)$ je BKG. Pro každé $x \in (N \cup T)^*$ je definováno $Last(x)$ [6] jako:

$$Last(x) = \left\{ a \,|\, a \in T,\, x \Rightarrow^* ya,\, y \in (N \cup T)^* \right\}.$$

Pro každý model budeme chtít spočítat $Last^{G_i}(S)$, abychom zjistili, kterými všemi symboly mohou končit věty tímto modelem generované. K tomu účelu použijeme algoritmus 5.3.3.

Algoritmus 5.3.3: Množina $Last$

Vstup: BKG (model) $G_i = (N_i, T_i, S_i, P_i)$, $1 \leq i \leq |\mathcal{G}|$
Výstup: množina $Last^{G_i}(A)$ pro každé $X \in N_i \cup T_i$

1: $\forall a \in T_i$: $Last^{G_i}(a) := \{a\}$
2: $\forall A \in N_i$: $Last^{G_i}(A) := \emptyset$
3: **while** některá množina $Last^{G_i}(A)$ změněna **do**
4: **if** $A \to X_1 X_2 \ldots X_k X_{k+1} \ldots X_m \in P_i$ **then**
5: $Last^{G_i}(A) := Last^{G_i}(A) \cup Last^{G_i}(X_m)$
6: **if** $Empty^{G_i}(X_m \ldots X_{k+1}) = \{\varepsilon\}$ **then**
7: $Last^{G_i}(A) := Last^{G_i}(A) \cup Last^{G_i}(X_k)$
8: **end if**
9: **end if**
10: **end while**

Nyní přejdeme k tvorbě množin nad komponentami – nad analyzátory $g_i \in \mathcal{P}$ zavedeme zcela novou množinu $Next_{SA}^{g_i}$. Tato množina se stanovuje v rámci SA systému $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$ a udává, které všechny symboly $a \in T$ se mohou vyskytovat za částí vstupu zpracovanou analyzátorem g_i (podsekvencí přijímanou g_i). Jinak řečeno, jedná se o takové symboly, které ve vstupní větě plní roli znaku konce vstupu \$ pro komponentu g_i. Narazí-li g_i na nějaký symbol z této množiny, je jasné, že nyní může pokračovat další

analyzátor v relaci $\circ$. Tato množina se tedy dále použije k určení momentu, kdy se předává řízení mezi analyzátory.

Definice 5.3.3. Nechť σ je SA systém, potom pro každý model $g_i \in \mathcal{P}$ definujeme $Next_{SA}^{g_i} \subseteq T \cup \{\$\}$ jako:

$$Next_{SA}^{g_i} = \begin{cases} \bigcup_{\forall g_j \in F_G} First^{g_j}(S_j) & \text{pokud } F_G \neq \emptyset, \\ \{\$\} & \text{pokud } F_G = \emptyset, \end{cases}$$
$$\text{kde } F_G \subseteq \mathcal{P} \wedge g_j \in F_G \iff (g_j, g_i) \in \circ.$$

Množina $Next_{SA}^{g_i}$ tedy obsahuje všechny takové symboly, které se mohou vyskytovat na první ($First$) pozici v libovolné větě přijímané nějakou navazující komponentou g_j ($g_j \circ g_i$). Postup pro vypočtení množiny $Next$ nám udává přímo její definice, z toho důvodu neuvádíme algoritmus samostatně. Začínáme-li s prázdnými množinami $Next_{SA}^{g_i}$ (pro každé $g_i \in \mathcal{P}$), pak stačí v jedné iteraci projít celou relaci $\circ$. Pro každý prvek této relace (g_j, g_i) pouze přidáme množinu $First^{g_j}$ do množiny $Next_{SA}^{g_i}$. Na závěr provedeme korekci tak, že do všech prázdných množin $Next_{SA}^{g_i}$ vložíme \$.

Poslední zaváděná množina $Follow_{SA}$ má velice podobný význam jako množina $Next_{SA}$, ale zatímco $Next_{SA}$ je zavedena nad celými syntaktickými analyzátory, $Follow_{SA}$ se spočítá pro každý symbol analyzátoru zvlášť.

Definice 5.3.4. Nechť σ je SA systém, potom pro každý analyzátor $g_i \in \mathcal{P}$ definujeme $Follow_{SA}^{g_i} \subseteq T \cup \{\$\}$ jako:

$$Follow_{SA}^{g_i}(A) = \{a \in T \mid S \Rightarrow^* xAay, x, y \in (N \cup T)^*\}$$
$$\cup\, (Next(g_i) \text{ pokud } S \Rightarrow^* xA, x \in (N \cup T)^* \text{ jinak } \emptyset)\,.$$

Na rozdíl od běžné definice $Follow$ zde není zahrnut pouze jeden symbol konce vstupu, ale celá množina těchto symbolů. Znak konce vstupu \$ bude ve $Follow_{SA}^{g_i}$ zahrnut pouze tehdy, jedná-li se o poslední komponentu v kaskádě, tj. v $\circ$ na ni nenavazuje žádná jiná komponenta ($\$ \in Follow_{SA}^{g_i}(S) \iff \neg(\exists g_j \in \mathcal{P} : (g_j, g_i) \in \circ)$). Výpočet $Follow_{SA}^{g_i}$ je naprosto shodný se standardním výpočtem $Follow$ s tím rozdílem, že v počátečních podmínkách uvedeme místo $Follow(S) = \{\$\}$ celou množinu $Next_{SA}^{g_i}$, tj. $Follow_{SA}^{g_i}(S) = Next_{SA}^{g_i}$.

Konstrukci komponenty typu LL s využitím množin $First$, $Empty$ a $Follow_{SA}$ si ukážeme v další kapitole. Množiny $First$ a $Follow_{SA}$ se dále využijí k sestavení akční (α) a přechodové (β) tabulky LR analyzátoru.

5.3.2 Konstrukce LL komponenty

Pro potřeby konstrukce komponenty typu LL zavádíme ještě jednu množinu – $Predict_{SA}^{g_i}$, kterou definujeme pomocí výše zavedených množin.

Definice 5.3.5. Pro každou komponentu $g_i^{LL} \in \mathcal{P}$ a pro každé pravidlo $A \to x \in P_i$, kde $G_i = (N_i, T_i, S_i, P_i)$ je model komponenty g_i^{LL}, zavádíme množinu $Predict_{SA}^{g_i}$ [6] jako:

$$Predict_{SA}^{g_i}(A \to x) = \begin{cases} First^{g_i}(x) \cup Follow_{SA}^{g_i} & \text{pokud } Empty^{g_i}(x) = \{\varepsilon\}, \\ First^{g_i}(x) & \text{pokud } Empty^{g_i}(x) = \emptyset. \end{cases}$$

Uvedená množina $Predict$ se od definice dostupné v literatuře liší pouze tím, že ji zavádíme zvlášť pro každou instanci. Nyní už lze sestavit LL tabulku α^{g_i} komponenty g_i, a to zcela běžným způsobem:

$$\alpha^{g_i}(A, a) = A \to X_1X_2 \dots X_m \in P_i \iff a \in Predict^{g_i}_{SA}(A \to X_1X_2 \dots X_m \in P_i).$$

Je zřejmé, že komponenty typu LL lze využít pouze, pokud jejich model G_i je LL gramatika. V literatuře [6] se LL gramatika chápe jako gramatika splňující podmínku:

> pro každé $a \in T$ a každé $A \in N$ existuje maximálně jedno A-pravidlo tvaru $A \to X_1X_2 \dots X_m \in P$ takové, že platí $a \in Predict(A \to X_1X_2 \dots X_m)$.

Tato podmínka je dostačující, abychom dokázali zkonstruovat LL analyzátor g_i odtržený od zbytku SA systému. Pro SA systémy jsme ovšem množinu $Predict$ definovali za pomoci relace $\circ$, podmínka LL gramatiky bude tedy upravena na:

> pro každé $a \in T_i$ a každé $A \in N_i$ existuje maximálně jedno pravidlo tvaru $A \to X_1X_2 \dots X_m \in P_i$ takové, že platí $a \in Predict^{g_i}_{SA}(A \to X_1X_2 \dots X_m)$.

Z toho plyne jedno výrazné omezení, jež není na první pohled patrné. Aby bylo možné přídat analyzátor g_i do relace $\circ$ s analyzátorem g_j, tj. $(g_j, g_i) \in \circ$, musí platit následující implikace:

$$A \to ax \in P_i \wedge a \in Next^{g_j}_{SA} \Longrightarrow Empty^{g_i}(A) = \emptyset, \tag{5.1}$$

kde $x \in (N_i \cup T_i)^*$. Tuto implikaci lze s využitím výrokového počtu převést do tvaru:

$$A \to ax \in P_i \wedge Empty^{g_i}(A) = \{\varepsilon\} \Longrightarrow a \notin Next^{g_j}_{SA}. \tag{5.2}$$

Důvod zavedení této implikace je prostý – pokud bychom mohli odvodit symbol a pomocí A-pravidla komponentou g_i (A je na zásobníku g_i, ale lze ho odstranit pomocí ε-pravidla) a zároveň říkáme, že následující komponenta umí odvodit a ($a \in Next(g_j)$, $g_j \circ g_i$), pak zde máme nedeterminismus. V takovém případě nelze určit, jestli je lepší zpracovat a současnou komponentou, nebo už s a zacházet jako s ukončujícím symbolem podsekvence, odstranit A pomocí ε-pravidla a předat řízení dalšímu analyzátoru.

Příklad 5.3.6. Nechť existují dvě komponenty g_1, g_2 přičemž platí $g_2 \circ g_1$. Nechť existují pravidla $S_1 \to A_1 \in P_1$, $A_1 \to ax \in P_1$, $A_1 \to \varepsilon \in P_1$ a zároveň platí $First^{g_2}(S_2) = \{a\}$. Předpokládejme, že g_1 je LL komponenta, tj. existuje nejvýše jedno A-pravidlo p takové, že $a \in Predict^{g_1}_{SA}(p)$. Na první pohled je patrné, že:

$$\begin{aligned} First^{g_1}(A_1) &= \{a\}, \\ First^{g_1}(S_1) &= \{a\}, \\ Empty^{g_1}(A_1) &= \{\varepsilon\}, \\ Empty^{g_1}(S_1) &= \{\varepsilon\}. \end{aligned}$$

Nyní spočteme $Next^{g_1}_{SA}$, podle definice platí:

$$Next^{g_1}_{SA} = \bigcup_{g_j \in F_G} First^{g_j}(S_j).$$

Je zřejmé, že $F_G = \{g_2\}$, protože $(g_2, g_1) \in \circ$, a tedy:

$$Next_{SA}^{g_1} = \bigcup_{g_j \in \{g_2\}} First^{g_j}(S_j) = First^{g_2}(S_2) = \{a\}.$$

Dále podle definice 5.3.4 stanovíme $Follow_{SA}^{g_1}$:

$$Follow_{SA}^{g_1}(S_1) = Follow_{SA}^{g_1}(A_1) = Next_{SA}^{g_1} = \{a\}.$$

Pak nutně (podle definice 5.3.5) platí:

$$Predict_{SA}^{g_1}(A_1 \to ax) = First^{g_1}(ax) \cup Follow_{SA}^{g_1}(A_1) = \{a\} \cup \{a\} = \{a\},$$
$$Predict_{SA}^{g_1}(A_1 \to \varepsilon) = \emptyset \cup Follow_{SA}^{g_1}(A_1) = \emptyset \cup \{a\} = \{a\}.$$

Tedy existují 2 A-pravidla taková, že a patří do $Predict_{SA}^{g_1}$ těchto pravidel, což je spor.

5.3.3 Deterministické SA systémy

Stavíme-li SA systém nad deterministickými zásobníkovými automaty, musíme pro správnou činnost tohoto systému zajistit, aby (i) veškeré jeho komponenty byly deterministické samy o sobě a (ii) libovolné dvě komponenty v relaci $\circ$ byly deterministicky provázány. Typicky se v systému bude vyskytovat jedna či více komponent pro každou podsekvenci jazyka přijímaného SA systémem (5.2).

První bod splníme tak, že analyzujeme všechny tyto podsekvence, tedy jazyky přijímané jednotlivými komponentami. Tato analýza se neliší od analýzy, kterou bychom prováděli při tvorbě bezkontextového analyzátoru. Pro každou podsekvenci sestrojíme DKA, který ji přijímá. Jsme-li schopni sestavit obecný DKA pro danou podsekvenci, můžeme poté zkusit sestrojit komponenty typu LL či LR pomocí zavedených metod bezkontextové analýzy (volba typu komponenty se odvíjí od toho, jestli podsekvence patří do třídy jazyků LL). Pokud nebude podsekvence deterministická, tj. nelze sestavit deterministickou komponentu, pak ji nelze v deterministickém SA systému použít. Pro některé nedeterministické podsekvence lze tuto situaci řešit – můžeme místo jednoho DKA zavést dvojici analyzátorů. První analyzátor provede hrubou analýzu s cílem zjistit nějakou informaci (například délku podsekvence či počet výskytů symbolu) a druhý analyzátor navázaný pomocí relace $\circ^{f(g)}$ již s touto informací může pracovat (například lze určit střed). Aplikace funkce nám poskytuje Turingovskou výpočetní sílu, tedy lze zpracovat podstatně složitější jazyky (přijímané LOA či TS). Tento model je velice těžko implementovatelný, ovšem pro konečný vstup lze nalézt řešení (v praxi mívá počítačový program konečný, i když shora neomezený počet řádků).

Druhý bod bude vždy splněn, pokud bude jazyk přijímaný systémem vnitřně deterministický, tj. bude možné jednoznačně určit podsekvence při dekompozici jazyka. Rozdělíme-li jazyk $L = \{xy\}$ na podsekvence x a y, kde x je zpracováváno analyzátorem g_i (y je zpracováno analyzátorem g_j, $g_j \circ g_i$), pak musí platit:

$$Last^{g_i}(S_i) \cap Next_{SA}^{g_i} = \emptyset. \tag{5.3}$$

Lze-li takovou dekompozici nalézt, je jazyk vnitřně deterministický. Jednoduše řečeno, každý syntaktický analyzátor musí mít za svojí větou takový ukončující symbol, který se nemůže vyskytnout na konci této věty, tj. žádný symbol věty nelze zaměnit s ukončujícím symbolem. Samozřejmě z definice $Next_{SA}$ plyne, že za posledním analyzátorem v kaskádě bude následovat znak \$.

Příklad 5.3.7. Chceme sestavit SA systém pro jazyk $L = \{ww \,|\, w \in \{a,b\}^*\}$. Provedeme vytýkání originálního vzoru

$$w \parallel L = \{w \bullet w \,|\, w \in \{a,b\}^*\},$$

a následně rozklad na podsekvence:

$$\begin{aligned} w \parallel L_1 &= \{w \,|\, w \in \{a,b\}^*\}, \\ \bullet w \parallel L_2 &= \{\bullet w \,|\, w \in \{a,b\}^*\}. \end{aligned}$$

Poté navrhneme gramatiku, sestavíme analyzátory a zavedeme omezení:

$$\begin{aligned} G_1 &= (\{S\}, \{a,b\}, \{S \to aS \,|\, bS \,|\, \varepsilon\}, S), \\ \mathcal{P} &= \{G_1^{LL}, G_1^C\}, \\ \circ &= \{(G_1^C, G_1^{LL})\}, \\ \mathcal{R} &= \{(G_1^{LL}, G_1^C, r)\} \end{aligned}$$

Nyní spočteme množiny $First$, $Last$ a $Next$:

$$\begin{aligned} First^{G_1^{LL}}(S) = \{a,b\}, First^{G_1^C}(S) = \{a,b\}, \\ Last^{G_1^{LL}}(S) = \{a,b\}, Last^{G_1^C}(S) = \{a,b\}, \\ Next_{SA}^{G_1^{LL}} = First^{G_1^C} = \{a,b\}, Next_{SA}^{G_1^C} = \{\$\}. \end{aligned}$$

Je zřejmé, že $Last^{G_1^{LL}}(S) \cap Next_{SA}^{G_1^{LL}} = \{a,b\} \neq \emptyset$, a proto je jazyk L vnitřně nedeterministický.

Na druhou stranu pokud zvolíme jazyk $\overline{L} = \{w \otimes w \,|\, w \in \{a,b\}^*\}$. Pak provedeme vytýkání originálního vzoru a rozklad na podsekvence:

$$\begin{aligned} w \parallel \overline{L} &= \{w \otimes \bullet w \,|\, w \in \{a,b\}^*\}, \\ w \parallel \overline{L_1} &= \{w \,|\, w \in \{a,b\}^*\}, \\ \emptyset \parallel \overline{L_2} &= \{\otimes\}, \\ \bullet w \parallel \overline{L_3} &= \{\bullet w \,|\, w \in \{a,b\}^*\}, \end{aligned}$$

Poté navrhneme gramatiky, sestavíme analyzátory a zavedeme omezení:

$$\begin{aligned} G_1 &= (\{S\}, \{a,b\}, \{S \to aS \,|\, bS \,|\, \varepsilon\}, S), \\ G_2 &= (\{S\}, \{\otimes\}, \{S \to \otimes\}, S), \\ \mathcal{P} &= \{G_1^{LL}, G_1^C, G_2^{LL}\}, \\ \circ &= \{(G_2^{LL}, G_1^{LL}), (G_1^C, G_2^{LL})\}, \\ \mathcal{R} &= \{(G_1^{LL}, G_1^C, r)\} \end{aligned}$$

Nyní spočteme množiny $Last$ a $Next$:

$$\begin{aligned} First^{G_1^{LL}}(S) = \{a,b\}, First^{G_1^C}(S) = \{a,b\}, First^{G_2^{LL}}(S) = \{\otimes\}, \\ Last^{G_1^{LL}}(S) = \{a,b\}, Last^{G_1^C}(S) = \{a,b\}, Last^{G_2^{LL}}(S) = \{\otimes\}, \end{aligned}$$

$$
\begin{aligned}
Next_{SA}^{G_1^{LL}} &= First^{G_2^{LL}} = \{\otimes\},\\
Next_{SA}^{G_1^{C}} &= \{\$\},\\
Next_{SA}^{G_2^{LL}} &= First^{G_1^{C}} = \{a, b\}.
\end{aligned}
$$

Je zřejmé, že pro všechny komponenty platí $Last \cap Next_{SA} = \emptyset$, proto je jazyk vnitřně deterministický.

Pomocí rovnice 5.3 lze odhalit nedeterminismus uvnitř SA systému, respektive nedeterministickou konstrukci uvnitř přijímaného jazyka. Jedná se o podmínku, kterou musí splňovat všechny komponenty systému (v relaci $\circ$). Splnění této implikace není dostačující pro konstrukci LL komponent – ty musí navíc splňovat rovnici 5.1, potažmo rovnici 5.3.2. Je vhodné zmínit, že pokud analýzu determinismu neprovedeme, bude možné systém zkonstruovat, jenže přijímaný jazyk bude odlišný od zamýšleného. Často pak dojde k situaci, kdy systém přijímá prázdný jazyk, což platí i pro výše uvedený příklad 5.3.7, tedy jazyk $\{ww \mid w \in \{a, b\}^*\}$.

Jak jsme si již uvedli na příkladě 5.3.7, vnitřně nedeterministický jazyk může být sestaven z deterministických podsekvencí. Nedeterminismus na úrovni celého jazyka lze také v některých případech řešit obdobným způsobem, jako nedeterminismus na úrovni jedné podsekvence (jednoho analyzátoru). Opět můžeme některý z analyzátorů nahradit kaskádou analyzátorů, kde první analyzátor v kaskádě provede pouze hrubou analýzu s cílem zjistit nějakou informaci a další analyzátory v kaskádě budou s touto informací již pracovat (informace se předává pomocí omezení). Tento postup nelze zcela automatizovat, ale uvedeme si dvě typická řešení:

- řešení pomocí synchronizace – tento přístup využíváme, je-li předávaná informace numerická, tj. například chceme určit střed vstupní věty, délku vstupu nebo počet výskytů symbolu. Pak zavedeme do SA systému nový analyzátor g_s a mezi ním a prvním analyzátorem kaskády $g_k \circ \ldots \circ g_1$ zpracovávající jazyk zavedeme omezení (g_s, g_1, s), případně $\neg s$. Tento nový analyzátor nebude v relaci $\circ$ s g_1 (prvky jsou nesrovnatelné, každý v jiné větvi), ale kvůli omezení bude při interpretaci systému g_s vyhodnoceno dříve než g_1. Je zřejmé, že g_1 a g_s pracují oba se stejnou vstupní větou.
- řešení pomocí aplikace funkce – tento přístup je univerzálnější, zpracovanou část vstupu lze transformovat a použít k další syntaktické analýze. Tato transformace může být pomocí libovolné Turingovsky vyčíslitelné funkce. Opět zavedeme nový analyzátor g_s, který není nebude v relaci $\circ$ s žádným jiným analyzátorem kaskády $g_k \circ \ldots \circ g_1$. Poté zavedeme omezení $(g_s, g_1, f(g))$ a definujeme g. Funkce g by měla ze zpracovaného řetězce odstranit všechny symboly, které nenesou žádanou informaci. Poté upravíme model analyzátoru g_1 taky, aby uměl zpracovat tento nový řetězec generovaný funkcí g.

5.4 Interpretace SA systémů

Doposud jsme se věnovali tomu, jak správně SA systémy navrhnout a sestrojit. Nyní se zaměříme na interpretaci SA systémů, tedy na proces samotné syntaktické analýzy s využitím těchto systémů. Pro správnou interpretaci systému je potřeba objasnit:

1. pořadí a typy vyhodnocování omezení,

2. interpretaci relace $\circ$.
   (a) interpretace toku vstupních dat pomocí reentrantního lexikálního analyzátoru,
   (b) pořadí vyhodnocování analyzátorů v kaskádě,
   (c) pořadí vyhodnocování samostatných kaskád v rozkladu (nezávislých a závislých),
3. vyhodnocování komponent.

5.4.1 Vyhodnocení omezení

Vyhodnocení omezení budeme provádět v rámci činnosti každé komponenty zvlášť, tj. toto vyhodnocení implementujeme do řídícího algoritmu každé komponenty. Omezení můžeme vyhodnotit před činností komponenty, při její činnosti nebo po až skončení činnosti komponenty. Vhodný okamžik vyhodnocení se liší pro použitá omezení. Právě podle okamžiku vyhodnocení rozdělíme omezení do 3 skupin – rozlišujeme tzv. *typ omezení*:

- *automatické vyhodnocení* – omezení není nijak kontrolováno ze strany syntaktického analyzátoru – ani při jeho činnosti, ani po skončení analýzy. Omezení jsou zadána před spuštěním syntaktického analyzátoru a jejich vyhodnocení je implicitní (zajišťuje jej algoritmus syntaktické analýzy). Toto vyhodnocení je vhodné pro replikaci (r, $\succ r$), konverzi ($c(\phi)$, $\succ c(\phi)$) a aplikaci funkce ($f(g)$, $\succ f(g)$).

- *průběžné vyhodnocení* – omezení jsou kontrolována na počátku každé iterace hlavního řídícího cyklu syntaktického analyzátoru. Typicky se toto vyhodnocení provede před výběrem derivačního pravidla. Pokud je některé omezení vyhodnoceno jako nesplněné, syntaktická analýza skončí odmítnutím, přestože by samotný analyzátor ještě mohl pokračovat ve své činnosti. Toto vyhodnocení je vhodné pro pozitivní synchronizaci (s) a pozitivní sledování (t). U synchronizace lze průběžně zkoumat, zdali je počet provedených kroků menší či roven zadanému limitu. Podobně u sledování lze zkoumat, zdali je sekvence provedených kroků podsekvencí zadaného omezení. Tento přístup má výhodu v tom, že lze rychleji odhalit případnou chybu (není nutné analyzovat celý vstup). Zároveň poskytuje větší míru flexibility – uvažujme následující scénář: provedli jsme $k-m$ kroků (za m dalších derivačních kroků dosáhneme limitu k), analyzátor g může pokračovat (nezpracoval celý vstup), další vstupní lexéma je v $Next(g)$, na zásobníku máme m neterminálů, přičemž pro každý z nich existuje ε-pravidlo. V takovém případě můžeme neterminály uměle odstranit, dostáváme větu a analýzu prováděnou komponentou g ukončujeme jako úspěšnou (udělali jsme k kroků, i když jsme zpracovali pouze část vstupu). Zbytek vstupu ponecháme dalším analyzátorům v relaci $\circ$. Nevýhodou tohoto přístupu je složitá dopředná analýza udávající, zdali můžeme zbývající neterminály odstranit v m krocích. Zároveň je nutný zásah do řídícího algoritmu analyzátoru při implementaci. Výhodou tohoto přístupu je zvýšení síly tak, že pomocí kaskády analyzátorů půjde zpracovat i některé nedeterministické kontextové jazyky.

- *dodatečné vyhodnocení* – omezení jsou kontrolována až po dokončení syntaktické analýzy a to pouze v případě přijetí. Jestliže analyzátor přijal a není splněno některé omezení, pak je výsledek syntaktické analýzy negativní, tj. řetězec je zamítnut. Toto vyhodnocení je vhodné pro synchronizaci (s, $\neg s$) a sledování (t, $\neg t$). Výhodou je jednodušší implementace (nemění se řídící algoritmus syntaktického analyzátoru).

5.4.2 Reentrantní lexikální analyzátor

Obsahuje-li relace $\circ$ pouze jednu kaskádu analyzátorů, pak nemusíme do algoritmu lexikálního analyzátoru vůbec zasahovat, ale v případě, že je mocnost relace $\circ$ (počet řetězců v relaci), budeme muset zajistit vícenásobný přístup ke vstupu. Mocnost $\circ$ nám udává maximální počet přístupů k jedné lexémě zpracované lexikálním analyzátorem.

Začneme u rozkladu relace $\circ$ – jednotlivé množiny (řetězce komponent, dále kaskády) rozkladu χ si očíslujeme $(1, \ldots, |\chi|)$. Poté každému analyzátoru $g \in \mathcal{P}$ přiřadíme seznam kaskád $\chi(g) = [\mathbb{N}]$, ve kterých se g vyskytuje. Lexikální analyzátor bude mít přístup k rozkladu χ i k seznamu kaskád každé komponenty systému.

Příklad 5.4.1. Rozklad $\chi = \{\{g_1, g_2, g_3, g_5\}, \{g_1, g_2, g_4, g_5\}, \{g_6\}\}$ si očíslujeme, pak $[1] = \{g_1, g_2, g_3, g_5\}$, $[2] = \{g_1, g_2, g_4, g_5\}$, $[3] = \{g_6\}$. Poté přiřadíme každému analyzátoru seznam kaskád:

$$\chi(g_1) = [1, 2], \chi(g_2) = [1, 2], \chi(g_3) = [1], \chi(g_4) = [2], \chi(g_5) = [1, 2], \chi(g_6) = [3].$$

Rozpoznávání lexém reentrantním lexikálním analyzátorem funguje zcela běžným způsobem, ale po rozpoznání je lexéma očíslována a vložena do vyrovnávací paměti. Pro každou kaskádu $1, \ldots, |\chi|$ si pamatujeme pozici ve vyrovnávací paměti. Při inicializaci jsou všechny tyto pozice nastaveny na začátek vyrovnávací paměti, která je prázdná. Lexikální analyzátor poté pracuje následujícím způsobem:

1. požaduje-li některá komponenta g další lexému (volá funkci `getNextToken` s označením komponenty g), pak lexikální analyzátor vybere z $\chi(g)$ takový rozklad, jehož pozice m je nejmenší.
   - Je-li lexéma ve vyrovnávací paměti, tj. pozice m je menší než pořadové číslo poslední lexémy, analyzátor vrací lexému na pozici $m + 1$ a inkrementuje m.
   - Není-li lexéma ve vyrovnávací paměti (m větší či rovno pořadovému číslu poslední lexémy), pak analyzátor provede lexikální analýzu vstupu (načte další lexému ze vstupu), tuto lexému očísluje číslem $m + 1$ a uloží ji do vyrovnávací paměti.
2. Jsou-li pozice všech kaskád větší než pořadové číslo lexémy na počátku vyrovnávací paměti, pak tuto lexému z vyrovnávací paměti odstraníme.
3. Ukončí-li komponenta g svoji činnost (příjme či odmítne), oznámí to lexikálnímu analyzátoru pomocí funkce `endReached`. Lexikální analyzátor poté vyřadí tuto komponentu ze všech kaskád v rozkladu χ. Je-li některá kaskáda prázdná, pak nastaví její pozici m na inf (v praxi na největší možné číslo) a zkontroluje, jestli nelze některé zpracované lexémy podle bodu (2) uvolnit z vyrovnávací paměti.

Z principu činnosti plyne, že vyrovnávací paměť je vhodné realizovat pomocí dynamického seznamu. Takto upravený lexikální analyzátor poté umožňuje vyhodnocování libovolně větvených relací $\circ$, kde některé komponenty pracují paralelně (se stejným vstupem).

5.4.3 Interpretace relací $\circ$ a $\mathcal{R}^{\succeq}$

Relaci $\circ$ jsme zavedli právě pro určení pořadí vyhodnocování pořadí komponent, ovšem toto pořadí nelze správně určit bez relace $\mathcal{R}^{\succeq}$. Než se ovšem dostaneme k interpretaci těchto relací, je nutné si zavést terminologii.

Definice 5.4.2. Nechť $g \in \mathcal{P}$ je komponenta SA systému σ. Nad komponentou g zavedeme predikáty udávající její stav:

- g není vyhodnocena $\iff \odot g$ (není jasné nebo nás nezajímá, jestli komponentu lze vyhodnotit),
- g není vyhodnocena a lze ji vyhodnotit $\iff \oplus g$,
- g není vyhodnocena a nelze ji vyhodnotit $\iff \ominus g$,
- g je vyhodnocena $\iff g\odot$ (není jasné nebo nás nezajímá, jestli komponenta přijala či omítla),
- g je vyhodnocena a přijala $\iff g\oplus$,
- g je vyhodnocena a odmítla $\iff g\ominus$.

Dále $\circ\gg \subseteq \mathcal{P}$, $g \in \circ\gg \iff g\odot$, je množina všech vyhodnocených komponent a $\gg\circ \subseteq \mathcal{P}$, $g \in \gg\circ \iff \oplus g$ je množina všech vyhodnotitelných komponent.

Je zcela zřejmé, že komponentu g lze vyhodnotit, právě když byly vyhodnoceny všechny komponenty v relaci $\circ$ před g. Formálně:

$$(g_i, g_{j_1}) \in \circ, \ldots, (g_i, g_{j_m}) \in \circ \wedge \forall i \in \{1, \ldots, m\} : g_{j_i} \odot \wedge \odot g_i \Longrightarrow \oplus g_i. \tag{5.4}$$

Zde už je naprosto zřejmé, proč jsme požadovali acykličnost relace $\circ$. Pro zjednodušení výkladu budeme nyní na chvíli předpokládat, že predikáty $\odot$ vychází pouze z relace $\circ$. Z implikace na první pohled plyne, že lze vždy vyhodnotit nejmenší prvky relace, tedy množina $\gg\circ$ bude ze začátku obsahovat všechny prvky z $inf(\circ)$. Pokud totiž nepředchází žádná komponenta g, pak se implikace 5.4 zjednodušuje na $\odot g \Longrightarrow g\odot$. $inf(\circ)$ je vždy nejmenším řezem (5.1.6) relace $\circ$ a dále platí, že na pořadí vyhodnocování komponent v řezu nezáleží. Komponenty v řezu totiž budou pracovat kvaziparalelně a korektní přístup ke vstupu zajistí reentrantní lexikální analyzátor. Triviální implementace by byla složena pouze ze čtyř kroků:

1. $\gg\circ \; = inf(\circ)$, $\circ\gg = \emptyset$,
2. vyber libovolný prvek g z $\gg\circ$, odstraň g z $\gg\circ$, vlož g do $\circ\gg$ a vyhodnoť g,
3. projdi množinu $\mathcal{P}$ a vlož do $\gg\circ$ všechny takové komponenty $g \in \mathcal{P}$, pro které je splněna implikace 5.4 a nejsou přítomny v $\circ\gg$,
4. pokud $\gg\circ \neq \emptyset$, jdi na bod (2).

Nyní je zajištěno, že komponenty v relaci $\circ$ na sebe budou navazovat. Výběr libovolného prvku lze ještě optimalizovat – je vhodné vybírat spíše prvky z jednoho řezu, než z jednoho řetězce komponent. Tento výběr poté vykazuje menší paměťové nároky (lexikální analyzátor nemusí pak držet mnoho lexém ve vyrovnávací paměti). Tento výběr je podstatný především pokud máme v systému dvě nezávislé (5.1.5) kaskády komponent, což si ukážeme na příkladu 5.4.3.

Příklad 5.4.3. Uvažujme SA systém s 6 komponentami, propojenými následovně: $g_3 \circ^s g_2 \circ^s g_1$, $g_6 \circ^s g_5 \circ^s g_4$. Komponenty používají pouze pravidla tvaru $A \to aB \mid a$, kde $A, B \in N$, $a \in T$. Vstup má délku n znaků, pak je zřejmé, že na jeho zpracování bude dohromady potřeba n derivačních kroků. Kvůli synchronizaci provede každá komponenta $\frac{n}{3}$ kroků. Pak platí:

- při výběru prvků po kaskádách, tj. vyhodnotíme-li prvky v pořadí $g_1, g_2, g_3, g_4, g_5, g_6$, budeme muset držet v paměti n lexém.
- při výběru prvků po řezech, tj. vyhodnotíme-li prvky v pořadí $g_1, g_4, g_2, g_5, g_3, g_6$, budeme muset držet v paměti $\frac{n}{3}$ lexém.

Výběr komponent se stane složitějším, pokud do něj vstoupí i relace $\mathcal{R}^{\succeq}$ – kaskády nezávislé v relaci $\circ$ mohou být závislé v $\mathcal{R}^{\succeq}$ a řezy těchto relací mohou být úplně odlišné. Implikace 5.4 se nám zkomplikuje na:

$$\begin{aligned}&(g_i, g_{j_1}) \in \circ, \ldots, (g_i, g_{j_m}) \in \circ \wedge \forall i \in \{1, \ldots, m\} : g_{j_i} \odot \wedge \odot g_i \\ &\wedge \forall g_q \in \{g_p \in \mathcal{P} \mid (g_p, g_i, \eta_1) \in \mathcal{R}, \eta_1 \in \eta\} : g_q \odot \Longrightarrow \oplus g_i.\end{aligned} \tag{5.5}$$

Důvod je zcela prostý, i když jsou g_i, g_q ve stejném řezu, tedy na pořadí vyhodnocení podle $\circ$ nezáleží, stále mohou existovat nějaká omezení mezi g_i a g_q. My musíme zajistit, aby byla později vyhodnocena komponenta, na jejíž činnost má toto omezení vliv (cílová komponenta). Jinak řečeno, nejprve je potřeba určit hodnotu omezení (vyčíslit omezení), což znamená vyhodnotit nejprve zdrojovou komponentu. Nezbude nám nic jiného, než prověřit i všechny prvky v relaci $\mathcal{R}^{\succeq}$ ve kterých figuruje g_i, abychom zjistili, jestli je možné g_i vyhodnotit. Hledání vhodné komponenty ve dvojici relací skýtá ještě jeden problém: přestože jsou $\circ$ i $\mathcal{R}^{\succeq}$ acyklické, $\circ \cup \mathcal{R}^{\succeq}$ může obsahovat cyklus. Takový cyklus lze detekovat, pokud si sestavíme graf toku. Pro každý uzel $g \in \mathcal{P}$ musí platit $\neg(g \twoheadrightarrow^* g)$, kde znak $\twoheadrightarrow$ reprezentuje přechod po hraně $\Rightarrow$ (relace $\circ$) nebo po hraně $\to$ (relace $\mathcal{R}^{\succeq}$) – je-li tato podmínka porušena, pak existuje cyklus. S existencí cyklu si ovšem dokážeme poradit, aniž bychom museli provádět analýzu nad grafem toku, celkem jednoduše lze zakomponovat detekci cyklu do algoritmu interpretace SA systému 5.4.4.

Algoritmus 5.4.4: Vyhodnocení SA systému

Vstup: SA systém $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$,

$\gg\circ := inf(\circ)$,
$\circ\gg := \emptyset$,
while $\gg\circ \neq \emptyset$ **do**
  $g_{selected} := \emptyset$ ▷ $\emptyset$ reprezentuje non-komponentu
  for $g_e \in \gg\circ$ **do** ▷ výběr komponenty
    $g_{preset} := \{g_j \in \mathcal{P} \mid (g_e, g_j) \in \mathcal{R}^{\succeq}\}$ ▷ vybereme $\forall$komp., kt. omezují g_e
    $\oplus g_e := \forall g_j \in g_{preset} : g_j \in \circ\gg$ ▷ $\forall$ tyto komp. musí být vyhodn.
    if $\oplus g_e$ **then**
      $g_{selected} := g_e$ ▷ vhodná komp. nalezena
      break
    end if
  end for
  if $g_{selected} \neq \emptyset$ **then**
    $\gg\circ := \gg\circ \setminus \{g_{selected}\}$,
    $\circ\gg := \circ\gg \cup \{g_{selected}\}$,
    $g_{selected}$.evaluate() ▷ vyhodnotíme vybranou komp.
  else
    throw „Nelze vybrat komp., existuje cyklus“
  end if
end while

Tento algoritmus pouze formálně vystihuje vše, co již bylo řečeno výše. Iterujeme přes prvky v relaci $\circ$ a vždy k vyhodnocení vybereme takový, pro který jsou již všechna omezení vyčíslena. Nepodaří-li se vybrat žádnou komponentu k vyhodnocení, pak SA systém obsahuje cyklus. Popis klíčové procedury *evaluate* necháme do další kapitoly věnující se interpretaci jednotlivých komponent.

Pro jazyk $L(\sigma \mid g_1, \ldots, g_k)$ můžeme o přijetí či odmítnutí řetězce rozhodnout, jakmile jsou vyhodnoceny všechny komponenty $g_1, \ldots, g_k$ (jistá podmnožina $sup(\circ)$). V tomto bodě lze říct, že SA systém přijal větu w, právě když podle definice 5.1.8 platí:

$$g_1 \oplus \wedge \ldots \wedge g_k \oplus \text{ pro } k \geq 0.$$

Zde se nabízí jistá optimalizace, a to sice odmítnout ve chvíli, kdy některá z komponent $g_1, \ldots, g_k$ odmítla. V další kapitole si ovšem představíme jednoduší a téměř stejně efektivní přístup k optimalizaci.

5.4.4 Vyhodnocení komponent

Nyní se podrobně podíváme na proceduru `evaluate`, tedy na vyhodnocení jednotlivých komponent SA systému.

Pozorný čtenář si zajisté povšiml, že algoritmus 5.4.4 vyhodnotí každou komponentu SA systému (celý SA systém) bez ohledu na to, jestli je to nutné. Z definice jazyka přijímaného kaskádou plyne, že pokud odmítne první komponenta kaskády, není nutné vyhodnocovat další komponenty kaskády, neboť věta bude každopádně odmítnuta. Tomuto přístupu se říká zkrácené vyhodnocování a intuitivně se jedná o správný přístup. Použitelný je ovšem pouze ve chvíli, kdy máme nezávislé kaskády. Ve chvíli, kdy je relace $\circ$ větvená, tento

přístup použít nelze. Bylo by potřeba provést analýzu nad grafem toku a algoritmus vyhodnocení 5.4.4 by se výrazně zkomplikoval. Triviální řešení je vyhodnotit všechny komponenty v hlavní smyčce programu a ponechat řešení zkráceného vyhodnocování do procedury `evaluate`. Stačí na začátku této této procedury implementovat následující podmínku:

$$(\exists g_j \in \{g_j \in \mathcal{P} \mid (g_i, g_j) \in \circ\} : g_j\ominus) \wedge \oplus g_i \implies g_i\ominus \tag{5.6}$$

Tato rovnice udává, že pokud některá komponenta g_j přímo předcházející aktuální vyhodnocovanou komponentu g_i odmítla, pak i současná komponenta g_i může odmítnout, tj. nemusí vůbec provádět syntaktickou analýzu. Tento přístup budeme nazývat *vyhodnocení v případě potřeby*.

Je čas shrnout výše řečené, vyhodnocení komponenty se skládá z následujících činností:

1. aplikace rovnice 5.6 (vyhodnocení v případě potřeby),
2. nastavení omezení a jejich validace,
3. vyhodnocení komponenty a aplikace průběžných omezení,
4. aplikace dodatečných omezení a korekce výsledku.

U každé vyhodnocené komponenty g_i si budeme pro práci s omezeními pamatovat:

- $g_i\oplus$, $g_i\ominus$ – stav komponenty po vyhodnocení (komponenta přijala či odmítla),
- w_{parsed} – přijatou část vstupní věty,
- n_{steps} – počet derivačních kroků provedených komponentou,
- p_{rules} – seznam pravidel aplikovaných komponentou g_i.

Pokud komponenta nepřijala, pak nejsou hodnoty definovány, je ovšem rozumné použít výchozí hodnoty $w_{parsed} = \varepsilon, n_{steps} = 0, p_{rules} = []$.

Máme-li validní omezení, tj. je-li zdrojová komponenta vyhodnocená, a tedy známe hodnotu omezení, můžeme vyhodnotit aktuální (cílovou) komponentu. Může ovšem nastat i situace (nikoliv ojediněle), kdy zdrojová komponenta poskytující omezení odmítla a hodnota tohoto omezení není vůbec stanovena. Pak nezbývá, než takové omezení označit na nevalidní a vyhodnocení aktuální komponenty řešit stejně jako v případě odmítnutí některé předcházející komponenty v kaskádě. Aktuální komponentu nelze správně vyhodnotit, protože nám chybí dostatek informací (hodnota omezení), a proto ani nebudeme provádět syntaktickou analýzu a rovnou řekneme, že komponenta odmítla.

Po validaci aplikujeme automatické omezení, které může být z definice nejvýše jedno. Na zásobník syntaktického analyzátoru g vložíme reverzovanou část vstupu přijatou zdrojovou komponentou (reverzace je nutná kvůli korektnímu porovnávání symbolů se vstupem). Poté necháme pracovat některý klasický algoritmus syntaktické analýzy. Po jeho skončení provedeme korekci výsledků pomocí dodatečných omezení, tedy provedeme prosté porovnání, zdali se výsledek syntaktické analýzy shoduje s očekávanou hodnotou danou omezením (od zdrojové komponenty). Vyhodnocení komponenty shrnuje algoritmus 5.4.5.

Algoritmus 5.4.5: Procedura `evaluate`, vyhodnocení komponenty g_i

Vstup: SA systém $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$, komponenta g_i k vyhodnocení,

1: $Prev :=$ **select all** g_j **where** $(g_i, g_j) \in \circ$
2: $Prev := Prev \cup$ (**select all** g_j **where** $(g_j, g_i, \eta_1) \in \mathcal{R}$)
3: **foreach** $g_j \in Prev$ **do** ▷ vyhodnocení v případě potřeby
4: **if** $g_j \ominus$ **then**
5: změn stav g_i na $g_i \ominus$ ▷ komponentu nelze vyhodn. g_i odmítá
6: **return**
7: **end if**
8: **end foreach**
▷ příprava pro automatické vyhodnocení omezení
9: **if** $(g_{scr}, g_i, \eta_1) \in \mathcal{R}$ **where** $\eta_1 \in \{r, \succ r, c(\phi), \succ c(\phi), f(g), \succ f(g)\}$ **then**
10: **if** $\eta_1 \in \{c(\phi), \succ c(\phi)\}$ **then**
11: $preset :=$ convert(ϕ, $g_{src}.w_{parsed}$) ▷ vypočteme konvertovaný řetezec
12: **else if** $\eta_1 \in \{f(g), \succ f(g)\}$ **then**
13: $preset :=$ g($g_{src}.w_{parsed}$) ▷ vypočteme funkci g
14: **else**
15: $preset := g_{src}.w_{parsed}$
16: **end if**
17: **if** $\eta_1 \in \{\succ r, \succ c(\phi), \succ f(g)\}$ **then**
18: $g_i.stack$.push(S_i) ▷ vložíme axiom
19: **end if**
20: g_i.push($preset$.reverse()) ▷ vložíme obracený řetezec na zás. g_i
21: **end if**

22: g_i.parse() ▷ vyhodn. komp. a aplikace dodat. omezení
23: $steps^+ :=$ **select all** $g_j.n_{steps}$ **where** $(g_j, g_i, s) \in \mathcal{R}$
24: $steps^- :=$ **select all** $g_j.n_{steps}$ **where** $(g_j, g_i, \neg s) \in \mathcal{R}$
25: $rules^+ :=$ **select all** $g_j.p_{rules}$ **where** $(g_j, g_i, t) \in \mathcal{R}$
26: $rules^- :=$ **select all** $g_j.p_{rules}$ **where** $(g_j, g_i, \neg t) \in \mathcal{R}$
27: **if** $(steps^+ \neq \emptyset \wedge g_i.n_{steps} \notin steps^+) \vee (steps^- \neq \emptyset \wedge g_i.n_{steps} \in steps^-) \vee (rules^+ \neq \emptyset \wedge g_i.p_{rules} \notin rules^+) \vee (rules^- \neq \emptyset \wedge g_i.p_{rules} \in rules^-)$ **then**
28: změn stav g_i na $g_i \ominus$ ▷ komp. g_i dodatěčně odmítá
29: **else**
30: změn stav g_i na $g_i \oplus$ ▷ komp. g_i přijala
31: **end if**

Průběžná omezení je nutné zakomponovat přímo do algoritmu syntaktické analýzy – nastíníme řešení pouze pro synchronizaci. Na začátku hlavního řídícího cyklu, tedy před výběrem derivačního pravidla, je nutné provést analýzu předčasného ukončení. Víme-li, že analyzátor provedl n_{steps} kroků a má provést právě k kroků, pak můžeme:

- předběžně odmítnout pokud $n_{steps} > k$,
- předběžně přijmout, pokud lze použitím $k - n_{steps}$ derivačních pravidel odstranit ze zásobníku všechny neterminály a obsah zásobníku se shoduje se vstupem.

Analýza předběžného přijmutí je ovšem poměrně náročná, proto je vhodné se zaměřit pouze na odstranění neterminálů pomocí ε-pravidla a provést poté kontrolu synchronizace v rámci dodatečného vyhodnocení omezení.

SA systémy nemusí být nutně jen interpretované, samozřejmě existuje způsob, jak SA systém přeložit na syntaktický analyzátor v binární podobě. V takovém případě potřebujeme předem určit pořadí vyhodnocování komponent. Algoritmus 5.4.4 lze pro tuto činnost přizpůsobit, stačí pouze:

1. na začátku zavést prázdný seznam $\sqsubseteq$,
2. místo vyhodnocení komponenty na řádku 16 vložit komponentu $g_{selected}$ do $\sqsubseteq$.

Hlavní řídící cyklus analyzátoru bude poté pouze iterovat přes seznam $\sqsubseteq$ a volat jednotlivé komponenty.

5.5 Generativní síla syntaktické analýzy založené na SA systémech

Na závěr kapitoly věnované SA gramatickým systémům je vhodné rozvinout téma generativní kapacity těchto struktur a podívat se, co jsme zavedením SA systémů získali, a kde je naopak využít nelze. V následujících formulích budeme zkratkou $L(LL)$ označovat rodiny jazyků generovaných LL gramatikou, $L(LR)$ rodiny jazyků generovaných přijímaných některým LR syntaktickým analyzátorem. $L(SA^X)$ bude označovat rodiny jazyků přijímaných SA systémy s využitím pouze komponent typu $X \in \{LL, LR\}$ či pomocí pravidel typu $X \in \{REG, CF, CS\}$. Neklademe-li omezení na použité komponenty ani formát použitých pravidel, pak píšeme $L(SA)$. Ve všech těchto případech připouštíme ε-pravidla, ale nepřipouštíme omezení v podobně aplikace funkce. Možnost využití tohoto omezení indikujeme pomocí f zapsaného dolním indexem, tedy $L(SA_f^X)$ značí rodiny jazyků přijímané s využitím aplikace funkce.Následující relace platí pro rodiny jazyků generovaných SA systémy:

1. $L(LL) \subset L(SA^{LL}) \subseteq L(SA)$,
2. $L(LR) \subset L(SA^{LR}) \subseteq L(SA)$,
3. $DCS \subseteq L(SA^{CS}) = L(SA)$,
4. $CF \subset CS = L(SA_f^{CF}) \subset REC$.
5. $L(SA^{LR}) = L(SA^{CF})$,
6. $CF \not\subset L(SA^{CF})$,
7. $DCF \subset L(SA^{CF}) \subset L(SA^{CS})$,
8. $(DCS \setminus DCF) \cap L(SA^{CF}) \neq \emptyset$,

Začneme konstatováním inkluze (1) – SA obsahující LL komponentu nutně musí zahrnovat všechny jazyky z $L(LL)$. Důkaz tohoto tvrzení je triviální, postačí nám zavést SA systém s jedinou komponentou, tedy $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$, kde $N = N_1, T = T_1, \mathcal{G} = \{G_1\}$, $\mathcal{P} = \{G_1^{LL}\}$, $\circ = \emptyset$ a $\mathcal{R} = \emptyset$. Snadno lze nahlédnout, že podle definice 5.1.8 platí vztah:

$$L(\sigma) = L(\sigma \mid sup(\circ)) = L(\sigma \mid G_1^{LL}) = \{w \in T^* \mid w \in L(G_1^{LL})\} = \{w \in T^* \mid G_1 \Rightarrow^* w\} = L(G_1).$$

Nyní už jen stačí najít jazyk, který není LL, ale je v $L(SA^{LL})$ – takových jazyků jsme si již ukázali mnoho, například $L = \{a^n b^n c^n \mid n \geq 1\}$. Tedy opravdu platí $L(LL) \subset L(SA^{LL})$. Obdobný důkaz (2) lze provést i pro rodinu jazyků $L(SA^{LR})$. Stejně tak při použití kontextové komponenty (lineárně ohraničeného automatu) se lze jednoduše dostat na sílu jazyků DCS (3). Je zřejmé (3), že pokud nejsilnější použitelná komponenta je LOA (síla CS), pak musí platit $L(SA) = L(SA^{CS})$.

Není vůbec překvapivé, že pokud povolíme aplikaci funkce (4), která má sama o sobě Turingovskou sílu, pak můžeme přijímat všechny kontextové jazyky. Funkce g může být libovolně složitá, jedinou podmínkou je, že ji přijme LOA, tedy musí být vyčíslitelná se stejnými nebo menšími paměťovými nároky, než je délka vstupní věty. Zároveň uvažujeme pouze takové funkce g, které jsou vyčíslitelné v konečném čase. Podíváme-li se na počítač jako na LOA (máme k dispozici omezenou paměť), pak můžeme přijetí jazyka $L \in CS$ redukovat na vyčíslení funkce g, a tedy můžeme přijímat libovolný jazyk z CS.

Celkem jednoduše lze i ukázat vlastnost $L(SA_f^{CF}) \subset REC$ – ukážeme, že existuje rekurzivní jazyk, který nelze přijímat ani pomocí SA systému s využitím aplikace funkce. Nechť $G_1, G_2, \ldots$ je výčet všech počitatelných kontextových gramatik. Dále $x_1, x_2, \ldots$ je výčet všech řetězců z abecedy Σ^*. Zvažme $\Sigma = \{0, 1\}$, poté platí $x_i \in \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Nyní uvažujme jazyk $L = \{x_i \mid x_i \in L(G_i)\}$ prezentovaný v [2] pro který platí:

1. L není kontextový jazyk. Pokud by byl, musela by existovat gramatika G_j, která tento jazyk generuje, ale zároveň musí platit $x_j \in L \wedge x_j \notin L(G_j)$.

2. L je rozhodnutelný – pro dané x_i pouze zkontrolujeme, zda-li G_i generuje tento řetězec (problém z třídy $PSPACE$).

3. Nelze sestrojit SA systém, který tento jazyk rozhoduje – už jen z faktu, že máme spočetně mnoho gramatik $\{G_1, G_2, \ldots\}$. Množina $\mathcal{G}$ musí být podle definice 5.1.1 konečná, pak zřejmě $\mathcal{G} \subset \{G_1, G_2, \ldots\}$, a tedy $\exists G_i \in \{G_1, G_2, \ldots\} : G_i \notin \mathcal{G}$. Zřejmě nemůže SA systém tento problém rozhodnout a nepřijímá L.

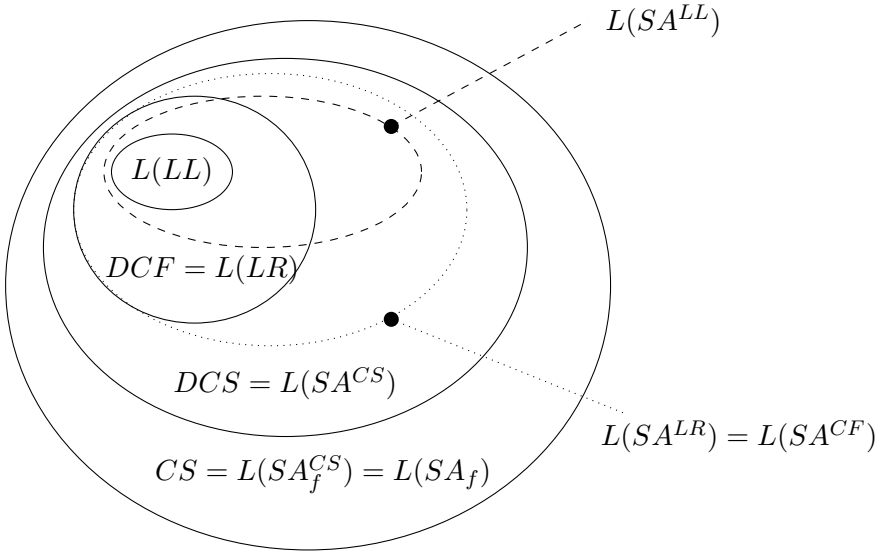
Zajímavější bude zkoumat vlastnosti SA systémů bez aplikace funkce. Překvapivé není ani tvrzení (5), tedy že lze pomocí SA systému s využitím LR komponent přijímat libovolný jazyk z DCF. Tato vlastnost plyne z ekvivalence rodin DCF a LR, která byla dokázána v [4]. Ovšem jednoduše lze ukázat, že existují jazyky z CF, například $L = \{ww^r \mid w \in \{a, b\}^*\}$, které nelze přijímat ani pomocí SA systému složeného z bezkontextových komponent (6), pokud nepoužijeme aplikaci funkce. S využitím aplikace funkce lze stanovit střed řetězce ww^r. Pokud je délka $|ww^r| = k$, zřejmě je $|w| = \frac{k}{2}$, ale k řešení nelze použít synchronizaci, neboť formální definice nám umožňuje pouze synchronizaci *právě* k kroků. Tento problém má triviální implentační řešení, zavést pro sychronizaci funkci nad k, ale z formální hlediska je to pouze jinak využitá aplikace funkce, tedy $L \in L(SA_f^{CF})$. Formální důkaz o nemožnosti přijetí L komponentou založenou na DZA je k nahlédnutí v [6]. Protože jednou bezkontextovou komponentu nelze L přijmout, zavedeme dvě komponenty (g_1 pro w, g_2 pro w^r). V takovém případě bude nutně platit $Last^{g_1}(S_1) = Next_S A^{g_1} \neq \emptyset$, a tedy jazyk nelze SA systémem zpracovat.

Pro potvrzení (7) ostré inkluze $DCF \subset L(SA^{CF})$ stačí pouze nalézt jazyk L, že $L \notin DCF \wedge L \in DCS$. Příkladů takových jazyků jsme si již v průběhu práce uvedli několik a v tomto bodě se pouze odkážeme na příklad 5.3.7, kde jsme sestrojili SA systém σ, pro který platí $L(\sigma) = \{w \otimes w \mid w \in \{a, b\}^*\}$. Zároveň jsme použili pouze bezkontextové komponenty, a proto platí $L(\sigma) \in L(SA^{CF})$.

Bod (8) je asi nejdůležitější – udává, že se podařilo naplnit cíle stanovené na počátku kapitoly o SA systémech. Jednoduše řečeno, podle (8) bodu existují deterministické kontextové jazyky, které můžeme zpracovat pouze pomocí bezkontextových pravidel. Samozřejmě v této oblasti existuje celá řada otevřených problémů, na ukázku můžeme zmínit:

1. $DCS =^{?} L(SA^{CF})$, lze přijímat jakýkoliv deterministický kontextový jazyk pomocí SA systému s bezkontextovými pravidly?
2. $CS \subset^{?} L(SA^{CS})$, existuje jazyk z CS, který nelze pomocí kontextových SA systémů bez aplikace funkce analyzovat?
3. $L(SA^{LL}) \subset^{?} L(SA^{LR})$, existuje jazyk z $L(LR)$, který nelze pomocí SA systémů s využitím LL komponent bez aplikace funkce analyzovat?

Výše uvedené relace jsou znázorněny na obrázku 5.5.1.

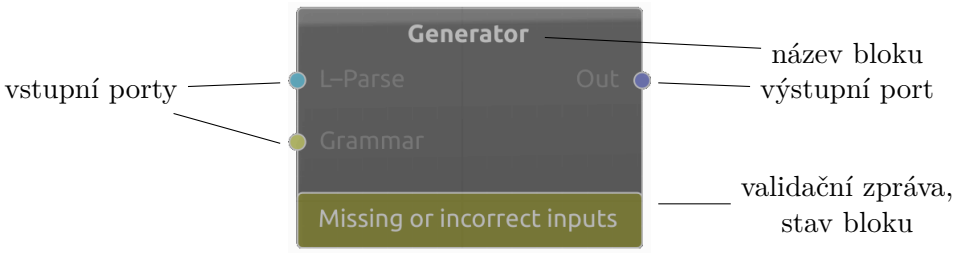


Obrázek 5.5.1 Hierarchie rodin jazyků $L(LL)$, $L(LR)$, DCF, DCS, CS, $L(SA^{LL})$, $L(SA^{LR})$, $L(SA^{CF})$, $L(SA^{CS})$, $L(SA_f^{CS})$ a $L(SA_f)$.

6 Implementace aplikace

Doposud jsme si z teoretického hlediska osvětlili SA systémy a stěžejní algoritmy pro práci s těmito systémy. Uvedené poznatky jsme využili k implementaci aplikace po návrh, modelování a analýzu SA systémů. Tato aplikace byla implementována v jazyce C++ s využitím grafického balíčku Qt. K vizualizaci byla použita open source knihovna `NodeEditor`, volně dostupná z https://github.com/paceholder/nodeeditor pod licencí BSD 3.

V rámci práce vznikl multiplatformní nástroj `PSEdit` (*Parser System Editor*), který poskytuje možnost modelovat SA systémy formou diagramu toku dat (*data flow diagram*). Model se skládá z funkčních bloků (obr. 6.0.1), které realizují specifickou funkci. Funkční bloky mají $0 - n$ vstupních datových portů a $0 - m$ výstupních datových portů. Skrze datové porty lze funkční bloky vzájemně propojovat. Vstupní porty mohou být propojeny nejvýše s jedním výstupním portem (pouze jeden zdroj dat), zatímco výstupní porty mohou být propojeny s libovolným počtem jiných portů (více spojů z jednoho výstupního portu). Při změně vstupních dat na vstupních portech se funkční blok vyhodnotí, nově vypočtené hodnoty nastaví na své výstupní porty, a poté se tato data propagují k dalším funkčním blokům prostřednictví spojů.



Obrázek 6.0.1 Ukázka funkčního bloku se dvěma vstupními a jedním výstupním portem.

6.1 Datové typy

Datové porty a spoje mezi funkčními jsou barevně rozlišeny podle typu přenášených dat – můžeme se tedy bavit o datových typech modelu. Spoje je možné vytvářet pouze mezi porty mající stejný datový typ. Pro reprezentaci přenášených dat jsou k dispozici následující datové typy:

- `Int+` (*Positive Integer, PosInt*, `#D7D069`) – nezáporné celé číslo odpovídající nativnímu datovému typu `unsigned int`.

- `[Int+]` (*PosInt List*, `#45D1FB`) – seznam nezáporných celých čísel, slouží například k reprezentaci použitých pravidel.
- `Text` (*Symbol*, `#69FB45`) – textový řetězec odpovídající nativnímu typu `std::string`, slouží k reprezentaci symbolu.
- `[Symbol]` (*Symbol List*, `#606AE0`) – seznam řetězců, slouží například k reprezentaci vstupní věty.
- `{Symbol}` (*Symbol Set*, `#44FBCD`) – množina řetězců (bez ohledu na pořadí), slouží k reprezentaci množin N či T.
- `[Rule]` (*Rule List*, `#D0DC64`) – seznam pravidel ve tvaru $[n]A \to x$, kde A je symbol, x je řetězec symbolů a n je unikátní číslo pravidla v rámci gramatiky.
- `Grammar` (*CFG*, `#E2EC54`) – gramatika, datová struktura ve tvaru N (`{Symbol}`), T (`{Symbol}`), P (`[Rule]`), S (`Symbol`).

Mezi některými datovými typy existují implicitní konverze. Porty, mezi jejichž datovými typy taková konverze existuje, mohou být také přímo propojeny. K dispozici jsou následující konverze:

- `Int+` $\longrightarrow$ `[Int]+` – z celého kladného čísla se vytvoří seznam obsahující právě toto číslo,
- `{Symbol}` $\longrightarrow$ `[Symbol]` – z množiny symbolů je vytvořen seznam, pořadí prvků seznamu není z principu množiny zaručeno,
- `Text` $\longrightarrow$ `{Symbol}` – z řetězce se vytvoří množina obsahující právě jeden prvek – zdrojový řetězec.

6.2 Funkční bloky

Funkční blok můžeme chápat jako funkci $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ – blok zapouzdře nějaký výpočet (matematickou funkci či složitější algoritmus) a z n vstupů vytvoří m výstupů. Program (model) je poté možné vytvářet kombinováním těchto funkcí. Některé funkční bloky mohou být navíc parametrizovány pomocí dialogu – ten lze vyvovat pomocí dvojtého kliknutí na blok.

Typická struktura bloku je vidět na obrázku 6.0.1. Každý blok má název, vstupní porty (vždy vlevo), výstupní porty (vždy vpravo), validační a stavovou oblast. Některé bloky navíc mohou mít další ovládací prvek umožňující nastavení bloku nebo zobrazující vypočtené údaje. Blok se může nacházet v jednom ze stavů:

- `Error` (chyba, `#AE0C0C`) – chyba při vyhodnocení mající takový charakter, že vyhodnocení nelze dokončit.
- `Warning` (varování, `#5D5D0C`) – chybně zadané vstupy či takový problém při vyhodnocování, který může vést k nevalidnímu výsledku. Vyhodnocení lze dokončit, ale správnost výsledku není zaručena.

- `Positive` (kladný výsledek, `#319031`) – blok byl správně vyhodnocen a výsledek je kladný (např. věta přijata).
- `Negative` (záporný výsledek, `#6E5454`) – blok byl správně vyhodnocen, ale výsledek je záporný (např. věta odmítnuta).
- `Note` (poznámka, `#5D5D5D`) – prostá indikace pozitivního či negativního faktu uživateli.
- `Valid` (normální, validní, `#373737`) – blok není ani v jednom z výše zmíněných stavů. Není podstatné, jestli je výsledek vyhodnocení kladný či záporný.

Stavy bloků mají přidělenou prioritu shodnou s pořadím, ve kterém byly uvedeny. Validační oblast má vždy barvu nejvíce prioritního stavu a zároveň zobrazuje zprávu, která odůvodňuje aktuální stav. Je-li blok ve stavu „normální“ (`Valid`), pak je validační oblast skrytá.

Funkční bloky lze rozdělit do několika kategorií podle jejich funkce. Konktrétní bloky si nyní představíme právě po kategoriích.

6.2.1 Zdrojové prvky (`Data Sources`)

Bloky z této katerie umožňují zadávaní dat. Nemají žádné vstupní porty, protože data se zadávají prostřednictvím vestavěných ovládacích prvků. Pro každý z výše zmíněných datových typů existuje nějaký blok z této kategorie, pomocí kterého může uživatel data zadat.

(a) `Text Source` (textový vstup, obr. 6.2.1a) – je blok s textovým vstupním polem. Data vložená do tohoto pole jsou dostupná na výstupním portu `Text`.

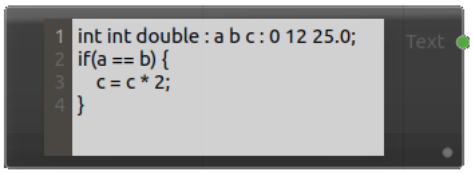
(b) `Program Source` (víceřádkový textový vstup, obr. 6.2.1b) – je blok s textovým polem, který umožňuje zadat víceřádkový vstup. Data zadaná do textového pole jsou dostupná na výstupním portu `Text`. Tento blok navíc poskytuje automatické číslování zdrojového textu.

(c) `SymbolSet Source` (vstupní blok pro množinu symbolů, obr. 6.2.1c) – blok s textovým polem sloužící k zadání množiny symbolů. Množina může být zadána buď ve formátu CSV, tedy jednotlivé symboly jsou odděleny čárkami, nebo ve formátu WS – jednotlivé symboly odděleny bílými znaky. Mezi formáty lze přepínat prostřednictvím radiových tlačítek ve spodní části bloku. Správně zadaná množina je poté k dispozici na výstupním portu `Set`. Při chybně zadaném vstupu (nedodržení formátu) je na výstupu prázdná množina.

(d) `SymbolList Source` (vstupní blok pro seznam symbolů, obr. 6.2.1d) – blok s textovým polem sloužící k zadání seznamu symbolů. Tento seznam může být zadán buď ve formátu CSV nebo ve formátu WS (symboly oddělené bílými znaky). Mezi formáty lze přepínat prostřednictvím radiových tlačítek ve spodní části bloku. Validní vstup je převeden na seznam, ten je poté k dispozici na výstupním portu `List`. Při chybně zadaném vstupu (nedodržení formátu) je na výstupu prázdný seznam.

(e) `Positive Integer Source` (kladný číselný vstup, obr. 6.2.1e) – blok s polem pro zadání kladného čísla. Zadané číslo je k dispozici na výstupním portu `Int+`.

(a) Zdrojový blok pro `Text`.



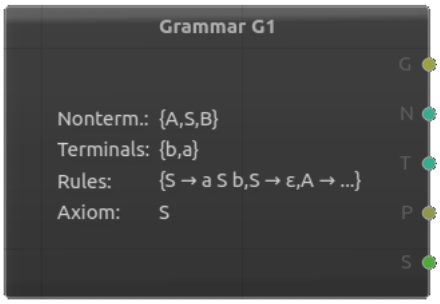
(b) Zdrojový blok pro program (`Text`).

(c) Zdroj. blok pro množinu symbolů (`{Symbol}`).

(d) Zdroj. blok pro seznam symbolů (`[Symbol]`).

(e) Zdrojový blok pro kladné číslo (`Int+`).

(f) Zdrojový blok pro seznam čísel (`[Int+]`).



(g) Zdrojový blok pro gramatiku (`Grammar`).

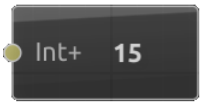
Obrázek 6.2.1 Zdrojové funkční bloky

(f) `PosInt List Source` (vstupní blok pro seznam kladných čísel, obr. 6.2.1f) – blok s textovým polem sloužící k zadání seznamu čísel. Seznam je zadáván v podobě textu v jednom z formátů CSV či WS (symboly oddělené bílými znaky). Mezi formáty lze přepínat prostřednictvím radiových tlačítek ve spodní části bloku. Validní vstup je převeden na seznam čísel a ten je poté k dispozici na výstupním portu `[Int+]`. Při chybně zadaném vstupu (nedodržení formátu, zadán řetězec nepřevoditelný na celé číslo) je na výstupu prázdný seznam.

(g) `Grammar Source` (vstupní blok pro gramatiku, obr. 6.2.1g) – blok sloužící pro specifikaci bezkontextové gramatiky ve tvaru (N, T, P, S). Gramatika je zadávána prostřednictvím dialogu, kde uživatel specifikuje:

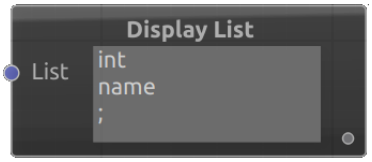
- množinu terminálů T ve formátu CSV,
- množinu neterminálů N ve formátu CSV,
- axiom S v textové podobě,
- množinu pravidel – jednotlivá pravidla jsou oddělena čárkami přičemž bílé znaky jsou ignorovány. Pravidlo má tvar `A -> x`, tedy levá strana obsahující právě jeden symbol je oddělena od pravé strany oddělovačem `->`. Pravá strana se sestává

(a) Zobrazovací blok pro `Text`.

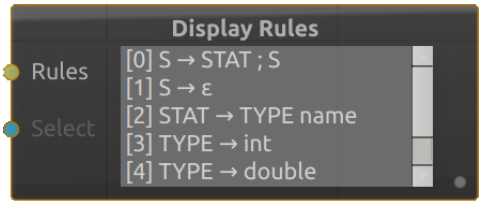


(b) Zobrazovací blok pro `Int+`.

(c) Zobrazovací blok pro `[Int+]`.



(d) Zobrazovací blok pro `[Symbol]` a `{Symbol}`.



(e) Zobrazovací blok pro (`[Rule]`).

Obrázek 6.2.2 Zobrazovací funkční bloky

z libovolného počtu symbolů z $N \cup T$. Pro popis ε-pravidel je vyhrazeno klíčové slovo `eps`. Například zápis `A -> eps` reprezentuje pravidlo $A \to \varepsilon$.

- volitelně název gramatiky (gramatika může být pojmenovaná).

Náhled zkrácené podoby gramatiky je poté uvnitř bloku. Blok má následující výstupní porty:

- `G` (`Grammar`) – gramatika jako čtveřice,
- `N` (`{Symbol}`) – množina neterminálů,
- `T` (`{Symbol}`) – množina terminálů,
- `P` (`[Rule]`) – množina pravidel,
- `S` (`Text`) – axiom.

6.2.2 Zobrazovací prvky (`Data Display`)

Pro ladění modelu a zobrazení výsledků mohou posloužit bloky z kategorie `Display`.

(a) `Text Display` (zobrazení textu, obr. 6.2.2a) – je blok sloužící k prosté indikaci textu, jenž je přítomný na vstupním portu `Text`.

(b) `Positive Integer Display` (zobrazení čísla N_0^+, obr. 6.2.2b) – je blok sloužící k prosté indikaci kladného celého čísla zadaného na vstupním portu `Int+`.

(c) `PosInt List Display` (zobrazení seznamu čísel, obr. 6.2.2c) – je blok zobrazující seznam kladných celých čísel, jenž je zadaný na vstupním portu `[Int+]`.

(d) `Symbol List Display` (zobrazení seznamu čísel, obr. 6.2.2d) – je blok zobrazující seznam, který je zadán na vstupním portu `List`. Tento blok lze využít i pro zobrazení

množiny symbolů (`{Symbol}`), neboť existuje implicitní konverze z množiny symbolů na seznam.

(e) `Rule List Display` (zobrazení seznamu pravidel, obr. 6.2.2e) – je blok zobrazující seznam pravidel. Tento blok má 2 vstupní porty – `Rules` (`[Rule]`) a `Selection` (`[Int+]`). Blok funguje v několika módech podle toho, které vstupy jsou využity:

- Je-li zadán pouze vstup `Selection` – blok zobrazí pouze pořadová čísla (id) použitých pravidel stejně, jako by to udělal blok `PosInt List Display`.
- Je-li zadán pouze vstup `Rules`, pak blok zobrazí všechna pravidla seznamu ve tvaru $[id]A \to x$.
- V případě využití obou vstupních portů (`Selection` i `Rules`) zobrazuje blok pravidla daná seznamem `Selection` ve tvaru $[id]A \to x$.

6.2.3 Distributory dat (`Data Repeaters`)

Pro každý datový typ existuje také blok z kategorie `Data Repeater`. Všechny bloky z této kategorie mají jeden vstup daného datového typu a jeden výstup stejného typu. Libovolná data přítomná na vstupním portu se nastaví na výstupní port. Tyto bloky slouží především pro zpřehlednění modelu. K dispozici jsou bloky `Text Repeater`, `Positive Integer Repeater`, `PosInt List Repeater`, , `Symbol List Repeater`, , `Symbol Set Repeater`, `Rule List Repeater` a `Grammar Repeater`.

Obrázek 6.2.3 Příklad bloku z kategorie `Repeaters` – `Rule List Repeater`.

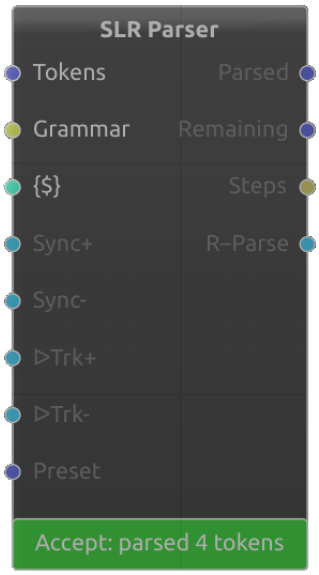
6.2.4 Komponenty (`Parsing`)

Nyní se konečně dostáváme k blokům tvořící jádro modelu SA systému – ke komponentám. Všechny komponenty z η mají v této kagorii svého zástupce, zároveň jsme sem zahrnuli lexikální analyzátor, u kterého zpracování vstupní věty začíná.

Blok `Tokenizer` reprezentuje lexikální analyzátor. Zároveň realizuje uzávěr $inf(\circ)$, neboť přímo připojené komponenty budou vždy vyhodnocovány přednostně (vyhodnocení SA systému začíná u lexikálního analyzátoru). Vstupní věta potažmo program je zadávána na portu `Text` (`Text`, povinný). Na portu `$` (`Text`, $) se poté zadává znak konce věty, který bude přiložen za vstupní větu. Samotný text je rozdělen na posloupnost lexém, které jsou k dispozici na výstupním portu `Tokens` (`[Symbol]`, []). Tokenizer může pracovat 3 různými způsoby:

1. `WS List` – vstup je rozdělen na lexémy pomocí bílých znaků.
2. `Char As Symbol` – vstup je rozdělen na lexémy po znacích (jeden znak odpovídá jedné lexémě). Tento mód byl zaveden pro jednodušší práci s teoretickými jazyky.

(a) Funkční blok `LL Parser` reprezentující komponentu typu LL.



(b) Funkční blok `SLR Parser` reprezentující komponentu typu LR.

Obrázek 6.2.4 Komponenty LL a LR.

3. `Regular Expr.` – vstup je analyzován pomocí regulárních výrazů. Výrazy lze zadávat pomocí dialogu – zadávají se dvojice $(token, regex)$, kde $token$ je název lexémy ve výstupním seznamu a $regex$ specifikuje konečný automat, který lexému přijímá. Pokud odpovídá prefix vstupní věty některému výrazu $regex$, je tento prefix odstraněn a do výstupního seznamu `Tokens` je přidána lexéma $token$. V opačném případě, tj. žádný prefix vstupní věty nebyl přijat žádným regulárním výrazem, lexikální analýza selhala (chybí pravidlo nebo vstup není validní) a na výstupu je prázdný seznam.

Nyní postupně projdeme komponenty z η – začneme u komponenty typu LL, kterou implementuje funkční blok `LLParser` (obr. 6.2.4a). Tento blok realizuje LL syntaktickou analýzu přesně podle zvyklostí. Ke své činnosti potřebuje externě zadanou LL gramatiku – blok si z této gramatiky sám vypočte množiny $First$, $Empty$ a $Follow_{SA}$. Poté sestaví LL tabulku a provede syntaktickou analýzu zadané vstupní věty. V případě přijetí je stav bloku změněn na `Positive`, v případě odmítnutí je stav nastaven na `Negative` a ve validační oblasti se zobrazí zpráva obsahující důvod zamítnutí věty. Blok má následující vstupní porty:

- `Tokens` (`[Symbol]`, povinný) – vstupní věta k syntaktické analýze.
- `Grammar` (`Grammar`, povinný) – LL gramatika použitá pro vytvoření LL tabulky.
- `{$}` (`{Symbol}`, volitelný / {$}) – množina znaků ukončující podsekvenci přijímanou komponentou. Tato množina odpovídá výše diskutované formální množině $Next$ a využije se k výpočtu $Follow_{SA}$. Není-li zadána, použije se množina obsahující znak $, který bývá typicky za koncem věty. Pro správný výsledek syntaktické analýzy celého SA systému musí uživatel správně specifikovat tuto množinu pro všechny analyzátory, a to na základě návrhu SA systému popsaného v sekci 5.2.
- `Sync+` (`[Int+]`, volitelný / [], dodatečné či průběžné vyhodnocení) – množina kroků využitých k pozitivní synchronizaci. Není-li port využit, pak může provést komponenta

libovolný počet kroků, v opačném případě musí komponenta přijmout v n krocích, kde n je některé číslo z množiny `Sync+`.

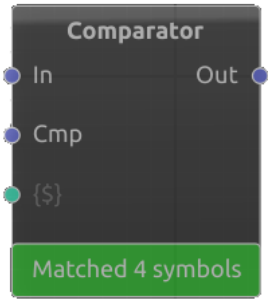
- `Sync-` (`[Int+]`, volitelný / [], dodatečné vyhodnocení) – množina kroků využitých k negativní synchronizaci. Není-li port využit, pak může provést komponenta libovolný počet kroků, v opačném případě musí komponenta přijmout v n krocích, přičemž n není žádné číslo z množiny `Sync-`.
- `▷Trk+` (`[Int+]`, volitelný / [], dodatečné vyhodnocení) – seznam pravidel pro pozitivní sledování. Komponenta musí přijmout pomocí pravidel ze seznamu `▷Trk+`, jinak odmítá. Samozřejmě zde záleží na pořadí aplikace pravidel. Na rozdíl od abstraktního algoritmu vyhodnocení komponenty (5.4.5) je zde povolena pouze jedna posloupnost aplikovatelných pravidel.
- `▷Trk-` (`[Int+]`, volitelný / [], dodatečné vyhodnocení) – seznam pravidel pro negativní sledování. Komponenta musí přijmout pomocí jiné posloupnosti pravidel, než udává seznam `▷Trk-` (záleží na pořadí aplikace pravidel). Na rozdíl od algoritmu 5.4.5 je zde povolena pouze jedna zakázaná posloupnost pravidel.
- `Preset` (`[Symbol]`, volitelný / [], automatické vyhodnocení) – seznam symbolů, které se mají vložit na zásobník komponenty před započetím samotné syntaktické analýzy. Tento port slouží k realizaci replikace, konverze a aplikace funkce. Symboly jsou na zásobník vkládány v opačném pořadí, než udává seznam `Preset`.

Výstupní porty jsou opět ve shodě s algoritmem vyhodnocení komponenty:

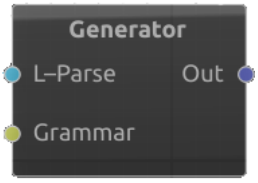
- `Parsed` (`[Symbol]`, []) – zpracovaná část vstupní věty. V případě přijetí je na výstupním portu `Parsed` seznam obsahující právě ty symboly, které odvodila komponenta v procesu syntaktické analýzy (zleva doprava). V případě odmítnutí je na tomto portu prázdný seznam.
- `Remaining` (`[Symbol]`, []) – zbývající část vstupní věty. V případě přijetí je na výstupním portu `Remaining` seznam obsahující všechny lexémy, které následují ve vstupní větě za zpracovanou částí `Parsed`. Na tento port lze pohlížet, jako na aktuální pozici vstupu daného lexikálním analyzátorem – vždy bude ukazovat na následující lexému. Byla-li již přijata celá věta, bude zřejmě seznam `Remaining` obsahovat pouze $. Propojením tohoto výstupního portu se vstupním portem `Tokens` jiné komponenty je možné modelovat relaci $\circ$.
- `Steps` (`Int+`, 0) – počet kroků komponenty. V případě přijetí je na výstupním portu `Steps` číslo reprezentující počet aplikovaných pravidel, v případě odmítnutí je zde konstanta 0.
- `L-Parse` (`[Int+]`, []) – levý rozbor. V případě přijetí je na výstupním portu `L-Parse` levý rozbor, tedy posloupnost aplikovaných pravidel. V případě odmítnutí obsahuje port prázdný seznam. Délka seznamu `L-Parse` je nutně shodná s hodnotou portu `Steps`.

Dále lze pomocí dialogu upravit parametry tohoto bloku, můžeme volit mezi:

(a) Funkční blok `Grammar-Controlled Parser` reprezentují komponentu typu CG.



(b) Funkční blok `Comparator` reprezentují komponentu typu C.



(c) Funkční blok `Generator` generující větu z levého rozboru.

Obrázek 6.2.5 Komponenty C, CG a G.

1. Dodatečným a průběžným vyhodnocením pozitivní synchronizace `Sync+` – dodatečné vyhodnocení je provedeno po ukončení syntaktické analýzy přesně podle algoritmu 5.4.5. Průběžné vyhodnocení se provádí v každé iteraci řídícího algoritmu – nejprve se spočte počet kroků k nutných k odstranění všech neterminálů, a poté se prověří, zdali množina `Sync+` neobsahuje právě číslo $k + n$, kde n je počet již provedených kroků). Tato analýza je prováděna pouze s využitím ε-pravidel. V případě nalezení $n + k$ v seznamu `Sync+` jsou neprodleně všechny neterminály zleva doprava odstraněny pomocí příslušných ε-pravidel.

2. Přiložením a nahrazením axiomu při využití portu `Preset`. Pokud je zvolena možnost přiložit axiom, pak port `Preset` realizuje slabou replikaci, slabou konverzi či slabou aplikaci funkce. Axiom je tomto případě vložen na zásobník před reverzovaným seznamem `Preset`. V opačném případě jsou využity silné verze těchto omezení a axiom je nahrazen reverzovaným seznamem `Preset`.

Jak naznačuje obrázek 6.2.4, funkční bloky pro LL a LR komponentu jsou téměř totožné. Z toho důvodu uvedeme u LR komponenty (obr. 6.2.4b) pouze rozdílné části. Samozřejmě uvnitř využívá blok jiný algoritmus syntaktické analýzy, konkrétně *Simple LR parsing*, který pracuje zdola nahoru. Tento algoritmus neprodukuje levý rozbor, nýbrž pravý rozbor, a proto byl port `L-Parse` nahrazen portem `R-Parse` (`[Int+]`, []). Seznam na tomto portu v případě přijetí obsahuje posloupnost aplikace pravidel (pravý rozbor) a v případě odmítnutí je prázdný. Tento blok navíc nelze parametrizovat – vyhodnocení synchronizace je vždy dodatečné (není implementován odhad počtu zbývajících kroků) a nelze využít slabé verze replikace, konverze či aplikace funkce (algoritmus SLR nezačíná od axiomu a nelze jej tedy zakomponovat do omezení). Všechny další funkce jsou shodné s blokem `LL Parser`.

Třetí zavedenou komponentou byla komponenta CG – řízená gramatika. Tuto komponentu realizuje funkční blok `Grammar-Controlled Parser` (obr. 6.2.5a). Tento blok má

nejblíže ke klasické bezkontextové gramatice – pomocí zadané posloupnosti pravidel generuje větu, kterou porovnává se vstupem. Přijme právě tehdy, když je generovaná věta prefixem vstupní věty a zároveň za generovanou větou následuje znak konce podseknvence. Výstupní porty jsou zcela shodné s blokem `LL Parser`, ale vstupní porty se liší:

- `Tokens` (`[Symbol]`, povinný) – vstupní věta k syntaktické analýze.
- `L-Parse` (`[Int+]`, povinný) – posloupnost identifikátorů pravidel, která mají být aplikována k odvození vstupu.
- `Grammar` (`Grammar`, povinný) – bezkontextová gramatika obsahující odvozovací pravidla.
- `{$}` (`{Symbol}`, volitelný / {$}) – množina znaků ukončující podsekvenci přijímanou komponentou. Není-li zadána, použije se množina obsahující znak $.
- `Sync+` (`[Int+]`, volitelný / [], dodatečné vyhodnocení) – množina kroků k pozitivní synchronizaci interpretovaná stejně jako u bloku `LL Parser`.
- `Sync-` (`[Int+]`, volitelný / [], dodatečné vyhodnocení) – množina kroků k negativní synchronizaci interpretovaná stejně jako u bloku `LL Parser`.

Komponenta typu CG samozřejmě musí mít zadané omezení typu t a kvůli tomu je vstupní port `L-Parse` povinný. Seznam `L-Parse` obsahuje pouze pořadová čísla pravidel, skutečná pravidla jsou poté dohledána v gramatice pomocí těchto identifikátorů. V případě zadání nevalidního seznamu, tj. pokud seznam obsahuje pravidlo, jež není v gramatice, blok samozřejmě nedokončí syntaktickou analýzu a bude indikovat chybu. Toto chování má ovšem jednu podstatnou výhodu související s realizací mapovaní $m(\mu)$. Při návrhu SA systému nám pravděpodobně vznikne více jednoduchých gramatik, které budeme propojovat. Pracuje-li předchůdce komponenty G_i^{CG} s gramatikou G_j zatímco komponenta G_i^{CG} používá model G_i, pak není potřeba v modelu SA systému explicitně specifikovat mapování mezi pravidly. Jednoduše pak n-té pravidlo z G_j odpovídá n-tému pravidlu z G_i a není potřeba vůbec uvádět množinu μ.

Poslední komponentou prezentovanou v teoretické části je komponenta typu C, ta je reprezentována funkčním blokem `Comparator`. Jedná se o velice jednoduchý blok, který má pouze tři vstupy:

- `In` (`[Symbol]`, povinný) – vstupní věta k syntaktické analýze.
- `Cmp` (`[Symbol]`, povinný) – očekávaná posloupnost symbolů (prefix vstupní věty).
- `{$}` (`{Symbol}`, volitelný / {$}) – množina znaků ukončující podsekvenci přijímanou komponentou.

Tento blok pouze prověří, jestli je seznam `Cmp` prefixem seznamu `In` a zároveň další lexéma v `In` je ukončující znak podsekvence (případě $ ukončující větu). Přesněji řečeno, komponenta příjme právě když: $\texttt{In} = xyz \wedge x = \texttt{Cmp} \wedge y \in \texttt{\{\$\}} \wedge z \in T^*$, kde `{$}` je množina ukončujících znaků podseknvence zpracovávané komponentou. V případě přijetí je na výstupním portu `Out` (`[Symbol]`, []) seznam obsahující vstupní lexémy následující za prefixem `Cmp`, respektive řetězec yz. V případě odmítnutí je na výstupním portu prázdný seznam.

Další komponentou, kterou jsme si dříve neuvedli, ale je logickým doplňkem ke komparátoru, je generátor G. Ostatně na komponentu CG se lze podívat jako na generátor G a komparátor C spojené do jednoho celku. Při implementaci jsme proto ponechali i oddělenou verzi komponenty G v podobě bloku `Generator`. Tato komponenta jednoduše využije levý rozbor z portu `L-Parse` (`[Int+]`, povinný) k výběru a aplikaci pravidel specifikovaných gramatikou na portu `Grammar` (`Grammar`, povinný). Výběr a aplikace probíhá stejným způsobem, jako u bloku `Grammar-Controlled Parser`. Generovaný řetězec je poté k dispozici na výstupním portu `Out`. Je-li špatně zadán levý rozbor (obsahuje-li pravidlo, které není v gramatice), pak blok indikuje chybu a na výstupu je prázdný seznam.

6.2.5 Operace (`Operations`)

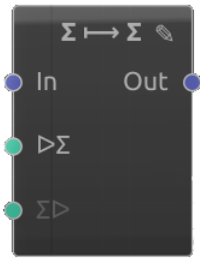
Mezi jednotlivými komponentami potřebujeme modelovat omezení z $\mathcal{R}$. Některá omezení lze modelovat přímo, například pozitivní (`Steps` → `Sync+`) či negativní synchronizaci (`Steps` → `Sync-`), sledování (`L-Parse` → `▷Trk+`) nebo replikaci (`Parsed` → `Preset`). Pro modelování složitějších omezení, jako jsou například konverze, aplikace funkce, mapování či synchronizace od více komponent, je nutné zavést další funkční bloky.

1. `Conversion` (konverze, obr. 6.2.6a) – blok překlad vstupní věty na výstupní prostou náhradou jednoho symbolu za jiný. Množinu náhrad udává množina ϕ definovaná jako součást bloku pomocí dialogu pro zadávání parametrů. Věta k překladu je zadána na vstupní port `In` (`[Symbol]`, povinný). Dále jsou k dispozici 2 porty pro zadávání abecedy, na portu $\triangleright\Sigma$ (`{Symbol}`, volitelný / $\emptyset$) se zadává vstupní abeceda, zatímco na portu $\Sigma\triangleright$ (`{Symbol}`, volitelný / $\emptyset$) se zadává abeceda výstupní. Nejsou-li tyto abecedy zadány, blok realizuje libovolnou substituci symbolu za jiný (symbol nemusí vůbec být v abecedě T). Se zadanými referenčními abecedami provádí blok i kontrolu, jestli je provedená konverze validní. Výsledek konverze (v případě, že je validní) je poté na výstupním portu `Out` (`[Symbol]`, []). Důležité je poznamenat, že pokud pro symbol $a \in T$ neexistuje pravidlo ve ϕ, pak je symbol a přidán na výstup v nezměněné podobě.

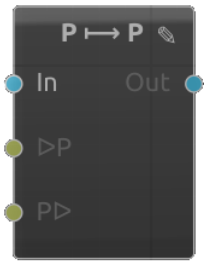
2. `Mapping` (mapování pravidel, obr. 6.2.6b) – blok převádějící vstupní seznam identifikátorů pravidel na výstupní seznam náhradou jednoho pravidla za jiné. Množinu náhrad udává množina μ, která je opět definována v rámci bloku pomocí parametrizačního dialogu. Originální seznam pravidel je zadáván na vstupu `In` (`[Int+]`, povinný) a mapovaný seznam na výstupním portu `Out` (`[Int+]`, []). Stejně jako u bloku `Conversion` je možné zadávat referenční množiny – `▷P` pro specifikaci vstupních pravidel a `P▷` pro specifikaci výstupních pravidel. Při zadání těchto množin blok kontroluje platnost mapování μ, v opačném případě pouze nahradí jeden identifikátor za jiný.

3. `Function Aplication` (aplikace funkce, obr. 6.2.6c) – je blok pro realizaci libovolného Turingovsky vyčíslitelného výpočtu. Prostřednictvím parametrizačního dialogu lze zadat libovolný skript v jazyce `ECMAScript`. Tento skript se vyhodnotí vždy při změně některého ze vstupů:

   (a) `In` (`[Symbol]`, povinný) – vstupní věta, na níž má být funkce aplikována. Ve skriptu je tato věta k dispozici jako pole `sentence`. Lze na ni aplikovat libovolné operace nad datovým typem `array`.



(a) Funkční blok `Conversion` realizující omezení typu $c(\phi)$.

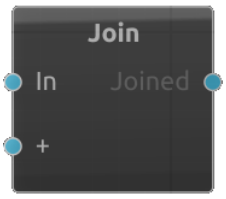


(b) Funkční blok `Mapping` realizující omezení typu $m(\mu)$.

(c) Funkční blok `Function Application` realizující obecné omezení typu $f(g)$.

(d) Funkční blok `Reverse` realizující reverzaci řetězce.

(e) Funkční blok `Concatenation` realizující konkatenaci řetězců.



(f) Funkční blok `Join` realizující spojení posloupností pravidel.

(g) Funkční blok `Inversion` realizující obrácení posloupnosti pravidel.

Obrázek 6.2.6 Operace pro realizaci omezení.

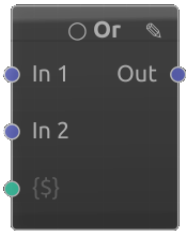
(b) ▷`Aux` (`Int`, volitelný) – pomocný vstup pro celé číslo. Ve skriptu lze k hodnotě přistupovat prostřednictvím proměnné `aux`.

Upravená hodnota výše zmíněných proměnných se po vyhodnocení skriptu propaguje na výstupní porty:

- `Out` (`[Symbol]`, []) – hodnota pole `sentence`,
- `Aux`▷ (`Int+`, 0) – hodnota proměnné `aux`.

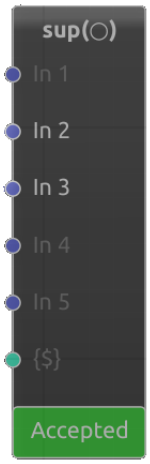
Výstupní porty nabývají platné hodnoty pouze pokud lze skript vyhodnotit a nevznikne běhová chyba. K vyhodnocení je využita implementace vestavěná v `Qt`, tedy modul `QScript`.

(a) Funkční blok ∘`And` realizující booleavskou operaci ∧ nad kompozicí ∘.



(b) Funkční blok ∘`Or` realizující booleavskou operaci ∨ nad kompozicí ∘.

(c) Funkční blok ∘`Not` realizující booleavskou operaci ¬ nad kompozicí ∘.



(d) Funkční blok `Language Acceptor` pro specifikaci jazyka přijímaného SA systémem.

Obrázek 6.2.7 Funkční bloky nad relací ∘.

4. `Reverse` (reverzace řetězce, obr. 6.2.6d) – blok, který provádí klasickou reverzaci řetězce. Blok obrátí vstupní seznam `List` (`[Symbol]`, povinný) na výstupní seznam `Rev` (`[Symbol]`, []).

5. `Concatenation` (konkatenace řetězců, obr. 6.2.6e) – blok provádějící běžnou konkatenaci řetězců. Na výstupním portu `Conc` (`[Symbol]`, []) je vždy přítomný vstupní seznam `In` (`[Symbol]`, povinný), za kterým následuje seznam `+` (`[Symbol]`, volitelný / []). Při nevyužití portu `+` realizuje blok konkatenaci řetězce `In` s ε.

6. `Join` (spojení pravidel, obr. 6.2.6f) – je blok spojující dva seznamy pravidel. Na výstupním portu `Joined` (`[Int+]`, []) je k dispozici vstupní seznam `In` (`[Int+]`, povinný), za kterým následuje seznam `+` (`[Int+]`, volitelný / []).

7. `Inversion` (inverze pravidel, obr. 6.2.6g) – blok realizující inverzi seznamu pravidel. Vstupní seznam `In` (`[Int+]`, povinný) je v opačném pořadí dostupný na výstupním portu `Inv` (`[Int+]`, []).

6.2.6 Prvky kompozice (relace ∘, `Composition`)

Poslední kategorií funkčních bloků jsou bloky nad relací ∘. Jak jsme již uvedli v sekci 5.1.1, relaci lze větvit a spojovat za pomoci logických operátorů ∧, ∨, ¬ a ⊥. Tyto operátory jsou realizovány bloky:

1. ∘`And` (operátor *a* nad ∘, obr. 6.2.7a) – blok realizující logické *a* nad komponentami. Na vstupních portech `In 1` (`[Symbol]`, povinný) a `In 2` (`[Symbol]`, povinný) je očekáván výstup z komponent. Jsou-li oba zadané vstupy validní (obě propojené komponenty přijaly), pak blok na výstupní port `Out` (`[Symbol]`, []) propaguje některý vstupní seznam v závislosti na zadaném módu (mód je zadáván prostřednictvím parametrizačního dialogu):

   - `Equal` – výsledek operace je kladný, pokud obě komponenty přijaly a přijaté věty se shodují,
   - `Left` – odpovídá $\wedge^L$. Výsledek operace je kladný, pokud obě komponenty přijaly. Na výstupu `Out` je seznam `In 1`.
   - `Right` – odpovídá $\wedge^R$. Výsledek operace je kladný, pokud obě komponenty přijaly. Na výstupu `Out` je seznam `In 2`.

   Blok nabízí ještě jednu funkci – v dialogu lze zvolit možnost „kontrolovat ukončující znaky“, pak se po vyhodnocení provede korekce a výsledek operace je platný pouze pokud oba vstupy začínají znakem z množiny `{$}` (`{Symbol}`, volitelný / {$}), která obsahuje všechny ukončující znaky podsekvence.

2. ∘`Or` (operátor *nebo* nad ∘, obr. 6.2.7b) – blok realizující logické *nebo* nad komponentami. Na vstupních portech `In 1` (`[Symbol]`, povinný) a `In 2` (`[Symbol]`, povinný) je očekáván výstup z komponent. Je-li alespoň jeden ze vstupů validní (některá propojená komponenta přijala), pak je výsledek operace kladný a výstupním portu `Out` (`[Symbol]`, []) je výstup z komponenty, která přijala. V případě přijetí obou komponent záleží na vybraném módu:

   - `Equal` – přijaté věty se musí shodovat, jinak je výsledek operace negativní.
   - `Left` – odpovídá $\vee^L$. Na výstupu `Out` je seznam `In 1`.
   - `Right` – odpovídá $\vee^R$. Na výstupu `Out` je seznam `In 2`.

   Blok umožňuje provádět korekci pomocí ukončujících znaků podsekvence zadávaných na vstupním portu `{$}` (`{Symbol}`, volitelný / {$}).

3. ∘`Not` (negace nad ∘, obr. 6.2.7c) – blok realizující negaci nad komponentami. Na vstupní porty `In` (`[Symbol]`, povinný) a ¬`In` (`[Symbol]`, povinný) jsou připojeny komponenty. Přijme-li komponenta na `In`, pak je její výstup propagován na výstup `Out` (`[Symbol]`, []). Odmítne-li tato komponenta, ale přijme-li druhá komponenta na vstupu ¬`In`, pak je výstup druhé komponenty zapsán na port `Out`. V případě odmítnutí obou komponent je na výstupu prázdný seznam. Opět je k dispozici korekce pomocí množiny `{$}` (`{Symbol}`, volitelný / {$}).

Nejpodstatnějším blokem z této kategorie je blok `Language Acceptor` (*sup*(∘), obr. 6.2.7d), který realizuje přijímání věty SA systémem pomocí logického součinu nad horními uzávěry relace ∘. Tento blok nabízí celkem 5 portů pro komponenty `In 1-5` (`[Symbol]`, volitelný, nejméně jeden musí být využitý) a jeden port pro zadání množiny `{$}` (`{Symbol}`, volitelný / {$}). Přesně podle definice 5.1.8 musí pro přijetí věty přijmout všechny připojené komponenty. Zároveň potřebujeme mít jistotu, že jsme zpracovali celou vstupní větu, a proto všechny vstupy `In 1-5` musí začínat znakem z `{$}`.

6.3 Příklad modelu

Nyní si ukážeme dva příklady modelů pro SA systém – na prvním, jednoduchém, si ukážeme zpracování morfismu, zatímco na druhém, složitějším, si ukážeme práci s Booleovou algebrou nad SA systémem. Uvažujme jazyk

$$L_{\pm} = \{wm(w)m(w)^r \mid w \in \{a,b\}^*, m(w) \in \{c,d\}, m(a) = dc, m(b) = cc\},$$

kde $m(w)$ je morfismus nad w. Pomocí vytýkání dostaneme jazyky:

$$\begin{aligned} w\|L_1 &= \{w \mid w \in \{a,b\}^*\}, \\ \bullet w\|L_2 &= \{m(\bullet w)m(\bullet w)^r \mid \bullet\, w \in \{a,b\}^*, m(\bullet w) \in \{c,d\}, m(a) = dc, m(b) = cc\}. \end{aligned}$$

Jazyk L_2 spadá do CF, ale není deterministický. Normálně by bylo potřeba provést další dekompozici, ale v tomto případě nám informaci o středu řetězce poskytne komponenta zpracovávající L_1, tudíž není potřeba L_2 dále členit a budeme k němu při návrhu systému přistupovat jako k běžné závorkové struktuře. Morfismus navíc přímo nabádá k využití mapování pravidel. Sestrojíme tedy gramatiky:

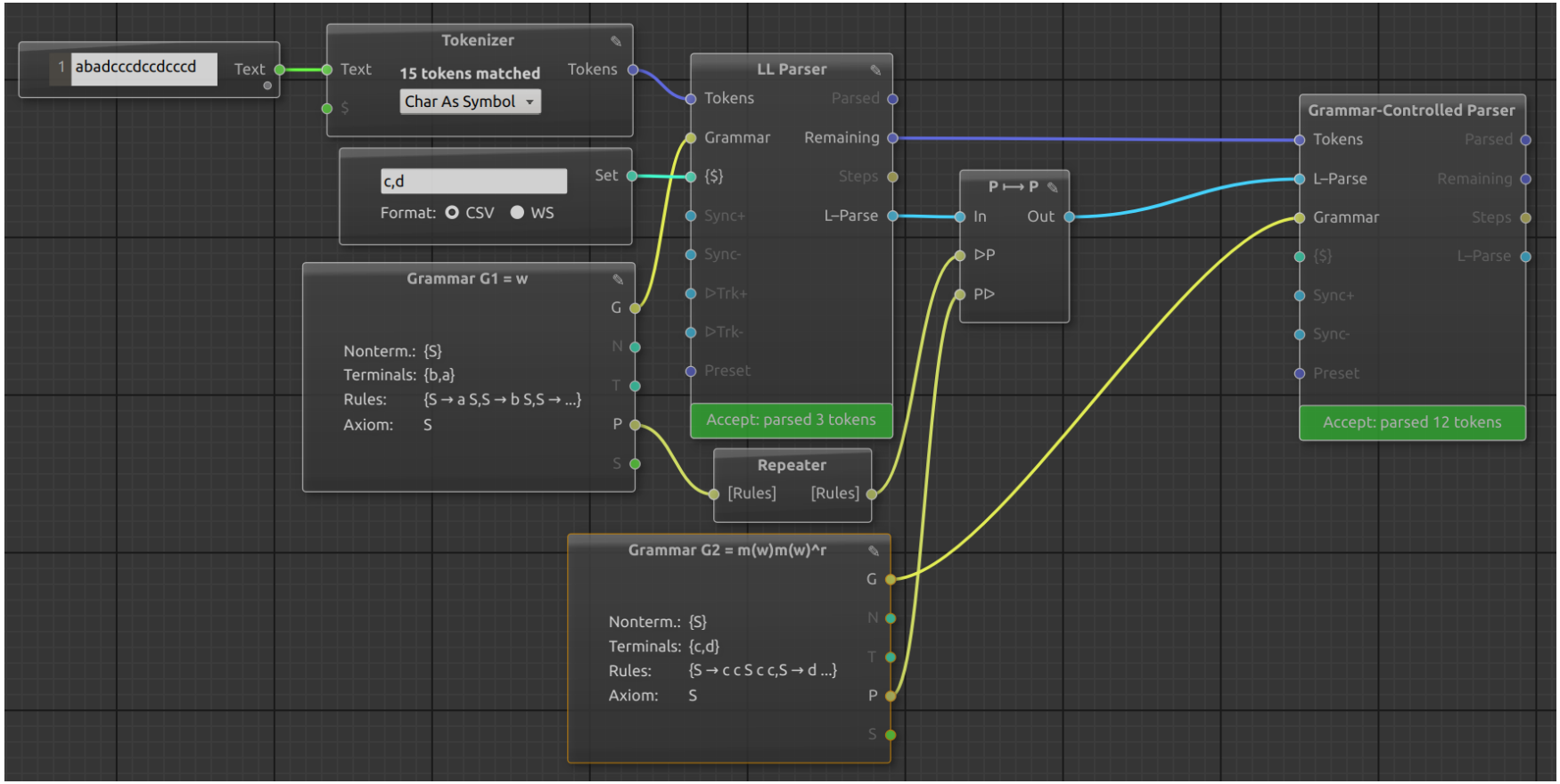
$$\begin{aligned} T &= \{a,b,c,d\}, \\ G_1 &= (\{S\}, T, \{S \to aS \mid bS \mid \varepsilon\}), \\ G_2 &= (\{S\}, T, \{S \to ccScc \mid dcScd \mid \varepsilon\}). \end{aligned}$$

Zavedeme SA systém:

$$\begin{aligned} &g_1 = G_1^{LL}, g_2 = G_2^{CG}, \circ = \{(g_2, g_1)\}, \mathcal{R} = \{(g_1, g_2, m(\mu))\}, \text{ kde} \\ &\mu = \{(S \to aS, S \to dcScd), (S \to bS, S \to ccScc), (S \to \varepsilon, S \to \varepsilon)\}. \end{aligned}$$

Jazykem tohoto systému jsou řetězce spadající do $L(g_2 \circ g_1)$. Bez dalších detailů uvádíme model zobrazený na obrázku 6.3.1.

Vstup modelu 6.3.1 (vstup SA systému) je udáván nejlevějším blokem `Program Source` – pro ověření je využita věta *aabdcdccccccdcd*. `Tokenizer` z této věty vytvoří posloupnost vstupních lexém, které jsou předány komponentám. Komponenta G_1^{LL} (g_1) přijme jakýkoliv řetězec nad abecedou $\{a,b\}$. Pravidla, která přitom použije, jsou mapována prostřednictvím množiny μ (viditelná v dialogovém okně) a jejich obraz poté použije řízená gramatika G_2^{CG} (g_2) k vygenerování morfismu a jeho reverzace. Věty přijaté g_2 jsou přijaté celým systémem, neboť se jedná o poslední komponentu v relaci $\circ$ (v modelu není nutné uvádět explicitně přijímaný jazyk). Mapování množiny μ jsme uvedli explicitně, ale při prohození pořadí pravidel $S \to ccScc$ a $S \to dcScd$ bychom mohli blok `Mapping` vynechat, a přesto by systém plnil zamýšlenou úlohu.



Obrázek 6.3.1 Model SA systému pro jazyk $L = \{wm(w)m(w)^r \mid w \in \{a,b\}^*, m(w) \in \{c,d\}, m(a) = dc, m(b) = cc\}$.

Další příklad je mírně komplexnější, neboť budeme pracovat s větvenou relací $\circ$, a tedy i Booleovou algebrou. Uvažujme kontextový jazyk $L_\diamond = \{a^n b^n c^n \vee a^n w^n w^n \mid n \geq 1, w \in \{b,c\}^*\}$. Je zřejmé, že tento jazyk je disjunkcí dvou jiných kontextových jazyků. Od toho se bude odvíjet náš postup při návrhu SA systému – nejprve rozložíme $L_\diamond$ na dva logické celky:

$$L_1 = \{a^n b^n c^n \mid n \geq 1\},$$
$$L_2 = \{a^n w^n w^n \mid n \geq 1, w \in \{b,c\}^*\}.$$

Poté vytkneme společnou část a^n a řetězec $w^n w^n$ rozdělíme na dva samostatné:

$$n \| L_1 = \{a^n \mid n \geq 1\},$$
$$\bullet n \| L_2 = \{b^{\bullet n} c^{\bullet n} \mid n \geq 1\},$$
$$\bullet n, w \| L_3 = \{w^{\bullet n} \mid w \in \{b,c\}^*, \bullet n \geq 1\},$$
$$\bullet n, \bullet w \| L_4 = \{\bullet w^{\bullet n} \mid \bullet\, w \in \{b,c\}^*, \bullet n \geq 1\}.$$

V dalším kroku si pro tyto jazyky vyjma L_4 sestrojíme gramatiky. Pro L_4 gramatiku neuvádíme, protože je shodná s gramatikou pro L_3.

$$T = \{a,b,c\},$$
$$G_1 = (\{S_1\}, T, \{S_1 \to aS_1 \mid \varepsilon\}, S_1),$$
$$G_2 = (\{S_2\}, T, \{S_2 \to bS_2c \mid \varepsilon\}, S_2),$$
$$G_3 = (\{S_3\}, T, \{S_3 \to bS_3 \mid cS_3 \mid \varepsilon\}, S_3).$$

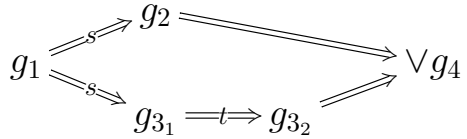
Dále zavedeme jednotlivé komponenty a omezení. Zřejmě je vhodné využít synchronizaci právě n kroků, navíc budeme muset zajistit opakování posloupnosti w_n, což lze například prostřednictvím sledování. Samozřejmě by bylo možné využít i replikaci. Dostáváme tedy relace:

$$g_1 = G_1^{LL}, g_2 = G_2^{LL}, g_{3_1} = G_{3|1}^{LL}, g_{3_2} = G_{3|2}^{CG},$$
$$\circ = \{(g_2, g_1), (g_{3_1}, g_1), (g_{3_2}, g_{3_1})\},$$
$$\mathcal{R} = \{(g_1, g_2, s), (g_1, g_{3_1}, s), (g_{3_1}, g_{3_2}, t)\}.$$

Nyní je ještě nutné systém logicky popsat, tedy zavést funkci λ. Podle definice 5.1.8 je jazyk systému konjunkcí koncových komponent ze $sup(\circ)$, ovšem my musíme podle specifikace jazyka $L_\diamond$ použít disjunkci namísto konjunkce. Tento rozpor lze vyřešit jednoduchým trikem – zavedeme gramatiku G_4 přijímající jazyk $\{\varepsilon\}$, která bude v relaci následovat za g_2 i za g_{3_2} a bude realizovat logické nebo, tedy:

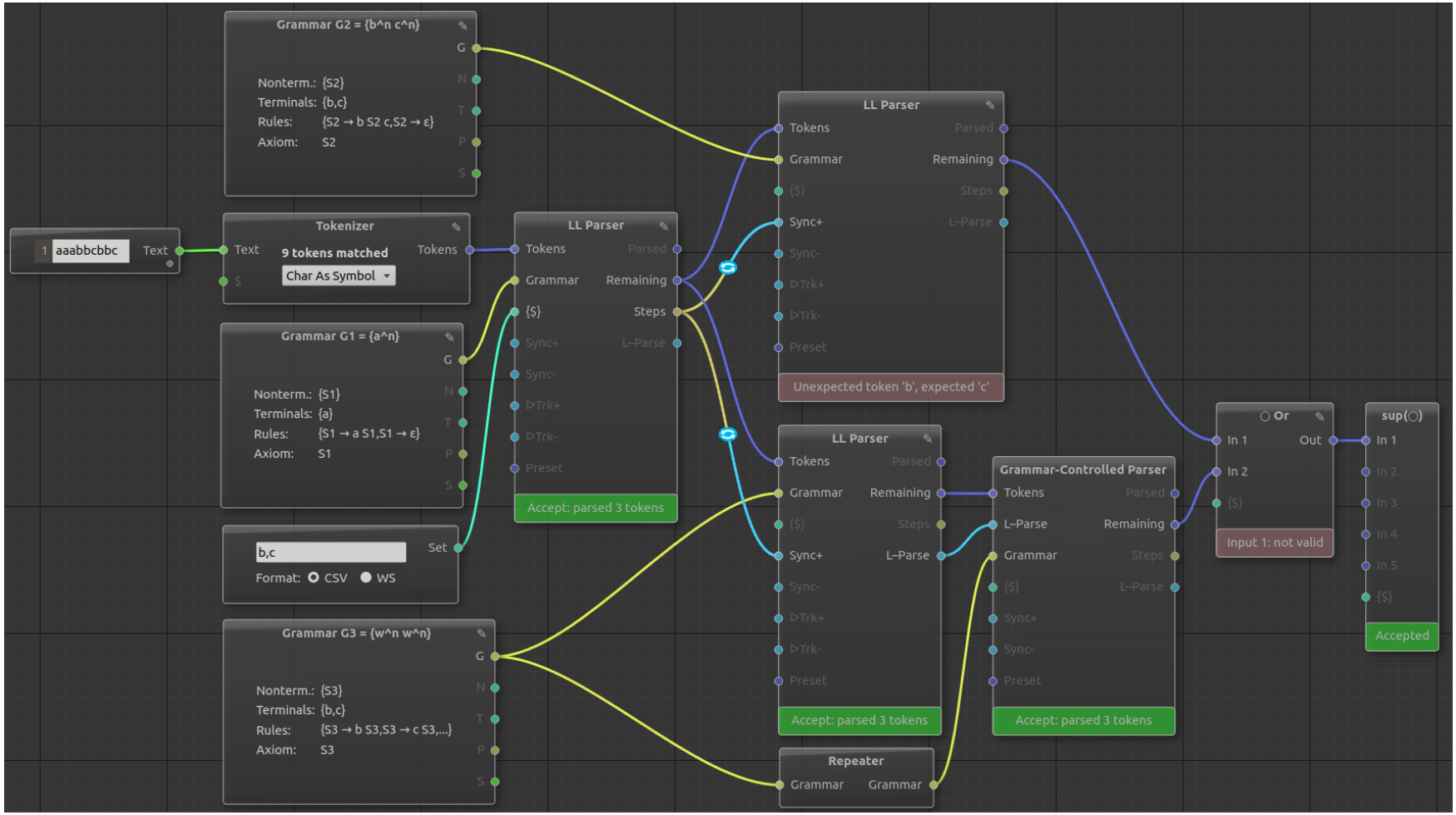
$$G_4 = (\{S_4\}, T, \{S_4 \to \varepsilon\}, S_4), g_4 = G_4^{LL},$$
$$\circ = \{(g_2, g_1), (g_{3_1}, g_1), (g_{3_2}, g_{3_1}), (g_4, g_2), (g_4, g_{3_2})\},$$
$$\mathcal{R} = \{(g_1, g_2, s), (g_1, g_{3_1}, s), (g_{3_1}, g_{3_2}, t)\},$$
$$\lambda(g_1) = \lambda(g_2) = \lambda(g_{3_1}) = \lambda(g_{3_2}) = \bot, \lambda(g_4) = \vee,$$
$$sup(\circ) = \{g_4\}, inf(\circ) = \{g_1\}.$$

Nyní je popis SA systému kompletní, celou strukturu zobrazuje následující graf toku:



Odpovídající model systému je zobrazen na obrázku 6.3.2. Vstupní blok `Program Source` specifikuje vstupní větu *aaabbcbbc*, z níž `Tokenizer` vytvoří posloupnost vstupních lexém. LL komponenta nad G_1 (g_1) pouze spočítá počet symbolů a a tuto informaci pomocí omezení předá G_2^{LL} (g_2) a $G_{3|1}^{LL}$ (g_{3_1}). Komponenta g_2 provádí analýzu závorkových struktur $b^n c^n$, zatímco g_{3_1} přijme libovolný řetězec nad $\{b, c\}^*$ o délce n. Za touto komponentou následuje řízená gramatika g_{3_2}, která použije shodná pravidla k odvození totožné podsekvence w^n. Blok $\circ$`Or` realizuje spojení relace $\circ$ ($\lambda(g_4) = \vee$). Podle formální specifikace systému bychom za tímto blokem měli použít komponentu g_4, ale protože tato komponenta by neměla žádný vliv na funkčnost ($x\varepsilon = x$), můžeme si dovolit ji vynechat – v programu je reprezentována přímo blokem $\circ$`Or`. Nakonec explicitně stanovíme $sup(\circ)$ pomocí bloku `Language Acceptor`.

Povšimněme si externě zadané množiny `{$}` u komponenty g_1. Tato množina (formální množina $Next_{SA}$) musí být vypočítána a zadána manuálně, neboť její výsledná podoba plyne ze tvaru jazyka. V našem případě vždy za lexémou a následuje b nebo c ($First^{g_2}(S_1) = \{b\}$, $First^{g_{3_1}}(S_3) = \{b, c\}$, $Next_{SA}^{g_1} = \{b, c\}$). Kdybychom ovšem pozměnili definici w na $w \in \{a, b, c\}^*$, zřejmě by platilo $First^{g_{3_1}}(S_3) = \{a, b, c\}$, respektive $Last^{g_1}(S_1) \cap Next_{SA}^{g_1} = \{a\} \neq \emptyset$, tedy jazyk by byl vnitřně nedeterministický a museli bychom navrhnout systém zcela odlišně (nejprve určit délku věty $3n$, a poté pomocí aplikace funkce zpracovat podsekvence a^n, w^n a w^n).



Obrázek 6.3.2 Model SA systému pro jazyk $L = \{a^n b^n c^n \vee a^n w^n w^n \mid n \geq 1, w \in \{b, c\}^*\}$.

7 Závěr

V první části práce jsme si z teoretického pohledu představili gramatické systémy jako další matematický model vhodný pro popis formálních jazyků. Diskutovali jsme jejich strukturu, generativní kapacitu a způsob práce. Rozčlenili jsme si je do dvou základních větví – sekvenčně pracujících CD gramatických systémů a paralelních PC gramatických systémů. Představili jsme dva základní cíle a zároveň důvody, proč se těmto strukturám věnovat, tj. zvýšení síly syntaktické analýzy při zachování bezkontextových pravidel a snížení složitosti navrhovaných gramatik. Jako cíl práce jsme si vytkli právě zvýšení generativní kapacity bezkontextových metod.

Prvním prezentovaným prostředkem, vedoucím k dosažení stanoveného cíle, byly týmové centralizované PC gramatické systémy ve tvaru $\sigma = \big(N, T, K, (S_1, P_1), \ldots, (S_n, P_n), R\big)$. Ke standardním PC gramatickým systémům jsme přidali množinu týmů $R \subseteq 2^P$. Týmem rozumíme libovolnou podmnožinu potenční množiny komponent. V každém týmu jsme stanovili leadera (vedoucí komponentu). Dále jsme se omezili pouze na 4 typy pravidel – derivační pravidla v Graibachové normální formě, čekací pravidla zavedená pro každý neterminál, komunikační pravidla obsahující symboly z Q a ε-pravidla, přičemž poslední 2 typy pravidel může mít pouze leader týmu.

Dále jsme se snažili upravit konvenční LL syntaktickou analýzu tak, aby pracovala s TCPC gramatickými systémy. Zavedli jsme pro každou komponentu samostatnou LL tabulku, která je rozšířená o identitu symbolizující použití čekacích pravidel (LLI tabulka). Abychom zabránili ztrátě informace při komunikaci, představili jsme mechanismus značení terminálních symbolů. Poté jsme zavedli funkci výběru aktivního týmu τ, která na základě posledního použitého týmu a vstupní lexémy určí následující tým, jenž má na větné formě pracovat. Uvedli jsme si možnosti algoritmického získávání LLI tabulek. Následně jsme představili dva stěžejní algoritmy TCPC syntaktické analýzy – algoritmus simulace kroku jedné komponenty a řídící algoritmus TCPC syntaktického analyzátoru.

Poté jsme si zavedli systémy syntaktických analyzátorů (SA systémy) ve tvaru $\sigma = (N, T, \mathcal{G}, \mathcal{P}, \circ, \mathcal{R})$, kde $\mathcal{G}$ je konečná množina gramatik (modelů) a $\mathcal{P}$ je konečná množina komponent (syntaktických analyzátorů) zavedených nad modely z $\mathcal{G}$. Zavedli jsme si komponenty typu:

- LL – syntaktický analyzátor pracující nad LL gramatikou (provádějící nejlevější derivace, provádějící syntaktickou analýzu zleva doprava). V obecné podobě lze využít místo LL komponenty libovolný jiný syntaktický analyzátor pracující shora dolů.
- LR – syntaktický analyzátor pracující zleva doprava provádějící nejpravější derivace. V obecné podobě lze využít místo LR komponenty libovolný jiný syntaktický analyzátor pracující zdola nahoru.

- CG – řízená gramatika, která jako vstup dostává seznam pravidel, jež má aplikovat k odvození vstupní věty. Odvozenou vstupní větu poté porovnává se vstupem.
- C – komparátor, tj. jednoduchý syntaktický analyzátor mající pouze pravidla typu $afa \to f$. Tento analyzátor začíná s neprázdným zásobníkem a přijímá právě pokud se mu povede vyprázdnit zásobník.

SA systémy jsme prezentovali jako rozšiřitelný koncept – množinu $\mathcal{G}$ lze rozšířit o libovolný typ gramatiky či jiný model, a poté v množině $\mathcal{P}$ zavést nad tímto modelem vlastní komponentu. Zde je zcela jistě prostor pro další zkoumání, mohli bychom v SA systému zkusit použít:

- syntaktický analyzátor zavedený nad regulární gramatikou,
- syntaktický analyzátor zavedený nad kontextovou gramatikou,
- syntaktický analyzátor zavedený nad regulovanou gramatikou,
- TCPC syntaktický analyzátor zavedený nad TCPC gramatickým systémem nebo
- GLL syntaktický analyzýtor [3] zavedený nad CD či HCD gramatickým systémem.

Přitom zůstanou platné veškeré uvedené algoritmy a relace – způsob vyhodnocení a práce SA systému se nezmění. Je ovšem možné, že zavedením těchto komponent získáme nějaké nové vlastnosti.

Výše uvedené komponenty lze skládat za sebe pomocí relace $\circ$, přičemž jsme si zavedli, že zápis $g_2 \circ g_1$ budeme číst „g_2 po g_1“ a myslíme tím, že nejdříve pracuje komponenta g_1. Přijme-li g_1 pak bude pracovat g_2. Přijme-li i g_2, pak je vstupní věta přijata. V případě odmítnutí g_1 či g_2 věta nepatří do jazyka $L(g_2 \circ g_1)$. Dále jsme na tuto relaci kladli požadavky ireflexivity, asymetričnosti a acykličnosti. Pomocí relace $\circ$ lze pak jednoduše zpracovávat sekvence.

Relace $\circ$ sice definuje pořadí, ve kterém budou komponenty pracovat na vstupní větě, což ovšem nevede ke zvýšení generativní síly. Generativní kapacitu lze zvýšit zavedením množiny $\mathcal{R}$. Tato množina obsahuje trojice udávající omezení mezi komponentami. Jedná se o obdobný koncept, jako poskytují regulované gramatiky, ale zde je využit na úrovni analyzátorů. Ke zvýšení generativní síly můžeme využít následující omezení, zavedená mezi komponentami g_i, g_j, kde g_j následuje g_i:

- Pozitivní synchronizace (s) – g_j musí přijmout pomocí stejného počtu kroků jako g_i.
- Negativní synchronizace $(\neg s)$ – g_j musí přijmout pomocí jiného počtu kroků než g_i.
- Replikace (r) – větu přijatou g_i vložíme při inicializaci na zásobník g_j místo axiomu a g_j musí přijmout.
- Slabá replikace $(\succ r)$ – větu přijatou g_i vložíme při inicializaci na zásobník g_j za axiom a g_j musí přijmout.
- Konverze $(c(\phi))$ – pro větu w přijatou g_i spočteme $\overline{w}$ prostou náhradou jednoho symbolu za jiný. Při inicializaci vložíme $\overline{w}$ na zásobník g_j místo axiomu a g_j musí přijmout.

- Slabá konverze ($\succ c(\phi)$) – pro větu w přijatou g_i spočteme $\overline{w}$ prostou náhradou jednoho symbolu za jiný. Při inicializaci vložíme $\overline{w}$ na zásobník g_j za axiom a g_j musí přijmout.
- Pozitivní sledování (t) – g_j musí přijmout pomocí stejných pravidel jako g_i.
- Negativní sledování ($\neg t$) – g_j musí přijmout pomocí jiných pravidel než g_i.
- Mapování ($m(\mu)$) – přijme-li g_i pomocí pravidel $r_1, \ldots, r_m$, pak musí g_j přijmout pomocí pravidel $\overline{r_1}, \ldots, \overline{r_m}$, kde $\overline{r_i}$ je obraz pravidla r určený množinou dvojic μ.
- Aplikace funkce ($f(g)$) – pro větu w přijatou g_i spočteme $\overline{w}$ aplikací funkce g, tj. $\overline{w} = g(w)$. Při inicializaci vložíme $\overline{w}$ na zásobník g_j místo axiomu a g_j musí přijmout.
- Slabá aplikace funkce ($\succ f(g)$) – pro větu w přijatou g_i spočteme $\overline{w}$ aplikací funkce g, tj. $\overline{w} = g(w)$. Při inicializaci vložíme $\overline{w}$ na zásobník g_j za axiom a g_j musí přijmout.

Zavedená omezení jsou do jisté míry specifická pro určité komponenty – například syntaktický analyzátor nad řízenou gramatikou (CG) bude mít vždy omezení typu t, slabá omezení lze využít pouze pro komponenty pracující shora dolů (LL) a mapování nebo sledování nelze použít pro komparátor (C). Při dalším bádání by jistě bylo zajímavé zavést nepřesná omezení známá z teorie CD gramatických systémů, tedy synchronizaci na $\geq k$ kroků (nejméně k kroků) nebo na $\leq k$ kroků (nejvýše k kroků), a poté zkoumat vlastnosti takových systémů. Zároveň při zavedení nových komponent může vyvstat potřeba zavést i nová, zcela odlišná omezení.

Dále jsme upřeli pozornost na bezkontextové SA systémy. Popsali jsme metody návrhu SA systému pro deterministické kontextové struktury, přičemž jsme využili následující metody:

1. rozklad na podsekvence – jednu větu rozložíme na více podsekvencí a pro každou podsekvenci zavedeme samostatný analyzátor,
2. vytýkání stejného vzoru – ve specifikaci jazyka nalezneme proměnnou a poté sestavíme analyzátor, který spočte její hodnotu a na základě této hodnoty řídí přijímání věty dalšími analyzátory systém.
3. nahrazení (substituce) – nalezneme izomorfismus mezi podsekvencemi věty jazyka a pomocí tohoto morfismu propojíme komponenty SA systému.

Zmínili jsme známé bezkontextové metody syntaktické analýzy založené na deterministických zásobníkových automatech a ukázali jsme upravené verze těchto metod, pomocí nichž lze navržený bezkontextový SA systém zkonstruovat. Velkou míru pozornosti jsme věnovali tvorbě LL komponent pomocí dobře známých množin $First$, $Empty$ a $Follow$ a nově zavedených množin $Last$ a $Next$. Tyto množiny jsme dále využili i k tvorbě LR komponent. Dále jsme se věnovali analýze determinismu a došli jsme ke stěžejní formuli, jež musí být splněna pro každou komponentu deterministického SA systému:

$$\forall g_i \in \mathcal{P} : Last^{g_i}(S_i) \cap Next^{g_i}_{SA} = \emptyset.$$

Tato formule udává, že musí být pevně stanoveny meze a nesmí nastat situace, kdy mohou pracovat obě komponenty v relaci $\circ$. Přesněji řečeno, musí platit tvrzení:

$g_j \circ g_i \iff$ komponenta g_i nemůže použít pravidlo $\vee$ komponenta g_j nemůže použít pravidlo.

Poté jsme se věnovali interpretaci SA systémů. Uvedli jsme způsoby vyhodnocování omezení – některá omezení se nastavují pouze při inicializaci komponenty a o vyhodnocení se implicitně stará řídící algoritmus komponenty (automatické vyhodnocení), některá omezení postačuje kontrolovat až po ukončení syntaktické analýzy s příznivým výsledkem (dodatečné vyhodnocení) a jiná omezení je nutné zabudovat přímo do řídícího algoritmu komponenty (průběžné vyhodnocení). Dále jsme si zavedli reentrantní lexikální analyzátor – automat provádějící klasickou lexikální analýzu vstupní věty, ale mající tu devízu, že k přečteným lexémám může nezávisle a kvaziparalelně přistupovat více komponent. Věnovali jsme pozornost interpretaci relace $\circ$ a množině $\mathcal{R}$ – provedli jsme analýzu acykličnosti a navrhli algoritmus, pomocí kterého lze jednoznačně určit pořadí vyhodnocování komponent systému. Před uzavřením této kapitoly jsme si ještě představili algoritmus vyhodnocení komponenty, který dokáže validovat a aplikovat omezení.

Na závěr jsme prezentovali pozitivní výsledky – SA systémy vedou ke zvýšení generativní kapacity. Navíc nejsou příliš náročné na implementaci a jsou velice dobře rozšiřitelné. Z práce vyplývá, že SA systémy zcela jistě mohou být alternativou ke kontextovým metodám syntaktické analýzy. Poskytují dostatečnou sílu na zpracování rodiny jazyků CS, respektive $CS = L(SA_f)$. Tato ekvivalence plyne z možnosti použít Turingovsky silnou aplikaci funkce, ovšem i bez aplikace funkce lze pokrýt mnoho jazyků z rodiny DCS, což je hlavní devíza těchto systémů. Podařilo se nám rozšířit repertoár bezkontextových metod syntaktické analýzy jejich propojením. Přestože pořád zůstáváme v rovině bezkontextových matematických modelů, můžeme uvedenými metodami zpracovávat i vybrané kontextové jazyky.

V jazyce C++ s využitím grafického balíčku Qt jsme vytvořili multiplatformní nástroj pro popis SA systémů. Pomocí tohoto nástroje lze ve formě diagramu funkčních bloků modelovat SA systémy a posléze ověřit jimi přijímaný jazyk. Implementovali jsme navržené algoritmy konstrukce komponent a interpretace systému. Pokud by pokračoval vývoj uvedeného nástroje, mohlo by se dosáhnout zajímavých výsledků – na základě diagramu SA systému je možné vygenerovat kód syntaktického analyzátoru (založeného na SA systému) například v jazyce C++. Při realizaci abstraktního syntaktického stromu a přidání sémantických kontrol by mohl vzniknout nástroj podobný nástrojům typu `yacc`, který je určený pro modelování překladačů a zároveň poskytuje výhodu vyšší generativní síly.

V této oblasti je stále veliký prostor pro další bádání – určitě by bylo zajímavé se věnovat analýze SA systému na základě grafu toku a na základě této analýzy navrhnout optimalizovaný algoritmus výběru komponenty. Námi navržený algoritmus bude zcela jistě dostačovat pro jednoduché (nevětvené systémy), ale zdaleka není optimální. Mnohem významnějším a zároveň prakticky zaměřeným problémem je celá sémantická analýza. Bylo by vhodné se zaměřit na metody generování abstraktních syntaktických stromů v rámci komponent, a především navrhnout formální model a algoritmus, který jednotlivé stromy poskládá v jeden jediný strom celého SA systému. Se zavedením SA analyzátoru se otevírá možnost hierarchického skládání SA systémů – jeden SA systém by zcela jistě mohl plnit roli komponenty v rámci jiného nadřazeného SA systému. Podobně se lze věnovat zařazení SA komponenty do klasických metod syntaktické analýzy – smysluplné může být například použít SA systém pro analýzu složitých kontextových výrazů zabudovaných do bezkontextové gramatiky. Nadřazená metoda by v takovém případě zpracovávala rámec programu, a v místě, kde je očekáván kontextový výraz, by využila podřízený SA systém. Neméně za-

jímavé by bylo věnovat se možnostem paralelního zpracování jazyků – SA systém lze velice dobře paralelizovat po řezech relace $\circ$. K praktickému využití je ovšem potřeba navrhnout algoritmus rozkladu obecného SA systému na potřebný počet částí (počet procesů podle počtu procesorů), který dovede pracovat i s omezeními. Práce se pouze dotýká široké oblasti gramatických systémů, ve které je mnoho prostoru pro další bádání. Snaží se poskytnout základní přehled znalostí z této oblasti, či jít příkladem, v naději, že na ni bude navázáno.

Literatura

[1] Aho, A. V.: *Compilers, Principles, Techniques & Tools*, 2nd ed. Boston: Addison Wesley, 1994, ISBN 0-301-48681-1.

[2] G., R.: Is there an example of a recursive language which is not context sensitive? 2016.
Dostupné z: https://cs.stackexchange.com/questions/56632/is-there-an-example-of-a-recursive-language-which-is-not-context-sensitive.

[3] Hrstka, J.: *Gramatické systémy a syntaktická analýza na nich založená*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2017.

[4] KNUTH, D. E.: On the Translation of Languages from Left to Right. *Information and control*, 8, 1965: s. 607–639.

[5] Lucian, I., Salomaa, A.: *2-testability and relabelings produce everything*. Turku: Turku Centre for Computer Science, 1996, ISBN 9516507522.

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

[7] Rozenberg, G., Salomaa, A., aj.: *Handbook of formal languages: background and application*. Berlin: Springer-Verlag, 1997, ISBN 3-540-60648-3.

[8] Rozenberg, G., Salomaa, A., aj.: *Handbook of formal languages: word, language, grammar*. Berlin: Springer-Verlag, 1997, ISBN 3-540-60420-0.

[9] Takita, B.: Is C++ context-free or context-sensitive? 2013.
Dostupné z: http://stackoverflow.com/questions/14589346/is-c-context-free-or-context-sensitive.

[10] Trevor, J.: Python is not context free. 2012.
Dostupné z: http://trevorjim.com/python-is-not-context-free.

[11] Černá, I., Kretínský, M., Kučera, A.: Automaty a formální jazyky I. 2002, verze 1.3.