Implementation and evaluation of k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars

An der Technischen Universität Dresden,
Fakultät Informatik, Institut für
Theoretische Informatik eingereichte

Masterarbeit

zur Erlangung des akademischen Grades
eines Master of Science (M. S.)

vorgelegt von

Thomas Ruprecht

Revision May 15, 2018

geboren am 24.03.1994 in Elsterwerda
Matrikel-Nr.: 3849967

Erstgutachter: Univ.-Prof. Dr.-Ing. habil. Heiko Vogler
Zweitgutachter: Dr.-Ing. Stefan Borgwardt

.

Aufgabenstellung für die Masterarbeit

„Implementation and evaluation of k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars“

Technische Universität Dresden
Fakultät Informatik

Student:	Thomas Ruprecht
Geburtsdatum:	24. März 1994
Matrikelnummer:	3849967
Studiengang:	Informatik (Master)
Immatrikulationsjahr:	2012
Studienleistung:	29 LP (23 Wochen)
Beginn am:	2. Oktober 2017
Einzureichen am:	11. März 2018
Verantw. Hochschullehrer:	Prof. Dr.-Ing. habil. Heiko Vogler
1. Gutachter:	Prof. Dr.-Ing. habil. Heiko Vogler
2. Gutachter:	TBA
Betreuer:	Dipl.-Inf. Tobias Denkinger

Multiple kontextfreie Grammatiken. Natürliche Sprachen weisen Merkmale auf, die von kontextfreien Grammatiken nicht darstellbar sind, z.B. kann ein Teilsatz eine Lücke haben, in die vom Kontext abhängiger Inhalt eingefügt wird. Um die hohe Parsingkomplexität kontextsensitiver Grammatiken (PSPACE-complete) zu vermeiden, untersucht man Formalismen, die solche Lücken zwar darstellen können, aber dennoch polynomiell parsebar sind. Man fasst solche Formalismen unter dem Begriff *mildly context-sensitive formalisms* zusammen. Dazu gehören z.B. head grammars, tree adjoining grammars, combinatory categorial grammars, linear indexed grammars, linear context-free rewriting systems, und minimalist grammars. *Multiple kontextfreie Grammatiken* (kurz: MCFGs) wurden von Pollard [Pol84] im Kontext natürlicher Sprachen eingeführt [siehe auch Sek+91]. Alle oben genannten (und noch einige weitere) Formalismen erzeugen eine kleinere oder die gleiche Klasse von Stringsprachen wie MCFGs [Sek+91; VWJ86; WJ88; Vij88; Mic01a; Mic01b]. Das Parsing von MCFGs ist daher von besonderer Bedeutung für die Verarbeitung natürlicher Sprachen [Eva11].

Gewichtete MCFGs. In der Verarbeitung natürlicher Sprache wird die Zugehörigkeit eines Wortes zu einer Sprache oftmals nicht als Wahrheitswert, sondern als reelle Zahl zwischen 0 und 1 formuliert. Man erhält damit einen *Grad der Zugehörigkeit* des Wortes zur Sprache. So können z.B. häufige von seltenen sprachlichen Konstruktionen unterschieden werden. Zur Gewichtung einer MCFG können nicht nur reelle Zahlen, sondern auch andere Algebren verwendet werden; nötig ist dabei nur eine Operation zur *Multiplikation* der Regelgewichte in einem Parsebaum und eine Operation zur *Addition* der Gewichte verschiedener Parsebäume. Siehe dazu Goodman [Goo99] und Nederhof [Ned03].

k-best Chomsky-Schützenberger-Parsing für gewichtete MCFGs. Unter Parsing versteht man die Überführung eines Wortes in die Menge aller Ableitungen dieses Wortes in einer gegebenen Grammatik. Das Chomsky-Schützenberger-Parsing für MCFGs [Den17] nutzt dazu die Representation der durch die gewichtete MCFG gegebenen gewichteten Sprache als das homomorphe Bild des Schnittes einer regulären Sprache mit einer multiplen Dycksprache [Den15]. Der in der Representation verwendete Homomorphismus soll im Folgenden mit h bezeichnet werden, die reguläre Sprache mit R und die multiple Dycksprache mit mD. Oft interessiert man sich beim Parsing einer gewichteten Grammatik nur für die im Hinblick auf ihr Gewicht besten k Ableitungen, das sogenannte *k-Best Parsing* [HC05; Büc+10].

Rustomata und OpenFST. *Rustomata*[1] ist ein am Lehrstuhl für Grundlagen der Programmierung entwickeltes Framework, in dem Algorithmen für die maschinelle Verarbeitung natürlicher Sprachen implementiert werden können. Der Fokus von rustomata liegt auf dem Einsatz von Automaten mit Speicher für das Parsing von natürlichen Sprachen. Das Framework ist in Rust geschrieben. Rustomata besteht aus einer Bibliothek mit allgemein spezifizierten Datenstrukturen und Algorithmen, und einer Sammlung von über ein Kommandozeileninterface erreichbaren Funktionen.

OpenFST[2] ist eine Open-Source-Bibliothek, die von Mitarbeitern von Google Research und dem NYU Courant Institute entwickelt wird. Es bietet Funktionen für die Konstruktion, die Kombination, die Optimierung und das Durchsuchen von gewichteten endlichen Transduktoren. Damit lässt es sich insbesondere auch für die Arbeit mit gewichteten endlichen Automaten einsetzen.

Aufgaben. Der Student soll folgende Teilaufgaben erfüllen:

1. Implementieren der Konstruktion eines endlichen Automaten, der R erkennt

   Eingabe: eine gewichtete MCFG
   Ausgabe: ein endlicher Automat, der R erkennt

2. Implementieren der Konstruktion eines gewichteten endlichen Automaten, der $h^{-1}(w)$ erkennt

[1] `https://github.com/tud-fop/rustomata`
[2] `http://www.openfst.org/`

Eingabe: der gewichtete Homomorphismus h und ein Wort w
Ausgabe: ein gewichteter Automat, der $h^{-1}(w)$ erkennt

3. Implementieren des Hadamardproduktes mithilfe von OpenFST

   Eingabe: zwei gewichtete endliche Automaten A_1 und A_2
   Ausgabe: ein endlicher Automat, der $L(A_1) \odot L(A_2)$ erkennt

4. Implementieren eines Entscheiders für Membership in einer multiple Dycksprache

   Eingabe: eine Bijektion φ zwischen der Menge von öffnenden und der Menge von schließenden Klammersymbolen, eine Partition P der Menge der öffnenden Klammersymbole und ein Wort u über den Klammersymbolen
   Ausgabe: „true“, wenn u ein Element der von φ und P induzierten multiplen Dycksprache ist; sonst „false“

5. Implementieren der Übersetzung eines Klammerwortes aus $mD \cap R$ in einen Parsebaum

   Eingabe: ein Element $u \in mD \cap R$
   Ausgabe: die dem Element u entsprechende Ableitung d

6. Mithilfe von 1–5 und OpenFST: Implementieren eines k-best CS-Parsers für gewichtete MCFG

   Eingabe: eine gewichtete MCFG G, eine für die Gewichtsalgebra von G geeignete partielle Ordnung $\leq$, eine Zahl $k \in \mathbb{N}_+$ und ein Wort w
   Ausgabe: eine Sequenz der Länge k von besten (bezüglich $\leq$) Parses von w in G

7. Optimierung des in 6 entwickelten Parsers bezüglich Laufzeit (z.B. durch Pruning; durch Optimierung der in 1, 2 und 3 konstruierten Automaten, der Klammersprache, und des in 4 entwickelten Entscheiders; und durch Approximation der Klammersprache)

8. Evaluation des entwickelten Parsers: Vergleich mit den state-of-the-art-Parsern für LCFRS/MCFG (rparse[3] und Grammatical Framework[4])

   (a) Gewinnung einer gewichteten MCFG aus einem Teil des NeGra-Korpus mithilfe von Vanda[5]
   (b) Bestimmung von Laufzeit und Bleu-Score der drei zu vergleichenden Parser mit der in (a) gewonnenen gewichteten MCFG auf einem Testkorpus
   (c) Wiederholung des Experiments für verschiedene Parameterbelegungen (z.B. die verwendete Approximationsstrategie) des im Rahmen der Arbeit entwickelten Parsers

Die Implementierung soll im Rahmen von rustomata durchgeführt werden.

[3] `https://github.com/wmaier/rparse`
[4] `https://www.grammaticalframework.org/`
[5] `https://www.inf.tu-dresden.de/index.php?node_id=2550`

Form. Die Arbeit muss den üblichen Standards wie folgt genügen. Die Arbeit muss in sich abgeschlossen sein und alle nötigen Definitionen und Referenzen enthalten. Die Urheberschaft von Inhalten – auch die eigene – muss klar erkennbar sein. Fremde Inhalte, z.B. Algorithmen, Konstruktionen, Definitionen, Ideen, etc., müssen durch genaue Verweise auf die entsprechende Literatur kenntlich gemacht werden. Lange wörtliche Zitate sollen vermieden werden. Gegebenenfalls muss erläutert werden, inwieweit und zu welchem Zweck fremde Inhalte modifiziert wurden. Die Struktur der Arbeit muss klar erkenntlich sein, und der Leser soll gut durch die Arbeit geführt werden. Die Darstellung aller Begriffe und Verfahren soll mathematisch formal fundiert sein. Für jeden wichtigen Begriff sollen Erläuterungen und Beispiele angegeben werden, ebenso für die Abläufe der beschriebenen Verfahren. Wo es angemessen ist, sollen Illustrationen die Darstellung vervollständigen. Bei Diagrammen, die Phänomene von Experimenten beschreiben, muss deutlich erläutert werden, welche Werte auf den einzelnen Achsen aufgetragen sind, und beschrieben werden, welche Abhängigkeit unter den Werten der verschiedenen Achsen dargestellt ist.

Für die Implementierung soll eine ausführliche Dokumentation erfolgen, die sich angemessen auf den Quelltext und die schriftliche Ausarbeitung verteilt. Dabei muss die Funktionsfähigkeit des Programms glaubhaft gemacht und durch geeignete Beispielläufe dokumentiert werden. Die durchgeführten Experimente sollen angemessen dokumentiert werden, sodass deren Reproduzierbarkeit gewährleistet ist.

Dresden, 1. August 2017

Unterschrift von Heiko Vogler

Unterschrift von Thomas Ruprecht

Literatur

[Büc+10] Matthias Büchse, Daniel Geisler, Torsten Stüber und Heiko Vogler. „n-Best Parsing Revisited“. In: *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*. Hrsg. von Frank Drewes und Marco Kuhlmann. Uppsala, Sweden: Association for Computational Linguistics, Juli 2010, S. 46–54. URL: http://www.aclweb.org/anthology/W10-2506.

[Den15] Tobias Denkinger. „A Chomsky-Schützenberger representation for weighted multiple context-free languages“. In: *Proceedings of the 12th International Conference on Finite-State Methods and Natural Language Processing (FSMNLP 2015)*. 2015. URL: http://aclanthology.info/papers/a-chomsky-schutzenberger-representation-for-weighted-multiple-context-free-languages.

[Den17] Tobias Denkinger. „Chomsky-Schützenberger parsing for weighted multiple context-free languages“. In: *Journal of Language Modelling (JLM)* 5.1 (Juli 2017), S. 3–55. DOI: 10.15398/jlm.v5i1.159.

[Eva11] Kilian Evang. „Parsing discontinuous constituents in English“. Masterarbeit. Universität Tübingen, 5. Jan. 2011. URL: http://kilian.evang.name/publications/mathesis.pdf.

[Goo99] Joshua Goodman. „Semiring Parsing“. In: *Computational Linguistics* 25 (4 Dez. 1999), S. 573–605. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=973226.973230.

[HC05] Liang Huang und David Chiang. „Better k-best Parsing“. In: *Proceedings of the Ninth International Workshop on Parsing Technology*. Parsing '05. Vancouver, British Columbia, Canada: Association for Computational Linguistics, 2005, S. 53–64. URL: http://dl.acm.org/citation.cfm?id=1654494.1654500.

[Mic01a] Jens Michaelis. „Derivational Minimalism Is Mildly Context-Sensitive“. English. In: *Logical Aspects of Computational Linguistics*. Hrsg. von Michael Moortgat. Bd. 2014. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 179–198. ISBN: 978-3-540-42251-8. DOI: 10.1007/3-540-45738-0_11.

[Mic01b] Jens Michaelis. „Transforming Linear Context-Free Rewriting Systems into Minimalist Grammars“. English. In: *Logical Aspects of Computational Linguistics*. Hrsg. von Philippe Groote, Glyn Morrill und Christian Retoré. Bd. 2099. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 228–244. ISBN: 978-3-540-42273-0. DOI: 10.1007/3-540-48199-0_14.

[Ned03] Mark-Jan Nederhof. „Weighted deductive parsing and Knuth's algorithm“. In: *Computational Linguistics* 29.1 (März 2003), S. 135–143. ISSN: 0891-2017. DOI: 10.1162/089120103321337467.

[Pol84] Carl Jesse Pollard. „Generalized phrase structure grammars, head grammars, and natural language“. Diss. Stanford University, 1984, S. 248. URL: http://searchworks.stanford.edu/view/1095753.

[Sek+91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii und Tadao Kasami. „On multiple context-free grammars“. In: *Theoretical Computer Science* 88.2 (1991), S. 191–229. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90374-B.

[Vij88] Krishnamurti Vijay-Shanker. „A study of tree adjoining grammars“. Diss. University of Pennsylvania, 1988. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.401.1695&rep=rep1&type=pdf.

[VWJ86] Krishnamurti Vijay-Shanker, David Jeremy Weir und Aravind K. Joshi. „Tree adjoining and head wrapping“. In: *Proceedings of the 11th Conference on Computational linguistics*. Association for Computational Linguistics. 1986, S. 202–207. DOI: 10.3115/991365.991425.

[WJ88] David Jeremy Weir und Aravind K. Joshi. „Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems“. In: *Proceedings of the 26th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 1988, S. 278–285. DOI: 10.3115/982023.982057.

Contents

Chapter 1

Introduction

The *Chomsky-Schützenberger theorem for context-free languages* [CS63, Proposition 2] proposes a regular language R, a homomorphism h and a Dyck language D for each context-free language L such that $L = h(R \cap D)$.

Chomsky-Schützenberger parsing for weighted and unweighted context-free languages was first introduced by Hulden [Hul09]. The approach [Hul09, section 3, page 154] uses the constructions of the Chomsky-Schützenberger characterization for context-free languages to process three steps:

1. the construction of the language $R^w = h^{-1}(w)$,
2. the construction of the language $R^{local} = R \cap R^w$, and
3. the enumeration of elements in R^{local} that are well bracketed (i.e. are also in D).

The first and second steps are realized with the construction of finite state automata that recognize the language R and R^w, respectively. Both automata are intersected in the second step. Finally, using the intersection automaton that recognizes R^{local}, we enumerate all runs that recognize a well-bracketed word [Hul09, Algorithm 1]. Each of those words is an encoding of an abstract syntax tree.

Multiple context-free languages are a generalization of context-free languages. In contrast to context-free languages, they are able to characterize discontinuous dependencies. Specifically in natural languages, we use multiple context-free grammars for language modeling.

Yoshinaka, Kaji, and Seki [YKS10] introduced a *Chomsky-Schützenberger characterization for unweighted multiple context-free languages* which Denkinger [Den16a] generalized for *weighted multiple context-free languages* with weights over *complete commutative strong bimonoids*. Very similar to the characterization for context-free languages, it proposes a regular language R, an alphabetic string homomorphism h and a multiple Dyck language D such that $L = h(R \cap D)$ for each multiple context-free language L. Furthermore, Denkinger [Den17b] generalized the approach by Hulden [Hul09] outlined above for the use with weighted multiple context-free grammars. It includes the same sequence of steps with slightly different constructions for R and R^w due to the constructions for the Chomsky-Schützenberger characterization. Moreover, the third step will check for the membership in the multiple Dyck language D instead of a Dyck language.

In the weighted case, h is a weighted alphabetic string homomorphism and the constructed language R^{local} is weighted according to h.

Denkinger [Den17b] pointed out that there are some restrictions in the case of weighted multiple context-free languages that concern practical problems [Den17b, Section 5.3: problems and restrictions].

The definition of the weighted alphabetic string homomorphism h as described by Denkinger [Den16a, Definition 4.2 and Theorem 4.10], implies the existence of cycles with the combined weight of 1 within the transitions of the weighted finite state automaton that is constructed to recognize R^{local}. This is a problem in the third step of the parsing approach since it introduces the possibility of infinite words in the support of R^{local} that have a lower weight according to h than the word with the lowest weight in $\text{supp}(R^{local}) \cap D$. Denkinger [Den17b, Definition 5.9 and Definition 5.14] solves this problem with the introduction of *factorizable weights* and *restricted grammars*. Intuitively, we restrict ourselves to only use proper weighted grammars and split the weights in h such that it is not possible to construct these *harmful loops* for the weighted finite state automaton. With this alternative definition of h, the weight of the words in $\text{supp}(R^{local}) \cap D$ according to R^{local} will not change.

In this thesis, we will introduce and evaluate an implementation of the *k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars* as described by Denkinger [Den17b]. We approach this topic step-by-step by first describing the approach for the Chomsky-Schützenberger parsing by Denkinger [Den17b] in great detail. We will then introduce a few optimizations that were essential for the implementation in real-world applications. Among these are an alternative, context-free definition of the language R, a different definition of the language R^w, and heuristic functions that help us to find the best words in R^{local}. After that, we will describe the implementation itself as a part of the *Rustomata* framework that was developed by Denkinger [Den17c]. And finally, we will describe and discuss an evaluation of the implementation using grammars and test corpora extracted from the *NeGra* corpus [SUBK98].

Chapter 2

Preliminaries

An *alphabet* is a finite set. Let Σ be an alphabet. We define an implicit set $\overline{\Sigma}$ of symbols that are distinct to Σ, i.e. $\Sigma \cap \overline{\Sigma} = \varnothing$, such that there is a bijection $\overline{(\cdot)}\colon \Sigma \to \overline{\Sigma}$. In this context, Σ usually is a set of opening brackets, and $\overline{\Sigma}$ the set of the matching closing brackets.

Let $n \in \mathbb{N}$. We denote the finite set of *sequences with at most n* symbols in Σ by $\Sigma^{\leq n} = \Sigma^0 \cup \ldots \cup \Sigma^n$. Moreover, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

Let $A_1, \ldots, A_k$ be some sets for some $k \in \mathbb{N}$. In the following, we will denote a function $f\colon A_1 \to \big(A_2 \to \ldots \to \big(A_{k-1} \to A_k\big) \ldots\big)$ just by $f\colon A_1 \to A_2 \to \ldots \to A_{k-1} \to A_k$, i.e. "$\to$" is right-associative.

Let X and Y be sets and f a function from X to Y. For each $y \in Y$, $f^{-1}(y) = \{x \in X \mid f(x) = y\}$ is the *preimage of y in f*.

Let A be an arbitrary set, $k \in \mathbb{N}$ and $a, a_1, \ldots, a_k \in A$. For the sequence $a_1 \ldots a_k$, we will abbreviate $a \in \{a_1, \ldots, a_k\}$ by $a \in a_1 \ldots a_k$. We call a set $\mathfrak{P} \subseteq \mathcal{P}(A)$ a *partition of A* if $\varnothing \notin \mathfrak{P}$, $\bigcup_{\mathfrak{p} \in \mathfrak{P}} \mathfrak{p} = A$, and $\mathfrak{p} \cap \mathfrak{p}' \neq \varnothing \iff \mathfrak{p} = \mathfrak{p}'$ for each $\mathfrak{p}, \mathfrak{p}' \in \mathfrak{P}$. Moreover, we call each element in $\mathfrak{P}$ a *partition cell in* $\mathfrak{P}$ and denote the partition cell that contains an element $a \in A$ by $\mathfrak{P}[a]$.

Let W be a set and $\trianglelefteq$ a total order on W. For each $W' \subseteq W$, we define the *minimum over* $\trianglelefteq$ *in* W' by $\min_{\trianglelefteq} W' = w$ for an arbitrary $w \in W'$ such that there is no $w' \in W'$ with $w \not\trianglelefteq w'$. Similarly, we define the *maximum over* $\trianglelefteq$ *in* W' by $\max_{\trianglelefteq} W' = \min_{\trianglerighteq} W'$. Moreover, let $C\colon A \to W$. We denote the *(infinite) ordered sequence of elements* $c_1, c_2, \ldots \in \operatorname{supp}(C)$ by $\operatorname{ordered}_{\trianglelefteq}(C) = c_1 c_2 \ldots$ such that $C(c_i) \trianglelefteq C(c_j) \iff i < j$ for all $i, j \in \mathbb{N}$. Furthermore, we define the three functions:

- $\operatorname{map}\colon (A \to B) \to A^* \to B^*$, for each set B where, for each $f \in A \to B, a \in A$ and $v \in A^*$, $\operatorname{map}(f)(av) = f(a) \cdot \operatorname{map}(f)(v)$ and $\operatorname{map}(f)(\varepsilon) = \varepsilon$,
- $\operatorname{filter}\colon \mathcal{P}(A) \to A^* \to A^*$ where , for each $A' \subseteq A, a \in A$ and $v \in A^*$,

$$\operatorname{filter}(A')(av) = \begin{cases} a \cdot \operatorname{filter}(A')(v) & \text{if } a \in A' \\ \operatorname{filter}(A')(v) & \text{otherwise} \end{cases}$$

  and $\operatorname{filter}(A')(\varepsilon) = \varepsilon$, and
- $\operatorname{take}\colon \mathbb{N} \to A^* \to A^*$ where, for each $k \in \mathbb{N}, a \in A$ and $v \in A^*$,

$$\operatorname{take}(k)(av) = \begin{cases} a \cdot \operatorname{take}(k-1)(v) & \text{if } k > 0 \\ \varepsilon & \text{otherwise.} \end{cases}$$

and $\mathrm{take}(k)(\varepsilon) = \varepsilon$.

Let $r, r' \subseteq A \times A$ be binary relations over A. For each $a \in A$, we denote the set $\{a' \in A \mid (a, a') \in r\}$ by $r(a)$ and the *composition of r and r'* as $r \circ r' = \{(a, a') \mid (a, \hat{a}) \in r, (\hat{a}, a') \in r'\}$. We denote the *identity relation* $\{(a, a) \mid a \in A\}$ by id_A.

Let S be a set. An *S-sorted set* is a tuple $(A, sort)$ of the set A and a function $sort\colon A \to S$. We denote the preimage $sort^{-1}(s) \subseteq A$ for each $s \in S$ by A_s. Let $(A, sort)$ be an $(S^* \times S)$-sorted set for some set S. We define the *S-sorted set of trees over* $(A, sort)$ as the tuple $\big(\mathcal{S}^{(A,sort)}, \overline{sort}\big)$ such that

$$\mathcal{S}^{(A,sort)} = \{a(t_1, \ldots, t_k) \mid k \in \mathbb{N}, s, s_1, \ldots, s_k \in S, a \in A_{(s_1 \ldots s_k, s)}, \forall i \in [k]\colon t_i \in \mathcal{S}^{(A,sort)}_{s_i}\}$$

and, for each $k \in \mathbb{N}$, $a(t_1, \ldots, t_k) \in \mathcal{S}^{(a,sort)}$ and $s, s_1, \ldots, s_k \in S$ such that $sort(a) = (s_1 \ldots s_k, s)$, $\overline{sort}\big(a(t_1, \ldots, t_k)\big) = s$.

We denote the set of all *unranked trees over A* by $\mathcal{U}_A = \{a(u_1, \ldots, u_k) \mid k \in \mathbb{N}, a \in A, u_1, \ldots, u_k \in \mathcal{U}_A\}$ and, for each $k \in \mathbb{N}$, the set of all *positions in a tree* $a(u_1, \ldots, u_k) \in \mathcal{U}_A$ by $\mathrm{pos}(a(u_1, \ldots, u_k)) = \{\varepsilon\} \cup \bigcup_{i \in [k]} \{i\} \cdot \mathrm{pos}(u_i)$. Furthermore, let $k \in \mathbb{N}$, $u = a(u_1, \ldots, u_k) \in \mathcal{U}_A$ and $\alpha \in \mathrm{pos}(u)$, we recursively define

- the *label* of u at α,

$$u(\alpha) = \begin{cases} a & \text{if } \alpha = \varepsilon \\ u_i(\alpha') & \text{if there are } \alpha' \in \mathbb{N}^*, i \in [k] \text{ such that } \alpha = i\alpha', \end{cases}$$

- the *substitution of the label* of u at α with some $a' \in A$,

$$u(\alpha/a) = \begin{cases} a'(u_1, \ldots, u_k) & \text{if } \alpha = \varepsilon \\ a(u_1, \ldots, u_i(\alpha'/a'), \ldots, u_k) & \text{if there are } \alpha' \in \mathbb{N}^*, i \in [k] \text{ such that } \alpha = i\alpha' \end{cases}$$

- the *number of children* in u at α,

$$\mathrm{children}(u, \alpha) = \begin{cases} k & \text{if } \alpha = \varepsilon \\ \mathrm{children}(u_i, a') & \text{if there are } \alpha' \in \mathbb{N}^*, i \in [k] \text{ such that } \alpha = i\alpha' \end{cases},$$

- the *subtree* of u at α,

$$u[\alpha] = \begin{cases} u & \text{if } \alpha = \varepsilon \\ u_i[\alpha'] & \text{if there are } \alpha' \in \mathbb{N}^*, i \in [k] \text{ such that } \alpha = i\alpha' \end{cases},$$

- and *inserting a sequence of subtrees* $u'_1, \ldots, u'_{k'} \in \mathcal{U}_A$,for some $k' \in \mathbb{N}$ as children of u at α,

$$\mathrm{push}(u, \alpha, u'_1, \ldots, u'_{k'}) = \begin{cases} a(u_1, \ldots, u_k, u'_1, \ldots, u'_{k'}) & \text{if } \alpha = \varepsilon \\ a(u_1, \ldots, \mathrm{push}(u_i, \alpha', u'_1, \ldots, u'_{k'}), \ldots, u_k) & \text{if there are } \alpha' \in \mathbb{N}^*, \\ & i \in [k] \text{ such that } \alpha = \\ & i\alpha' \end{cases}$$

A *bimonoid* is a tuple $(W, +, \cdot, 0, 1)$ such that $(W, +, 0)$ and $(W, \cdot, 1)$ are monoids over the same carrier set W. $(W, +, \cdot, 0, 1)$ is a *strong bimonoid* if $+$ is commutative and 0 is an annihilating element for $\cdot$, i.e.

$w + w' = w' + w$ and $w \cdot 0 = 0 \cdot w = 0$ for each $w, w' \in W$. We call $(W, +, \cdot, 0, 1)$ *complete* if the results of the operations

- $\sum_{w \in W'} w$, and
- $\prod_{w \in W'} w$

for each $W' \subseteq W$ are in W. Lastly, let $\trianglelefteq$ be a partial order on W. We call $(W, +, \cdot, 0, 1)$ *factorizable with respect to* $\trianglelefteq$ if, for each $n \in \mathbb{N}$ and $w \in W$, there are $w_1, \ldots, w_n \in W$ such that $w = w_1 \cdot \ldots \cdot w_n$, and $w_i \trianglelefteq w$ for each $i \in [n]$. In this context, we call the sequence $w_1 \ldots w_n$ a $(\trianglelefteq, n)$*-factorization of* w. [1]

2.1 Weighted string homomorphism

Definition 1 (monomial)**.** Let $(W, +, \cdot, 0, 1)$ be a bimonoid and A a set. We call a function $f \colon A \to W$ a *monomial* if there is at most one $a \in A$ such that $f(a) \neq 0$. ■

Definition 2 (alphabetic weighted string homomorphism)**.** Let Σ and Γ be alphabets and $(W, +, \cdot, 0, 1)$ a bimonoid. We call the function $h \colon \Sigma^* \to \Gamma^* \to W$ an *alphabetic* W*-weighted string homomorphism* if there is a function $h' \colon \Sigma \to \Gamma \cup \{\varepsilon\} \to W$ such that

$$h(\sigma_1 \ldots \sigma_\ell)(v) = \sum_{\substack{\gamma_1, \ldots, \gamma_\ell \in \Gamma \cup \{\varepsilon\} \colon \\ v = \gamma_1 \ldots \gamma_\ell}} h'(\sigma_1)(\gamma_1) \cdot \ldots \cdot h'(\sigma_\ell)(\gamma_\ell)$$

for each $\ell \in \mathbb{N}, \sigma_1, \ldots, \sigma_\ell \in \Sigma$ and $v \in \Gamma^*$, and $h'(\sigma)$ is a monomial for each $\sigma \in \Sigma$. ■

Let $(W, +, \cdot, 0, 1)$ be a bimonoid, A and B be sets, and $f \colon A \to B \to W$ a function such that $f(a)$ is a monomial for each $a \in A$. We will use $f_{\mathbb{B}}$ to denote a partial function from A to B such that $f_{\mathbb{B}}(a) = b$ iff $f(a)(b) \neq 0$ for each $a \in A$ and $b \in B$. We use the following definition of the domain of word in a homomorphism by Denkinger [Den17b, Definition 5.5, Lemma 5.6].

Definition 3 (domain language)**.** Let $h \colon \Sigma^* \to \Gamma^* \to W$ be a alphabetic W-weighted string homomorphism, $\gamma_1, \ldots, \gamma_n \in \Gamma$ for some $n \in \mathbb{N}$, and let $\Sigma_\gamma = \{\sigma \in \Sigma \mid h_{\mathbb{B}}(\sigma) = \gamma\}$ for each $\gamma \in \Gamma \cup \{\varepsilon\}$.

We define the *domain language of* $\gamma_1 \ldots \gamma_n$ *in* h as the regular language

$$R_{h, \gamma_1 \ldots \gamma_n} = \Sigma_\varepsilon^* \cdot \Sigma_{\gamma_1} \cdot \Sigma_\varepsilon^* \cdot \ldots \cdot \Sigma_{\gamma_n} \cdot \Sigma_\varepsilon^*.$$ ■

2.2 Automata with storage

In this section, we will define *finite state automata* and *push-down automata* as instances of *automata with storage*. Our definitions of automata with storage are based on the definition by Denkinger [Den17a, Definition 6].

Definition 4 (data storages)**.** Let C be a set. We call the tuple (C, R, C_F, c_{init}) a *data storage (over* C*)* if

- $R \subseteq \mathcal{P}(C \times C)$ is a set of binary relations over C (*instructions*),
- $C_F \subseteq C$ (*accepting storage configurations*)

[1] Denkinger [Den17b, Table 3 in Section 5] gave a broad overview for examples of factorizable bimonoids.

- $c_{init} \in C$ (*initial storage configuration*). ■

Definition 5 (automata with storage)**.** Let $S = (C, R, C_F, c_{init})$ be a data storage over some set C. An S-automaton is a tuple $(Q, \Sigma, q_{init}, Q_F, T)$ where

- Q is a finite set (*states*),
- Σ is an alphabet,
- $q_{init} \in Q$ (*initial state*),
- $Q_F \subseteq Q$ (*final states*), and
- $T \subseteq Q \times \Sigma \times R \times Q$ is a finite set (*transitions*). ■

Definition 6 (deterministic automata)**.** Let $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ be an S-automaton for some data storage S over some set C. We call $\mathcal{M}$ *deterministic* if, for each $\sigma \in \Sigma$ and $q \in Q$, there is at most one transition of the form $(q, \sigma, r, q') \in T$, and, for each transition $(q, \sigma, r, q') \in T$ and each $c \in C$, $|r(c)| \leq 1$. ■

Definition 7 (runs and language of an automaton)**.** Let $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ be an S-automaton for some data storage $S = (C, R, C_F, c_{init})$ over some set C. We call the string $q_0 c_0 \sigma_1 q_1 c_1 \dots \sigma_n q_n c_n$ a *run for $\sigma_1 \dots \sigma_n$ in $\mathcal{M}$ starting with q_0 and c_0* if, for each $i \in [n]$ there is a transition $(q_{i-1}, \sigma_i, r_i, q_i) \in T$ such that $c_i \in r_i(c_{i-1})$; for each $n \in \mathbb{N}$, $q_0, \dots, q_{n-1} \in Q, q_n \in Q_F, c_0, \dots, c_{n-1} \in C, c_n \in C_F$, and $\sigma_1, \dots, \sigma_n \in \Sigma$. For each $w \in \Sigma^*$, we denote the *set of all runs for w in $\mathcal{M}$ starting with some $q_0 \in Q$ and $c_0 \in C$* by $\mathrm{Runs}(\mathcal{M}, q_0, c_0)(w)$. Furthermore, we use the following abbreviations:

- the *set of all runs for $w \in \Sigma^*$ in $\mathcal{M}$*: $\mathrm{Runs}(\mathcal{M})(w) = \mathrm{Runs}(\mathcal{M}, q_{init}, c_{init})(w)$,
- the *set of all runs in $\mathcal{M}$*: $\mathrm{Runs}(\mathcal{M}) = \bigcup_{w \in \Sigma^*} \mathrm{Runs}(\mathcal{M}, q_{init}, c_{init})(w)$, and
- the *set of all runs in $\mathcal{M}$ starting with $q_0 \in Q$ and $c_0 \in C$*:

$$\mathrm{Runs}(\mathcal{M}, q_0, c_0) = \bigcup_{w \in \Sigma^*} \mathrm{Runs}(\mathcal{M}, q_0, c_0)(w).$$

The *language of $\mathcal{M}$* is the set $\mathrm{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \mathrm{Runs}(\mathcal{M})(w) \neq \varnothing\}$. For each $w \in \Sigma^*$, if $w \in \mathrm{L}(\mathcal{M})$, we say $\mathcal{M}$ recognizes w. ■

There is a difference in our definition of runs in an automaton to the definition by Denkinger [Den17a] which also affects the definition of *weighted automata with storage*. In particular, Denkinger [Den17a, Section 3.2, page 93] defines runs as sequence of transitions that do not directly include any storage configuration. We chose to deviate because this definition is more consistent with non-determinism when we compare the state-transitions and the storage-transitions. Since we include the storage configurations in the runs but not the instructions, our set of runs is different if an instruction yields multiple configurations or multiple instructions lead to the same configuration.

Lemma 8. *Let $\mathcal{M} = (Q, \Sigma, q, Q_F, T)$ be an S-automaton for some storage S over some set C. If $r = q_0 c_0 \sigma_1 q_1 c_1 \dots \sigma_n q_n c_n$ is a run in $\mathcal{M}$ starting with q_0 and c_0 for some $n \in \mathbb{N}, q_0, \dots, q_n \in Q$, $c_0, \dots, c_n \in C$ and $\sigma_1, \dots, \sigma_n \in \Sigma$, and $n > 1$, then $q_1 c_1 \dots \sigma_n q_n c_n \in \mathrm{Runs}(\mathcal{M}, q_1, c_1)$ is a run in $\mathcal{M}$ starting with q_1 and c_1.*

Proof. This follows directly from Definition 7. □

Definition 9 (weighted automata)**.** Let $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ be an S-automaton for some data storage S over a set C, $(W, +, \cdot, 0, 1)$ a strong and complete bimonoid and $\mu : T \to (W \setminus \{0\})$. We call $(\mathcal{M}, \mu)$ a *W-weighted S-automaton.*

Let $n \in \mathbb{N}, q_0, \ldots, q_n \in Q, \sigma_1, \ldots, \sigma_n \in \Sigma$, and $c_0, \ldots, c_n \in C$ such that $q_0c_0\sigma_1q_1c_1 \ldots \sigma_nq_nc_n \in \mathrm{Runs}(\mathcal{M})$. The *weight of a run* is the product of the used transitions

$$\mu(q_0c_0\sigma_1q_1c_1 \ldots \sigma_nq_nc_n) = \prod_{i \in [n]} \mu(q_{i-1}, \sigma_i, r_i, q_i)$$

if $(q_{i-1}, \sigma_i, r_i, q_i) \in T$ and $c_i \in r_i(c_{i-1})$ for each $i \in [n]$. The *weight of a word* $w \in \Sigma^*$ is the sum of weights over all runs for w in $\mathcal{M}$, i.e.

$$\mu(w) = \sum_{r \in \mathrm{Runs}(\mathcal{M})(w)} \mu(r). \qquad \blacksquare$$

Corollary 10. *Let $(\mathcal{M}, \mu)$ a W-weighted S-automaton for some storage S over some set C where $\mathcal{M} = (Q, \Sigma, q, Q_F, T)$. If $r = q_0c_0\sigma_1q_1c_1 \ldots \sigma_nq_nc_n$ is a run in $\mathcal{M}$ starting with q_0 and c_0 for some $n \in \mathbb{N}, q_0, \ldots, q_n \in Q$, $c_0, \ldots, c_n \in C$ and $\sigma_1, \ldots, \sigma_n \in \Sigma$, and $n > 1$ then there is a transition $(q_0, \sigma_1, r_1, q_1) \in T$ and a run $r' \in \mathrm{Runs}(\mathcal{M}, q_1, c_1)$ such that $c_1 \in r_1(c_0)$ and*

$$\mu(r) = \mu(\tau) \cdot \mu(r').$$

Proof. This follows directly from Lemma 8 and Definition 9. □

Corollary 11. *Let $(W, +, \cdot, 0, 1)$ be a strong and complete bimonoid, and $\trianglelefteq$ a partial order on W such that $w \trianglelefteq w \cdot w'$ for each $w, w' \in W$. Furthermore, let $\mathcal{M} = (Q, \Sigma, q, Q_F, T)$ be an S-automaton for some storage S over some set C and $\mu \colon T \to W$ a weight assignment such that $(\mathcal{M}, \mu)$ is a W-weighted S-automaton.*

If $r = q_0c_0\sigma_1q_1c_1 \ldots \sigma_nq_nc_n$ is a run in $\mathcal{M}$ starting with q_0 and c_0 for some $n \in \mathbb{N}, q_0, \ldots, q_n \in Q$, $c_0, \ldots, c_n \in C$ and $\sigma_1, \ldots, \sigma_n \in \Sigma$, then there is a transition $(q_0, \sigma_1, r_1, q_1) \in T$ such that $c_1 \in r_1(c_0)$ and

$$\mu(r) \trianglerighteq \mu(q_0, \sigma_1, r_1, q_1) \cdot \min_{\trianglelefteq}\{\mu(r') \mid r' \in \mathrm{Runs}(\mathcal{M}, q_1, c_1)\} \trianglerighteq \min_{\trianglelefteq}\{\mu(r') \mid r' \in \mathrm{Runs}(\mathcal{M}, q_0, c_0)\}.$$

Proof. By Lemma 8 and Corollary 10, the run r can be decomposed into a transition $\tau = (q_0, \sigma_1, r_1, q_1) \in T$ and a run $r' = q_1c_1 \ldots \sigma_nq_nc_n \in \mathrm{Runs}(\mathcal{M}, q_1, c_1)$ such that the weight of the run $\mu(r)$ is decomposed into $\mu(\tau) \cdot \mu(r')$. Since $\trianglelefteq$ is a partial order, the minimum over $\trianglelefteq$ in the set of weights of the runs in $\mathrm{Runs}(\mathcal{M}, q_1, c_1)$ is at least $\mu(r')$, thus $\mu(\tau) \cdot \mu(r') \trianglerighteq \mu(\tau) \cdot \min_{\trianglelefteq}\{\mu(r') \mid r' \in \mathrm{Runs}(\mathcal{M}, q_1, c_1)\}$. The same reason holds for the second part. Since $r \in \mathrm{Runs}(\mathcal{M}, q_0, c_0)$, the weight of r is at least the minimum of the weight of all runs in $\mathrm{Runs}(\mathcal{M}, q_0, c_0)$. □

2.2.1 Finite state automata

Definition 12 (finite state automata)**.** Let $\mathrm{S}_{\mathrm{finite}} = (\{\varnothing\}, \{\mathrm{id}_{\{\varnothing\}}\}, \{\varnothing\}, \varnothing)$ be a storage. We call each $\mathrm{S}_{\mathrm{finite}}$-automaton a *finite state automaton* (*FSA*). ■

Let $\mathcal{A} = (Q, \Sigma, q_{init}, Q_F, T)$ be a finite state automaton. We denote each transition $(q, \sigma, \mathrm{id}_{\{\varnothing\}}, q') \in T$ by (q, σ, q') and each run $q_{init}\varnothing\sigma_1 q_1 \varnothing \dots \sigma_n q_n \varnothing \in \mathrm{Runs}(\mathcal{A})$ by $q_0\sigma_1 q_1 \dots \sigma_n q_n$ for $q_0, \dots, q_n \in Q$ and $\sigma_1, \dots, \sigma_n \in \Sigma$.

Definition 13 (intersection)**.** Let $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ be an S-automaton for some data storage S over a set C and $\mathcal{A} = (Q', \Sigma, q'_{init}, Q'_F, T')$ be an FSA. We call the S-automaton

$$\mathcal{M} \times \mathcal{A} = (Q \times Q', \Sigma, (q_{init}, q'_{init}), Q_F \times Q'_F, \bar{T})$$

where

$$\bar{T} = \{((q_1, q_2), \sigma, r, (q'_1, q'_2)) \mid \sigma \in \Sigma, (q_1, \sigma, r, q'_1) \in T', (q_2, \sigma, q'_2) \in T'\}$$

the *intersection of $\mathcal{M}$ and $\mathcal{A}$.* ■

Theorem 14. *Let $\mathcal{M}$ be an S-automaton for some data storage S and $\mathcal{A}$ be an FSA. The language of the intersection $\mathcal{M} \times \mathcal{A}$ is the intersection of the languages of $\mathcal{M}$ and $\mathcal{A}$, i.e.* $\mathrm{L}(\mathcal{M} \times \mathcal{A}) = \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\mathcal{A})$.

Proof. Let $S = (C, R, C_F, c_{init})$ be a storage over some set C, $\mathcal{M} = (Q, \Sigma, q_0, Q_F, T)$ an S-automaton, and $\mathcal{A} = (Q', \Sigma, q'_0, Q'_F, T')$ an fsa. By Definition 13, $\mathcal{M} \times \mathcal{A} = (Q \times Q', \Sigma, (q_0, q'_0), Q_F \times Q'_F, \bar{T})$, where $\bar{T} = \{((q_1, q'_1), \sigma, r, (q_2, q'_2) \mid \sigma \in \Sigma, (q'_1, \sigma, q'_2) \in T', (q_1, \sigma, r, q_2) \in T\}$.

By Definition 7, a word $w \in \Sigma^*$ is in the language of an automaton over Σ if the set of runs for w in that automaton is not empty. We will prove that, for each $w \in \Sigma^*$, $\mathrm{Runs}(\mathcal{M} \times \mathcal{A})(w) \neq \varnothing \iff \mathrm{Runs}(\mathcal{M})(w) \neq \varnothing \wedge \mathrm{Runs}(\mathcal{A})(w) \neq \varnothing$.

Let $n \in \mathbb{N}$ and $\sigma_1, \dots, \sigma_n \in \Sigma$.

$\sigma_1 \dots \sigma_n \in \mathrm{L}(\mathcal{M} \times \mathcal{A})$

$\iff \mathrm{Runs}(\mathcal{M} \times \mathcal{A})(\sigma_1 \dots \sigma_n) \neq \varnothing$ (by Definition 7)

$\iff \exists (q_1, q'_1), \dots, (q_n, q'_n) \in Q \times Q', c_1, \dots, c_n \in C$: (by Definition 7)

$(q_0, q'_0)c_0\sigma_1(q_1, q'_1)c_1 \dots \sigma_n(q_n, q'_n)c_n \in \mathrm{Runs}(\mathcal{M} \times \mathcal{A})(\sigma_1 \dots \sigma_n)$

$\iff \exists q_1, \dots, q_n \in Q, q'_1, \dots, q'_n \in Q', c_1, \dots, c_n \in C$: (by Definition 13)

$(q_0, q'_0)c_0\sigma_1(q_1, q'_1)c_1 \dots \sigma_n(q_n, q'_n)c_n \in \mathrm{Runs}(\mathcal{M} \times \mathcal{A})(\sigma_1 \dots \sigma_n)$

$\iff \exists q_1, \dots, q_{n-1} \in Q, q_n \in Q_F, q'_1, \dots, q'_{n-1} \in Q', q'_n \in Q'_F, c_1, \dots, c_{n-1} \in C, c_n \in C_F, r_1, \dots, r_n \in R$:

$\forall i \in [n]$: $((q_{i-1}, q'_{i-1}), \sigma_i, r_i, (q_i, q'_i)) \in \bar{T} \wedge c_i \in r_i(c_{i-1})$

(by Definition 7)

$\iff \exists q_1, \dots, q_{n-1} \in Q, q_n \in Q_F, c_1, \dots, c_{n-1} \in C, c_n \in C_F, r_1, \dots, r_n \in R, q'_1, \dots, q'_{n-1} \in Q', q'_n \in Q'_F$:

$\forall i \in [n]$: $(q_{i-1}, \sigma_i, r_i, q_i) \in T \wedge c_i \in r_i(c_{i-1}) \wedge (q'_{i-1}, \sigma_i, q'_i) \in T'$

(by Definition 13)

$\iff \exists q_1, \dots, q_n \in Q, c_1, \dots, c_n \in C, r_1, \dots, r_n \in R, q'_1, \dots, q'_n \in Q'$: (by Definition 7)

$q_1 c_1 \sigma_1 q_1 c_1 \dots \sigma_n q_n c_n \in \mathrm{Runs}(\mathcal{M})(\sigma_1 \dots \sigma_n)$

$\wedge\, q'_1 \sigma_1 q'_1 \dots \sigma_n q'_n \in \mathrm{Runs}(\mathcal{A})(\sigma_1 \dots \sigma_n)$

$\iff \mathrm{Runs}(\mathcal{M})(\sigma_1 \dots \sigma_n) \neq \varnothing \wedge \mathrm{Runs}(\mathcal{A})(\sigma_1 \dots \sigma_n) \neq \varnothing$

$\iff \sigma_1 \dots \sigma_n \in \mathrm{L}(\mathcal{M}) \wedge \sigma_1 \dots \sigma_n \in \mathrm{L}(\mathcal{A})$ (by Definition 7)

$\iff \sigma_1 \dots \sigma_n \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\mathcal{A})$ □

Lemma 15. *Let $\mathcal{A} = (Q, \Sigma, q_{init}, Q_F, T)$ be an FSA. $\mathcal{A}$ is deterministic if, for every $\sigma \in \Sigma$ and $q \in Q$, there is at most one transition $(q, \sigma, \mathrm{id}_{\{\varnothing\}}, q') \in T$.*

Proof. Let $\mathcal{A} = (Q, \Sigma, q_{init}, Q_F, T)$ be an FSA, i.e. $\mathcal{A}$ is a $\mathrm{S}_{\mathrm{finite}}$-automaton. By Definition 6, $\mathcal{A}$ is deterministic if there is at most transition $(q, \sigma, \mathrm{id}_{\{\varnothing\}}, q') \in T$ for each $q \in Q$ and $\sigma \in \Sigma$, and $|\mathrm{id}_{\{\varnothing\}}(\varnothing)| \leq 1$. We only use the instruction $\mathrm{id}_{\{\varnothing\}}$ in each transition, $\varnothing$ is our only configuration, and $|\mathrm{id}_{\{\varnothing\}}(\varnothing)| = 1$. Thus, $\mathcal{A}$ is deterministic if there is at most transition $(q, \sigma, q') \in T$ for each $q \in Q$ and $\sigma \in \Sigma$. □

Theorem 16. *If $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ is a deterministic S-automaton for some data storage S and $\mathcal{A} = (Q', \Sigma, q'_{init}, Q'_F, T')$ are deterministic FSA, then the intersection $\mathcal{M} \times \mathcal{A}$ is a deterministic S-automaton.*

Proof. Let $S = (C, R, C_F, c_{init})$ be a storage over some set C, $\mathcal{M} = (Q, \Sigma, q_{init}, Q_F, T)$ be and S-automaton, and $\mathcal{A} = (Q', \Sigma, q'_{init}, Q'_F, T')$ be an fsa. By Definition 13,

$$\mathcal{M} \times \mathcal{A} = (Q \times Q', \Sigma, (q_{init}, q'_{init}), Q_F \times Q'_F, \bar{T}),$$

where

$$\bar{T} = \{((q_0, q'_0), \sigma, r, (q_1, q'_1) \mid \sigma \in \Sigma, (q'_0, \sigma, q'_1) \in T', (q_0, \sigma, r, q_1) \in T\}.$$

$\mathcal{M}$ is deterministic and $\mathcal{A}$ is deterministic

$\iff$ for each $\sigma \in \Sigma$ and $q \in Q$ there is at most one transition (by Definition 6, Lemma 15)
$(q, \sigma, r, q') \in T$ and $r(c)$ for each $c \in C$, and for each $\sigma \in \Sigma$ and
$q \in Q'$ there is at most one transition $(q, \sigma, q') \in T'$

$\iff \forall q_0 \in Q, q'_0 \in Q', \sigma \in \Sigma, c \in C$:
$|\{(q, \sigma, c', q_1) \mid (q_0, \sigma, r, q_1) \in T, c' \in r(c)\}| \leq 1 \wedge |\{(q'_0, \sigma, q'_1) \mid (q'_0, \sigma, q'_1) \in T'\}| \leq 1$

$\implies \forall q_0 \in Q, q'_0 \in Q', \sigma \in \Sigma, c \in C$:
$|\{((q_0, q'_0), \sigma, c', (q_1, q'_1)) \mid (q_0, \sigma, r, q_1) \in T, c' \in r(c), (q'_0, \sigma, q'_1) \in T'\}| \leq 1$

$\iff \forall (q_0, q'_0) \in Q \times Q', \sigma \in \Sigma, c \in C$: (by Definition 13)
$|\{((q_0, q'_0), \sigma, c', (q_1, q'_1)) \mid ((q_0, q'_0), \sigma, r, (q_1, q'_1)) \in \bar{T}, c' \in r(c)\}| \leq 1$

$\iff$ for each $q \in Q \times Q'$ and $\sigma \in \Sigma$,
there is at most one $(q, \sigma, r, q') \in \bar{T}$ and, for each $c \in C$, $|r(c)| \leq 1$

$\iff \mathcal{M} \times \mathcal{A}$ is deterministic (by Definition 6)

□

2.2.2 Push-down automata

Definition 17 (push-down automaton)**.** Let Γ be an alphabet. We define the following binary relations over Γ^*:

- $\mathrm{push}(\gamma) = \{(v, \gamma v) \mid v \in \Gamma^*\}$ for each $\gamma \in \Gamma$,

- $\text{pop}(\gamma) = \{(\gamma v, v) \mid v \in \Gamma^*\}$ for each $\gamma \in \Gamma$, and
- $\text{replace}(\gamma, \gamma') = \{(\gamma v, \gamma' v) \mid v \in \Gamma^*\}$ for each $\gamma, \gamma' \in \Gamma$.

A *push-down storage over* Γ is the data storage $\mathrm{S}^{\Gamma}_{\mathrm{pd}} = (\Gamma^*, R, \{\varepsilon\}, \varepsilon)$ where

$$R = \{\text{push}(\gamma) \mid \gamma \in \Gamma\} \cup \{\text{pop}(\gamma) \mid \gamma \in \Gamma\} \cup \{\text{replace}(\gamma, \gamma') \mid \gamma, \gamma' \in \Gamma\} \cup \{\text{id}_{\Gamma^*}\}.$$

We call every $\mathrm{S}^{\Gamma}_{\mathrm{pd}}$-automaton a *$\Gamma$-push-down automaton.* ■

Lemma 18. *Let $\mathcal{K} = (Q, \Sigma, q_{init}, Q_F, T)$ be a Γ-push-down automaton for some alphabet Γ. $\mathcal{K}$ is deterministic if, for every $q \in Q$ and $\sigma \in \Sigma$, there is at most one transition $(q, \sigma, r, q') \in T$.*

Proof. Let $\mathcal{K} = (Q, \Sigma, q_{init}, Q_F, T)$ be a Γ-push-down automaton for some alphabet Γ, i.e. $\mathcal{K}$ is a $\mathrm{S}^{\Gamma}_{\mathrm{pd}}$-automaton.

By Definition 6, $\mathcal{K}$ is deterministic if for each $q \in Q$ and $\sigma \in \Sigma$ there is at most one transition $(q, \sigma, r, q') \in T$ and $|r(v)| \leq 1$ for each $v \in \Gamma^*$. For each instruction $r \in \{\text{push}(\gamma), \text{pop}(\gamma) \mid \gamma \in \Gamma\} \cup \{\text{replace}(\gamma, \gamma') \mid \gamma, \gamma' \in \Gamma\} \cup \{\text{id}_{\Gamma^*}\}$ and each $v \in \Gamma^*$ obviously holds $|r(v)| \leq 1$. Thus, $\mathcal{K}$ is deterministic if, for every $q \in Q$ and $\sigma \in \Sigma$, there is at most one transition $(q, \sigma, r, q') \in T$. □

Let Γ be an alphabet, $\gamma_1, \ldots, \gamma_n \in \Gamma$ for some $n \in \mathbb{N}$, and $d \in \mathbb{N}$. We denote *d-prefix of* $\gamma_1 \ldots \gamma_n$, $\gamma_1 \ldots \gamma_{\min(n,d)}$, by $[\gamma_1 \ldots \gamma_n]_d$.

Definition 19 (finite approximation of push-down automata)**.** Let $\mathcal{K} = (Q, \Sigma, q_{init}, Q_F, T)$ be a Γ-push-down automaton for some alphabet Γ, and $d \in \mathbb{N}$. The d-approximation of $\mathcal{K}$ is the FSA $[\mathcal{K}]_d = (Q \times \Gamma^{\leq d}, \Sigma, (q_{init}, \varepsilon), Q_F \times \{\varepsilon\}, \bar{T})$, where $\bar{T}$ is the smallest set T' such that

1. for each transition $\big(q_{init}, \sigma, \text{id}_{\Gamma^*}, q\big) \in T$, $\big((q_{init}, \varepsilon), \sigma, (q, \varepsilon)\big)$ is a transition in T',
2. for $\gamma \in \Gamma$ and each transition $\big(q_{init}, \sigma, \text{push}(\gamma), q\big) \in T$, $\big((q_{init}, \varepsilon), \sigma, (q, [\gamma]_d)\big)$ is a transition in T',
3. for each pair of transitions $\big((\hat{q}, \hat{v}), \hat{\sigma}, (q, v)\big) \in T'$ and $\big(q, \sigma, r, q'\big) \in T$, for each $v' \in r(v)$, $\big((q, v), \sigma, (q', [v']_d)\big)$ is a transition in T',
4. for each pair of transitions $\big((\hat{q}, \hat{v}), \hat{\sigma}, (q, \varepsilon)\big) \in T'$ and $\big(q, \sigma, \text{pop}(\gamma), q'\big) \in T$ for some $\gamma \in \Gamma$, $\big((q, \varepsilon), \sigma, (q', \varepsilon)\big)$ is a transition in T', and
5. for each pair of transitions $\big((\hat{q}, \hat{v}), \hat{\sigma}, (q, \varepsilon)\big) \in T'$ and $\big(q, \sigma, \text{replace}(\gamma, \gamma'), q'\big) \in T$ for some $\gamma, \gamma' \in \Gamma$, $\big((q, \varepsilon), \sigma, (q', [\gamma']_d)\big)$ is a transition in T'. ■

Theorem 20. *If $\mathcal{K}$ is a deterministic Γ-push-down automaton for some alphabet Γ, then for each $d \in \mathbb{N}$, the d-approximation $[\mathcal{K}]_d$ is a deterministic finite state automaton.*

Proof. Let $\mathcal{K} = (Q, \Sigma, q_{init}, Q_F, T)$ be a deterministic Γ-push-down automaton for some alphabet Γ, and the push-down-storage $\mathrm{S}^{\Gamma}_{\mathrm{pd}} = (\Gamma^*, R, \{\varepsilon\}, \varepsilon)$. Thus, by Lemma 18, there is at most one transition $(q, \sigma, r, q') \in T$ for each $q \in Q$ and $\sigma \in \Sigma$. By Definition 19, we construct $\bar{T}$, the set of transitions of $[\mathcal{K}]_d$, from the set of transitions in $\mathcal{K}$ for every state $(q, v) \in (Q, \Gamma^{\leq k})$ that is either (q_{init}, ε) or appeared in a previously constructed transition. Now, we show that for every transition $(q, \sigma, r, q') \in T$ and every $(q, v) \in (Q, \Gamma^{\leq k})$, there is at most one constructed transition in the set of transitions of $[\mathcal{K}]_d$. We distinguish two cases.

- $v \neq \varepsilon$ (Definition 19, Item 3), or $v = \varepsilon$ and either $r = \mathrm{id}_{\Gamma^*}$ or $r = \mathrm{push}(\gamma)$ for some $\gamma \in \Gamma$ (Definition 19, Items 1 to 3): Since $|r(v)| \leq 1$ for every $r \in R$ (see Lemma 18), there is at most one transition $((q, v), \sigma, (q', v')) \in \bar{T}$ where $r(v) = \{v'\}$.

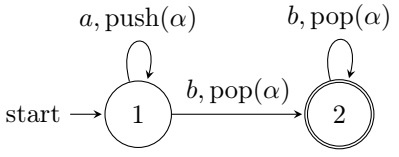
- $v = \varepsilon$ and either $r = \mathrm{pop}(\gamma)$ for some $\gamma \in \Gamma$ or $r = \mathrm{replace}(\gamma, \gamma')$ for some $\gamma, \gamma' \in \Gamma$ (Definition 19, Items 3 to 5): Obviously, both instructions can not be applied to the empty push-down configuration, i.e. $r(v) = \varnothing$. So, Item 3 will not coincide with Items 4 and 5. Both other cases, Items 4 and 5, construct exactly one transition.

Hence, there is at most one transition in $\bar{T}$ for each $(q, v) \in Q \times \Gamma^{\leq d}$ and $\sigma \in \Sigma$, and thus, $[\mathcal{K}]_d$ is deterministic by Lemma 15. □

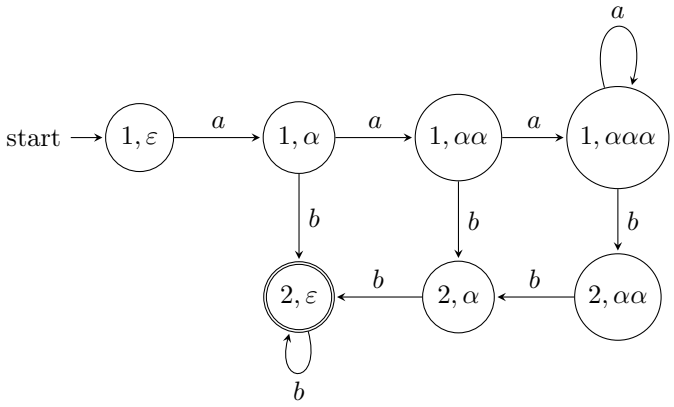
Theorem 21. *For each $d \in \mathbb{N}$, the d-approximation of a push-down automaton $\mathcal{K}$ is a* superset approximation *[Den17a, Theorem 21] with respect to the language of $\mathcal{K}$ and the language of $[\mathcal{K}]_d$. Thus $\mathrm{L}(\mathcal{K}) \subseteq \mathrm{L}([\mathcal{K}]_d)$ for each $d \in \mathbb{N}$.*

Proof. For each $d \in \mathbb{N}$ the d-approximation described above is the $A_{\mathrm{top},d}$ strategy described by Denkinger [Den17a, Example 24 (ii)]. □

Example 22. We consider the push-down automaton $\mathcal{K}$ shown in Figure 2.1a. It is obvious that it recognizes the language $\mathrm{L}(\mathcal{K}) = \{a^n b^n \mid n \in \mathbb{N} \setminus \{0\}\}$. The 3-approximation $[\mathcal{K}]_3$ is shown in Figure 2.1b. It is an FSA that accepts the language $\mathrm{L}([\mathcal{K}]_3) = \{a^n b^k \mid n, k \in \mathbb{N} \setminus \{0\}, n \leq k \vee n \geq 3 \wedge k \geq 3\}$. Thus, we found a superset approximation with $\mathrm{L}(\mathcal{K}) \subset \mathrm{L}([\mathcal{K}]_3)$.



(a) Push-down automaton $\mathcal{K}$.



(b) 3-approximation $[\mathcal{K}]_3$ of $\mathcal{K}$.

Figure 2.1: A push-down automaton and a 3-approximation of it.

■

2.3 Weighted multiple context-free grammars

Definition 23 (composition representation)**.** Let Σ be an alphabet and $X_{s_1 \dots s_k} = \{x_i^j \mid i \in [k], j \in [s_i]\}$ be a set of variables distinct from Σ for each $k, s_1, \dots, s_k \in \mathbb{N}$. We call the $(\mathbb{N}^* \times \mathbb{N})$-sorted set $C^{(\Sigma)}$ the *set of all composition representations over Σ* where

$$C^{(\Sigma)}_{s_1 \dots s_k, s} = ((\Sigma \cup X_{s_1 \dots s_k})^*)^s$$

for each $k, s, s_1, \ldots, s_k \in \mathbb{N}$. ■

Let $c \in C^{(\Sigma)}_{s_1 \ldots s_k, s}$ be a composition representation for some alphabet Σ and $k, s_1 \ldots s_k, s \in \mathbb{N}$. We call c *linear* if each variable $x_i^j \in X_{s_1 \ldots s_k}$ occurs at most once in c for each $i \in k, j \in s_i$. Furthermore, we call c *non-deleting* if each $x_i^j \in X_{s_1 \ldots s_k}$ occurs at least once.

For each $k, s, s_1, .., s_k \in \mathbb{N}$, each composition representation $c \in C^{(\Sigma)}_{s_1 \ldots s_k, s}$ represents a function from $\Sigma^{s_1} \times \ldots \times \Sigma^{s_k}$ to Σ^s that, for each $i \in [k]$ and $j \in s_i$, replaces the variable x_i^j with the j-th component of the i-th argument. We call this function the *composition represented by c*.

Definition 24 (multiple context-free grammar)**.** Let Σ be an alphabet. A *multiple context-free grammar (MCFG) over* Σ is a tuple $G = (N, \Sigma, S, P)$ where

- N is a finite, $\mathbb{N}$-sorted set (*nonterminal symbols*),
- $S \in N_1$ (*initial nonterminal*), and
- $P \subseteq N \times C^{(\Sigma)} \times N^*$ is a finite, $(\mathbb{N}^* \times \mathbb{N})$-sorted set (*rules*) where

$$P_{(s_1 \ldots s_k, s)} \subseteq N_s \times C^{(\Sigma)}_{(s_1 \ldots s_k, s)} \times (N_{s_1} \times \ldots \times N_{s_k})$$

  for each $k, s, s_1, \ldots, s_k \in \mathbb{N}$ such that the composition in each rule is linear. ■

Let $G = (N, \Sigma, S, P)$ be an MCFG. We will denote each rule of the form $(A, c, A_1 \ldots A_k)$ by $A \to c(A_1, \ldots, A_k)$. For each rule $A \to c(A_1, \ldots, A_k) \in P_{s_1 \ldots s_k, s}$ for some $k, s, s_1, \ldots, s_k \in \mathbb{N}$, we define $\operatorname{rank}(A \to c(A_1, \ldots, A_k)) = k$, $\operatorname{fanout}(A \to c(A_1, \ldots, A_k)) = \operatorname{fanout}(A) = s$, and $\operatorname{fanout}_i(A \to c(A_1, \ldots, A_k)) = \operatorname{fanout}(A_i) = s_i$ for each $i \in [k]$.

We call G *non-deleting* if the composition representation in each rule in P is non-deleting. Let $p = A \to c(A_1, \ldots, A_k)$ be a rule in $P_{s_1 \ldots s_k, s}$ for some $s, s_1, \ldots, s_k \in \mathbb{N}$. We call each variable in $X_{s_1 \ldots s_k}$ that does not occur in c a *deleted variable*. Moreover, we will refer to each variable $x_i^j \in X_{s_1 \ldots s_k}$ as a variable of the i-th successor for some $i \in [k]$ and each $j \in s_i$.

Let $nts \colon P \to (N, N^*)$ be a function such that $nts(A \to c(A_1, \ldots, A_k)) = (A, A_1 \ldots A_k)$ for each $A \to c(A_1, \ldots, A_k) \in P$. We call the N-sorted set of trees over (P, nts) the *set of abstract syntax trees of* G and denote it by $D^{(G)}$. Furthermore, we define the function $\operatorname{yield}_A \colon D_A^{(G)} \to (\Sigma^*)^{\operatorname{fanout}(A)}$ for each $A \in N$ where

$$\operatorname{yield}_A\big(A \to c(A_1, \ldots, A_k)(d_1, \ldots, d_k)\big) = f\big(\operatorname{yield}_{A_1}(d_1), \ldots, \operatorname{yield}_{A_k}(d_k)\big).$$

if f is the function represented by c. If $\operatorname{yield}_S(d) = [w]$ for some $d \in D_S^{(G)}$ and $w \in \Sigma^*$, then we call w the *yield of* d.

Definition 25 (weighted MCFG)**.** Let $(W, +, \cdot, 0, 1)$ be a complete and strong bimonoid. A *W-weighted multiple context-free grammar* is a tuple (G, μ) where $G = (N, \Sigma, S, P)$ is an MCFG and μ is a function $\mu \colon P \to (W \setminus \{0\})$. ■

Let (G, μ) be a W-weighted MCFG with $G = (N, \Sigma, S, P)$. We recursively define the *weight of an abstract syntax tree* $\big(A \to f(A_1, \ldots, A_k)\big)(d_1, \ldots, d_k) \in D_A^{(G)}$ for each $A \in N$ and $A \to f(A_1, \ldots, A_k) \in P$ as

$$\mu(A \to f(A_1, \ldots, A_k)(d_1, \ldots, d_k)) = \mu(A \to f(A_1, \ldots, A_k)) \cdot \prod_{i \in [k]} \mu(d_i),$$

and the *language of* (G, μ) as the function $[\![G, \mu]\!] \colon \Sigma^* \to W$ where

$$[\![G, \mu]\!](w) = \sum_{d \in D_S^{(G)} \colon \mathrm{yield}_S(d) = [w]} \mu(d)$$

for each $w \in \Sigma^*$.

Example 26. Example for a multiple context-free grammar. Let (G, μ) be a weighted MCFG with $G = (\{S, A, B\}, \{a, b, c, d\}, S, \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\})$ and

$$\begin{aligned}
\rho_1 &= S \to [x_1^1 x_2^1 x_1^2 x_2^2](A, B), \\
\rho_2 &= A \to [a x_1^1, c x_1^2](A), \\
\rho_3 &= A \to [\varepsilon, \varepsilon](), \\
\rho_4 &= B \to [b x_1^1, d x_1^2](B), \\
\rho_5 &= B \to [\varepsilon, \varepsilon]()
\end{aligned}
\qquad
\mu(\rho) = \begin{cases}
1 & \text{if } \rho = \rho_1 \\
0.3 & \text{if } \rho = \rho_2 \\
0.7 & \text{if } \rho = \rho_3 \\
0.4 & \text{if } \rho = \rho_4 \\
0.6 & \text{if } \rho = \rho_5
\end{cases}$$

Figure 2.2 shows an abstract syntax tree d of G that produces the 1-dimensional vector $\mathrm{yield}_S(d) = [aabccd]$. The weight of d is

$$\begin{aligned}
\mu(d) &= 1 \cdot \mu(\rho_2(\rho_2(\rho_3))) \cdot \mu(\rho_4(\rho_5)) \\
&= 1 \cdot 0.3 \cdot \mu(\rho_2(\rho_3)) \cdot 0.4 \cdot \mu(\rho_5) \\
&= 1 \cdot 0.3 \cdot 0.3 \cdot \mu(\rho_3) \cdot 0.4 \cdot 0.6 \\
&= 1 \cdot 0.3 \cdot 0.3 \cdot 0.7 \cdot 0.4 \cdot 0.6 = 0.01512.
\end{aligned}$$

Since d is the only abstract syntax tree in G that produces $[aabccd]$, $[\![G, \mu]\!](aabccd) = \mu(d)$. ■

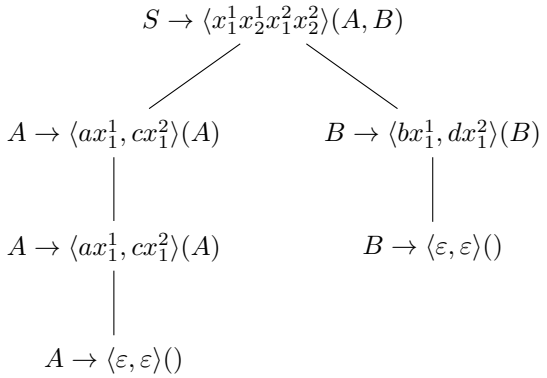


Figure 2.2: An abstract syntax tree of the example grammar in Example 26.

2.4 Multiple Dyck languages

Definition 27 (Dyck language)**.** Let Σ be an alphabet. The *Dyck language over* Σ is the equivalence class $\mathrm{D}(\Sigma) = [\varepsilon]_{\equiv_\Sigma}$ of the congruence relation $\equiv_\Sigma$ over the free monoid $((\Sigma \cup \overline{\Sigma})^*, \cdot)$ where

$$\sigma\overline{\sigma} \equiv_\Sigma \varepsilon$$

for each $\sigma \in \Sigma$. ■

Example 28. Let $\Sigma = \{ \text{(} , \langle , \text{[} \}$ and $\overline{\Sigma} = \{ \text{)} , \rangle , \text{]} \}$ such that $\overline{\text{(}} = \text{)}$, $\overline{\langle} = \rangle$ and $\overline{\text{[}} = \text{]}$. Intuitively, the Dyck language $\mathrm{D}(\Sigma)$ contains all words over the brackets in $\Sigma \cup \overline{\Sigma}$ that are correctly opening and closing brackets. E.g., the words $\text{( )}, \text{([} \langle \text{ } \rangle \text{])}$, and $\langle \text{ ( ) [} \langle \text{ } \rangle \text{] } \rangle$ are in $\mathrm{D}(\Sigma)$ whereas $\text{( ]}, \langle \text{ ( ] [ ) } \rangle$, and $\langle \text{ ( ) [} \rangle \text{]}$ are not. ■

Yoshinaka, Kaji, and Seki [YKS10] introduced multiple Dyck languages in terms of a multiple context-free grammar. We will continue with an alternative definition by Denkinger [Den16a] using a congruence relation.

Definition 29 (multiple Dyck language)**.** Let Σ be an alphabet and $\mathfrak{P}$ a partition of Σ. The *multiple Dyck language over Σ with respect to $\mathfrak{P}$* is the equivalence class $\mathrm{mD}(\Sigma, \mathfrak{P}) = [\varepsilon]_{\equiv_{\Sigma,\mathfrak{P}}}$ of the congruence relation $\equiv_{\Sigma,\mathfrak{P}}$ over the free monoid $((\Sigma \cup \overline{\Sigma})^*, \cdot)$ where

$$\sigma_1 v_1 \overline{\sigma_1} u_1 \dots u_{k-1} \sigma_k v_k \overline{\sigma_k} \equiv_{\Sigma,\mathfrak{P}} u_1 \dots u_{k-1}$$

for each $k \in \mathbb{N}$, $u_1, \dots, u_{k-1}, v_1, \dots, v_k \in \mathrm{D}(\Sigma)$ and $\{\sigma_1, \dots, \sigma_k\} \in \mathfrak{P}$ if $v_1 \dots v_k \equiv_{\Sigma,\mathfrak{P}} \varepsilon$. ■

Lemma 30. *Let Σ be an alphabet and $\mathfrak{P}$ a partition of Σ. The multiple Dyck language over Σ with respect to $\mathfrak{P}$ is a subset of the Dyck language over Σ, i.e.*

$$\mathrm{mD}(\Sigma, \mathfrak{P}) \subseteq \mathrm{D}(\Sigma).$$

Proof. In Definition 29, we define the congruence relation $\equiv_{\Sigma,\mathfrak{P}}$ by a dissection of a word into Dyck words that are enclosed by matching opening and closing symbols in $\Sigma \cup \overline{\Sigma}$. This satisfies the cancelling relation of the Dyck language $\mathrm{D}(\Sigma)$. Concatenating these parts satisfies the congruence relation of the Dyck language over Σ as well. Thus, we require each multiple Dyck word over Σ to be a Dyck word over Σ. □

Example 31. Again, let $\Sigma = \{ \text{(} , \langle , \text{[} \}$ and $\overline{\Sigma} = \{ \text{)} , \rangle , \text{]} \}$ such that $\overline{\text{(}} = \text{)}$, $\overline{\langle} = \rangle$ and $\overline{\text{[}} = \text{]}$. Now, we consider the partition $\mathfrak{P} = \left\{ \{ \text{(} \}, \{ \langle , \text{[} \} \right\}$ of Σ.

As in Example 28, all words in $\mathrm{mD}(\Sigma, \mathfrak{P})$ are well-bracketed. But additionally to Dyck words, we must be able to cancel all brackets in a partition cell of $\mathfrak{P}$ at once. So, e.g. $\text{([} \langle \text{ } \rangle \text{])} \notin \mathrm{mD}(\Sigma, \mathfrak{P})$. By Definition 29, we dissect $\text{([} \langle \text{ } \rangle \text{])}$ into $\sigma_1 v_1 \overline{\sigma_1}$ where $\sigma_1 = \text{(}$ and $v_1 = \text{[} \langle \text{ } \rangle \text{]}$. In this case, $\{\sigma_1\} \in \mathfrak{P}$ and $v_1 \in \mathrm{D}(\Sigma)$ so we check if $\text{[} \langle \text{ } \rangle \text{]} \in \mathrm{mD}(\Sigma, \mathfrak{P})$. Now, we dissect $\text{[} \langle \text{ } \rangle \text{]}$ into $\sigma_1' v_1' \overline{\sigma_1'}$ where $\sigma_1' = \text{[}$ and $v_1' = \langle \text{ } \rangle$ and see that $\{\sigma_1'\} \notin \mathfrak{P}$. ■

Chapter 3

Chomsky-Schützenberger parsing

In this chapter we will summarize the approach of *k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars* as it was introduced by Denkinger [Den17b]. For each $k \in \mathbb{N}$, we define the k-best parsing problem for weighted multiple context-free grammars as Denkinger [Den17b, Definition 5.3] did.

Definition 32 (k-best parsing problem of MCFG)**.**
given: a W-weighted MCFG (G, μ) over Σ, partial order $\trianglelefteq$ on W and $w \in \Sigma^*$,
output: a sequence $d_1 \dots d_k$ such that

- $d_1, \dots, d_k \in D^{(G)}(w)$,
- $\mu(d_1) \trianglelefteq \dots \trianglelefteq \mu(d_k)$, and
- for each $d \in D^{(G)}(w) \setminus \{d_1, \dots, d_k\}$ holds $\mu(d_k) \trianglelefteq \mu(d)$

We denote the k-best parsing problem for (G, μ), $\trianglelefteq$, and w by $\mathrm{parse}_{k,\trianglelefteq}(G, \mu, w)$. ■

For the remainder of this chapter, let (G, μ) be a W-weighted non-deleting multiple context-free grammar with $G = (N, \Sigma, P, S)$, and let w be a word over Σ.

Much like Hulden [Hul09, Section 3, page 154], we can distinguish three steps in the parsing algorithm: [1]

0. the definitions of the regular language R, a weighted alphabetic string homomorphism h and a multiple Dyck language D according to the Chomsky-Schützenberger representation of (G, μ) [Den16a, Theorem 4.10]
1. $R^w = \{v \mid h(v)(w) \neq 0\}$ [Den17b, Definition 5.5],
2. $R^{local} \colon (R \cap R^w) \to W$, such that $R^{local}(v) = h(v)(w)$ for each $v \in R \cap R^w$ [Den17b, Definition 5.21], and, finally,
3. $\mathrm{parse}_{k,\trianglelefteq}(G, \mu, w) = \Big(\mathrm{ordered}_{\trianglelefteq} \circ \mathrm{filter}(D) \circ \mathrm{map}(\mathrm{toderiv} \circ \mathrm{take}(k))\Big)\big(R^{local}\big)$ [Den17b, Definition 5.21].

Figure 3.1 shows these steps in a flow chart. We want to point out that these constructions imply an encoding of abstract syntax trees of the grammar G in words of $R \cap D$ that is decoded with the function toderiv.

[1] We will not take the preparations into account that are listed in Item 0 since the constructions in this step only depend on the grammar and not the word to parse.

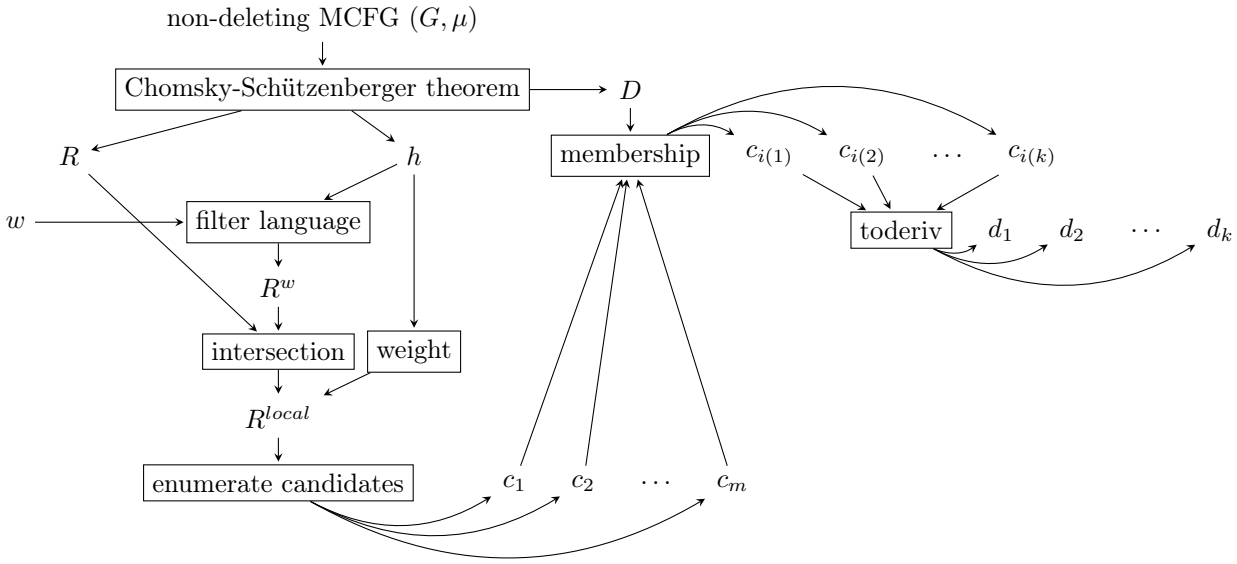


Figure 3.1: Flow chart showing our approach to k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars.

In this chapter, we will describe each of those steps separately and emphasize the construction of the automata that recognize the languages R, R^w, R^{local}, and D. Specifically, we will describe the finite state automaton $\mathcal{A}$ that recognizes R in Section 3.2, the finite state automaton $\mathcal{F}^{(w)}$ that recognizes R^w in Section 3.3, the weighted finite state automaton that recognizes R^{local} in Section 3.4, and the tree-stack automaton $\mathcal{T}$ that recognizes D in Section 3.5. The relations between these automata are visualized in Figure 3.2.

Beyond these automata, we will describe the construction of a non-deleting MCFG for each MCFG in Section 3.1, and the function toderiv in Section 3.6.

3.1 Non-deleting MCFG

In this section, we will briefly describe the construction of a non-deleting multiple context-free grammar (or LCFRS) for each MCFG. We will not further discuss the topic, but rather keep it for integrity since we provide an implementation of it.

Seki, Matsumura, Fujii, and Kasami [SMFK91, Lemma 2.2] proved by construction that there is an equivalent non-deleting MCFG for each MCFG. Denkinger [Den17b, Lemma 2.6] generalized the construction to weighted MCFGs.

In this construction, we extend the set of nonterminal symbols with the information about deleted components. The fanout of each resulting nonterminal is reduced by the amount of deleted components. We construct the set of rules for each of these nonterminals by removing the components of the composition representation according to the deletions. Finally, the weights assigned to these constructed rules are the same as the weights of the original rules.

With the construction of each non-deleting rule we have to take care of the following:

- we need to determine the right-hand side nonterminals[2], and

[2] The information about the deletion for the right-hand side nonterminals is unambiguously determined by the variables contained in the remaining components of the composition representation.

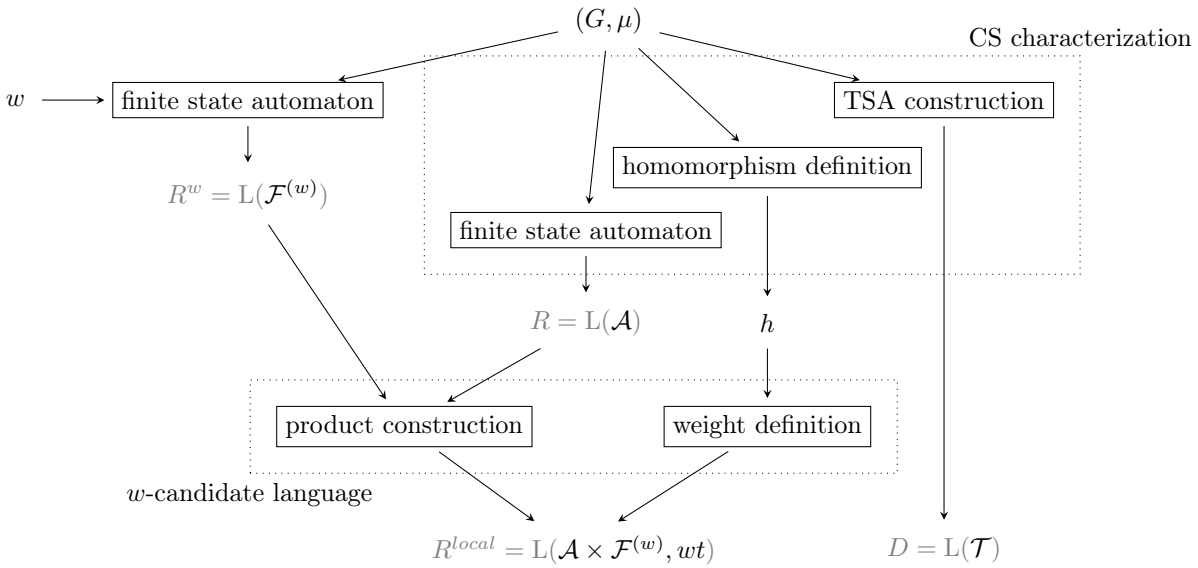


Figure 3.2: Overview over the automata constructions in this thesis. The box 'CS characterization' corresponds to definition of the homomorphism h and automata constructions that recognize the languages R and D in the Chomsky-Schützenberger characterization of MCFG. The box 'w-candidate language' refers to the same box in Figure 3.1.

- we need to shift the second indices of the variables in the remaining components of the composition representation[3]. For example, consider a composition representation that contains the variables x_1^2, x_1^5 and x_1^6. We substitute them with x_1^1, x_1^2 and x_1^3, respectively.

Definition 33 (non-deleting MCFG)**.** Let $((N, \Sigma, P, S), \mu)$ be a W-weighted MCFG. We construct the W-weighted non-deleting MCFG $((N', \Sigma, P', \{(S, \varnothing)\}), \mu')$ where

$$N' = \{(A, F) \mid A \in N, F \subseteq [\mathrm{fanout}(A)]\},$$

and P' is the smallest set such that, for each $k, l \in \mathbb{N}$, $A \to [u_1, \ldots, u_l](A_1, \ldots, A_k) \in P$, $m \in [k] \cup \{0\}$ and $j(1), \ldots, j(m) \in [l]$ with $j(1) < \ldots < j(m)$,

- for each $i \in [k]$, $\Phi_i \subseteq [\mathrm{fanout}(A_i)]$ is the largest set such that $j \in \Phi_i$ iff the variable x_i^j does not occur in the composition representation $[u_{j(1)}, \ldots, u_{j(m)}]$,
- we obtain the non-deleting composition representation

  $$[u'_{j(1)}, \ldots, u'_{j(m)}] \in CF_{\Sigma,(j(m),(s_1-|\Phi_1|)\ldots(s_k\ldots-|\Phi_k|))}$$

  from $[u_{j(1)}, \ldots, u_{j(m)}]$ by shifting the second indices of all remaining variables to the smallest value while preserving the natural order, and
- $(A, \{j(1), \ldots, j(m)\}) \to [u'_{j(1)}, \ldots, u'_{j(m)}]((A_1, \Phi_1), \ldots, (A_k, \Phi_k))$ is a rule in P'.

Furthermore, we define $\mu' \colon P' \to W$ such that $\mu'(\rho') = \mu(\rho)$ for each $\rho' \in P'$ if $\rho \in P$ is the rule we used to construct ρ'. ■

[3] Since we reduce the fanout of nonterminals by removing components in rules, we also need to alter the access to the remaining components. We fill the gaps, i.e. deleted variables, by shifting the variables to smaller values.

Example 34. Consider the MCFG $(\{S, A\}, \{a, b\}, \{\rho_1, \rho_2\}, S)$ with

$$\rho_1 = S \to [x_1^2](A),$$
$$\rho_2 = A \to [ax_1^1, bx_1^2](A).$$

Clearly, we see that the composition representation of ρ_1 deletes x_1^1. We construct the non-deleting MCFG $G = (\{(S, \varnothing), (S, \{1\}), (A, \varnothing), (A, \{1\}), (A, \{2\}), (A, \{1, 2\})\}, \{a, b\}, \{\rho_1', \rho_2', \rho_3', \rho_4', \rho_5', \rho_6'\}, (S, \varnothing))$ with the following rules

$$\rho_1' = (S, \varnothing) \to [x_1^1]\big((A, \{1\})\big),$$
$$\rho_2' = (S, \{1\}) \to [\,]\big((A, \{1, 2\})\big),$$
$$\rho_3' = (A, \varnothing) \to [ax_1^1, bx_1^2]\big((A, \varnothing)\big),$$
$$\rho_4' = (A, \{1\}) \to [bx_1^1]\big((A, \{1\})\big),$$
$$\rho_5' = (A, \{2\}) \to [ax_1^1]\big((A, \{2\})\big),$$
$$\rho_6' = (A, \{1, 2\}) \to [\,]\big((A, \{1, 2\})\big).$$

Inspecting ρ_1', we see that the right-hand side nonterminal is now $(A, \{1\})$, indicating that we were deleting the first component of the production of A in the original rule. Moreover, we see that the composition representation of ρ_1 now contains the variable x_1^1 instead of x_1^2. ■

We construct the set of rules and nonterminals for each possible combination of deletions, including the deletion of all components. With this construction, we will most probably obtain nonterminals and rules that can not be reached. Thus, we will construct a grammar that only contains the set of *reachable rules and nonterminals*.

Definition 35 (reachable nonterminals and rules)**.** Let $G = (N, \Sigma, P, S)$ be an MCFG. The set of *reachable nonterminals in* G is the smallest set such that

- S is reachable, and
- if the nonterminal $A \in N$ is reachable and there is a rule $A \to f(A_1, \ldots, A_k) \in P$, then each nonterminal $A_1, \ldots, A_k$ is reachable.

The set of *reachable rules in* G contains each rule $A \to f(A_1, \ldots, A_k) \in P$ such that A is a *reachable nonterminal*. ■

Consider the MCFGs G and $G' = (N, \Sigma, P, S)$ where N is the set of reachable nonterminals in G and P is the set of reachable rules in G. Since the definition of reachability is closely related to the derivation of MCFG, it is easy to see that $D_S^{(G)} = D_S^{(G')}$ and $[\![G]\!] = [\![G']\!]$.

Example 36. We continue Example 34 and recall the constructed non-deleting MCFG G. It is easy to see that $(S, \varnothing)$ and $(A, \{1\})$ are the only two reachable nonterminals. $(S, \varnothing)$ is the initial nonterminal and there is only ρ_1' with it on its left-hand side. $(A, \{1\})$ again, is the only right-hand side nonterminal of ρ_1' and also of ρ_4', the only rule with $(A, \{1\})$ as left-hand side nonterminal.

We can construct the MCFG $G' = \Big(\big\{(S, \varnothing), (A, \{1\})\big\}, \Sigma, \{\rho_1', \rho_4'\}, (S, \varnothing)\Big)$ that contains the set of reachable nonterminals and rules. ■

3.2 Generator language

In this section, we deal with the definition of the *generator language* R with respect to a non-deleting MCFG. As we already mentioned in the overview in the beginning of Chapter 3, R is a part of the Chomsky-Schützenberger characterization of our MCFG and was introduced by Yoshinaka, Kaji, and Seki [YKS10, Section 3.2]. Although, we will use a more restrictive definition by Denkinger [Den17b, Definition 5.15].

Definition 37 (generator alphabet)**.** Let $G = (N, \Sigma, P, S)$ be an MCFG. The *generator alphabet with respect to* G is the set

$$\begin{aligned}\Delta(G) = &\{\langle_\sigma \mid \sigma \in \Sigma\} \\ &\cup \{\langle_\rho^{(j)} \mid \rho \in P, j \in \text{fanout}(\rho)\} \\ &\cup \{\langle_{\rho,i}^{(j)} \mid \rho \in P, i \in [\text{rank}(\rho)], j \in [\text{fanout}_i(\rho)]\}.\end{aligned}$$

If G is clear from the context, we will denote $\Delta(G)$ by Δ. ■

Let Δ be the generator alphabet of our grammar G. We define the set $\overline{\Delta}$ that contains the matching closing brackets to Δ. For the rest of this section, we will denote the symbols $\overline{\langle_\sigma}, \overline{\langle_\rho^{(j)}}$ and $\overline{\langle_{\rho,i}^{(j)}} \in \overline{\Delta}$ for some $\sigma \in \Sigma$, $\rho \in P$ and $i, j \in \mathbb{N}$ by $\rangle_\sigma, \rangle_\rho^{(j)}$ and $\rangle_{\rho,i}^{(j)}$, respectively. Additionally, we will denote the string $\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_n}\rangle_{\sigma_n} \in (\Delta \cup \overline{\Delta})^*$ by $\widetilde{\sigma_1 \dots \sigma_n}$ for each $\sigma_1, \dots, \sigma_n \in \Sigma$.

Additionally to the generator alphabet Δ, we define the *set of generator fragments* $\widehat{\Delta} \subset (\Delta \cup \overline{\Delta})^*$, a finite subset of words over the brackets in Δ and $\overline{\Delta}$.

Definition 38 (generator fragments)**.** Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $\widehat{\Delta}(G)$ be the smallest set such that for each rule $\rho = A \to [v_1, \dots, v_s](B_1, \dots, B_k) \in P$ and each component of the composition representation v_c for $c \in [s]$ of the form

$$v_c = u_{c,0} x_{i(c,1)}^{j(c,1)} u_{c,1} \dots x_{i(c,m_c)}^{j(c,m_c)} u_{c,m_c}$$

- $\langle_\rho^{(c)} \widetilde{u_{c,0}} \rangle_\rho^{(c)} \in \widehat{\Delta}$ only if $m_c = 0$,
- $\langle_\rho^{(c)} \widetilde{u_{c,0}} \langle_{\rho,i(c,1)}^{(j(c,1))} \in \widehat{\Delta}$ if $m_c > 0$,
- $\rangle_{\rho,i(c,l)}^{(j(c,l))} \widetilde{u_{c,l}} \langle_{\rho,i(c,l+1)}^{(j(c,l+1))} \in \widehat{\Delta}$ for each $l \in [m_c - 1]$, and
- $\rangle_{\rho,i(c,m_c)}^{(j(c,m_c))} \widetilde{u_{c,m_c}} \rangle_\rho^{(c)} \in \widehat{\Delta}$ if $m_c > 0$,

where $u_{c,0}, \dots, u_{c,m_c} \in \Sigma^*$ and each $x_{i(c,0)}^{j(c,0)}, \dots, x_{i(c,m_c)}^{j(c,m_c)}$ is a variable.

We call $\widehat{\Delta}(G)$ the *set of generator fragments with respect to* G. If G is clear from the context, we will denote the set $\widehat{\Delta}(G)$ just by $\widehat{\Delta}$. ■

Since $\widehat{\Delta}$ is finite, we will use it like any other alphabet, regardless that its elements are words over $\Delta \cup \overline{\Delta}$. Usually, we will consider words over $\widehat{\Delta}$ as words over $\Delta \cup \overline{\Delta}$, for example we consider $\widehat{\Delta} \subset (\Delta \cup \overline{\Delta})^*$.

We want to point out that the first and last symbol of every element in $\widehat{\Delta}$ occurs exactly once in all words of $\widehat{\Delta}$. We can thus unambiguously split words over $\widehat{\Delta}$ into the elements of $\widehat{\Delta}$. Also note that each element in $\widehat{\Delta}$ is of form $\delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_k}\rangle_{\sigma_k}\delta'$ for some $k \in \mathbb{N}, \sigma_1, \dots, \sigma_k \in \Sigma$ and $\delta, \delta' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\})$.

Definition 39 (generator language)**.** Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG. The *generator automaton with respect to* G is the deterministic FSA $\mathcal{A}(G) = (Q, \widehat{\Delta}, S^{(1)}, \overline{S^{(1)}}, T)$ where

- $Q = \{A^{(i)} \mid A \in N, i \in \text{fanout}(A)\} \cup \{\overline{A^{(i)}} \mid A \in N, i \in \text{fanout}(A)\}$,
- for each rule $\rho = A \to [v_1, \ldots, v_s](B_1, \ldots, B_k)$ in P and each component of the composition representation v_c for $c \in [s]$ of form $v_c = u_{c,0} x_{i(c,1)}^{j(c,1)} u_{c,1} \ldots x_{i(c,m_c)}^{j(c,m_c)} u_{c,m_c}$ where each $u_{c,0}, \ldots, u_{c,m_c} \in \Sigma^*$ and each $x_{i(c,0)}^{j(c,0)}, \ldots, x_{i(c,m_c)}^{j(c,m_c)}$ is a variable, T is the smallest set that contains
  - $\left(A^{(c)}, \langle_{\rho}^{(c)} \widetilde{u_{c,0}} \rangle_{\rho}^{(c)}, \overline{A^{(c)}}\right)$ only if $m_c = 0$,
  - $\left(A^{(c)}, \langle_{\rho}^{(c)} \widetilde{u_{c,0}} \langle_{\rho,i(c,1)}^{(j(c,1))}, B_{i(c,1)}^{(j(c,1))}\right)$ only if $m_c > 0$,
  - $\left(\overline{B_{i(c,l-1)}^{(j(c,l-1))}}, \rangle_{\rho,i(c,l-1)}^{(j(c,l-1))} \widetilde{u_{c,l-1}} \langle_{\rho,i(c,l)}^{(j(c,l))}, B_{i(c,l)}^{(j(c,l))}\right)$ only if $m_c > 0$ for each $l \in [m_c]$, and
  - $\left(\overline{B_{i(c,m_c)}^{(j(c,m_c))}}, \rangle_{\rho,i(c,m_c)}^{(j(c,m_c))} \widetilde{u_{c,m_c}} \rangle_{\rho}^{(c)}, \overline{A^{(c)}}\right)$ only if $m_c > 0$.

We call $\mathrm{L}(\mathcal{A}(G))$ the *generator language with respect to* G and abbreviate $\mathcal{A}(G)$ by $\mathcal{A}$ and $\mathrm{L}(\mathcal{A}(G))$ by R, if G is clear from the context. ■

Example 40. Recall the grammar G from Example 26 with the set of rules $R = \{\rho_1, \ldots, \rho_5\}$

$$\begin{aligned}
\rho_1 &= S \to [x_1^1 x_2^1 x_1^2 x_2^2](A, B),\\
\rho_2 &= A \to [a x_1^1, c x_1^2](A),\\
\rho_3 &= A \to [\varepsilon, \varepsilon](),\\
\rho_4 &= B \to [b x_1^1, d x_1^2](B),\\
\rho_5 &= B \to [\varepsilon, \varepsilon]().
\end{aligned}$$

The generator alphabet is the set of brackets

$$\begin{aligned}
\Delta = &\left\{\langle_a, \langle_b, \langle_c, \langle_d\right\}\\
&\cup \left\{\langle_{\rho_1}^{(1)}, \langle_{\rho_2}^{(1)}, \langle_{\rho_2}^{(2)}, \langle_{\rho_3}^{(1)}, \langle_{\rho_3}^{(2)}, \langle_{\rho_3}^{(1)}, \langle_{\rho_4}^{(2)}, \langle_{\rho_5}^{(1)}, \langle_{\rho_5}^{(2)}\right\}\\
&\cup \left\{\langle_{\rho_1,1}^{(1)}, \langle_{\rho_1,2}^{(1)}, \langle_{\rho_1,1}^{(2)}, \langle_{\rho_1,2}^{(2)}, \langle_{\rho_2,1}^{(1)}, \langle_{\rho_2,1}^{(2)}, \langle_{\rho_4,1}^{(1)}, \langle_{\rho_4,1}^{(2)}\right\}.
\end{aligned}$$

The generator automaton

$$\mathcal{A} = \left(\left\{ S^{(1)}, \overline{S^{(1)}}, A^{(1)}, A^{(2)}, \overline{A^{(1)}}, \overline{A^{(2)}}, B^{(1)}, B^{(2)}, \overline{B^{(1)}}, \overline{B^{(2)}} \right\}, \ \Delta \cup \overline{\Delta}, \ \left\{ \overline{S^{(1)}} \right\}, \ S^{(1)}, \ T_G \right)$$

contains the set of transitions as shown in Figure 3.3;

The word $\langle_{\rho_1}^{(1)} \langle_{\rho_1,1}^{(1)} \langle_{\rho_3}^{(1)} \rangle_{\rho_3}^{(1)} \rangle_{\rho_1,1}^{(1)} \langle_{\rho_1,2}^{(1)} \langle_{\rho_5}^{(1)} \rangle_{\rho_5}^{(1)} \rangle_{\rho_1,2}^{(1)} \langle_{\rho_1,1}^{(2)} \langle_{\rho_3}^{(2)} \rangle_{\rho_3}^{(2)} \rangle_{\rho_1,1}^{(2)} \langle_{\rho_1,2}^{(2)} \langle_{\rho_5}^{(2)} \rangle_{\rho_5}^{(2)} \rangle_{\rho_1,2}^{(2)} \rangle_{\rho_1}^{(1)}$ is an element of $\mathrm{L}(\mathcal{A})$. ■

3.3 Homomorphism and filter language

Let us recall the summary in the beginning of Chapter 3. In this section, we define the weighted alphabetic string homomorphism $h\colon (\Delta \cup \overline{\Delta})^* \to \Sigma^* \to W$, the *filter language* $R^w \subseteq (\Delta \cup \overline{\Delta})^*$ of w in h, and the finite state automaton $\mathcal{F}^{(w)}$ that recognizes R^w.

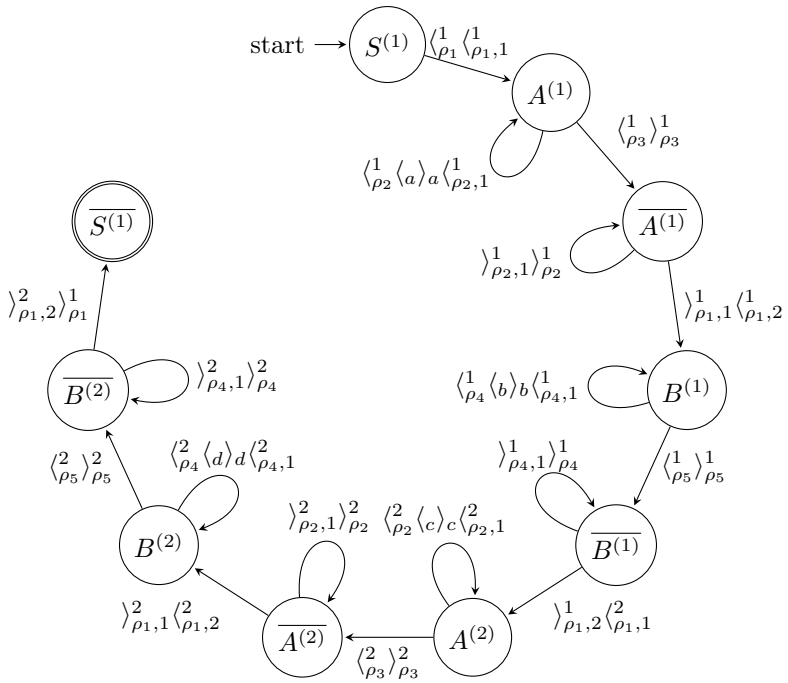


Figure 3.3: Generator automaton as constructed in Example 40 using the grammar from Example 26.

As we recall from the previous sections, we use words over the bracket alphabet $\Delta \cup \overline{\Delta}$ to represent abstract syntax trees of G. The homomorphism h will translate each of those bracket words into the yield of the abstract syntax tree. Furthermore, the weight of this translation is the weight of this abstract syntax tree in the weighted grammar (G, μ).

Figure 3.2 points out that both, the homomorphism h and the deterministic finite state automaton $\mathcal{F}^{(w)}$, will be used for the construction of $(\mathcal{A} \times \mathcal{F}^{(w)}, wt)$.

In the following, we define the homomorphism h. This definition is a composition of the two homomorphisms that were introduced by Yoshinaka, Kaji, and Seki [YKS10, Section 3.2, definition of h] and Denkinger [Den17b, Definition 5.17]. The latter one utilizes a factorization of the weights in our weighted grammar [Den17b, Section 5]. Denkinger [Den17b, Definition 3.2, Lemma 3.10, and Example 3.11] already composed the first homomorphism with another homomorphism that does not factorize the weights. We will apply this construction in the following definition.

Definition 41. Let (G, μ) be a W-weighted MCFG with $G = (N, \Sigma, P, S)$, $\trianglelefteq$ a partial order on W, and $(W, +, \cdot, 0, 1)$ a $\trianglelefteq$-factorizable, complete and strong bimonoid. Furthermore, let Δ be the generator alphabet with respect to G.

We define the weighted alphabetic string homomorphism $h^{(\Delta,\mu,\trianglelefteq)} \colon (\Delta \cup \overline{\Delta})^* \to (\Sigma^* \to W)$ such that

$$h(\delta_1 \dots \delta_k)(w) = \sum_{\substack{\sigma_1,\dots,\sigma_k \in \Sigma \cup \{\varepsilon\} \colon \\ w = \sigma_1 \dots \sigma_k}} \prod_{i \in [k]} \hat{h}(\delta_i)(\sigma_i)$$

where $\hat{h}$ is a monomial such that for each $\delta \in \Delta \cup \overline{\Delta}$ and $\sigma \in \Sigma \cup \{\varepsilon\}$

$$\hat{h}(\delta)(\sigma) = \begin{cases} 1 & \text{if } \delta = \langle_\sigma \text{ and } \sigma \in \Sigma \\ p_{\rho,2\cdot i-1} & \text{if } \delta = \langle_\rho^i \text{ for some } \rho \in P, i \in \text{fanout}(\rho) \text{ and } \sigma = \varepsilon \\ p_{\rho,2\cdot i} & \text{if } \delta = \rangle_\rho^i \text{ for some } \rho \in P, i \in \text{fanout}(\rho) \text{ and } \sigma = \varepsilon \\ 1 & \text{if } \delta \text{ is neither of the form } \langle_\rho^i \text{ nor } \rangle_\rho^i \text{ for some } \rho \in P, i \in \text{fanout}(\rho) \\ & \text{nor } \langle_\sigma \text{ for some } \sigma \in \Sigma, \text{ and } \sigma = \varepsilon \\ 0 & \text{otherwise,} \end{cases}$$

and, for each $\rho \in P$, $p_{\rho,1}, \ldots, p_{\rho,2\cdot\text{fanout}(\rho)} \in W$ such that $p_{\rho,1} \ldots p_{\rho,2\cdot\text{fanout}(\rho)}$ is the $(\trianglelefteq, 2 \cdot \text{fanout}(\rho))$-factorization of $\mu(\rho)$.

If (G, μ) and $\trianglelefteq$ are clear from the context, we will denote $h^{(\Delta,\mu,\trianglelefteq)}$ by h and call it the *homomorphism with respect to* (G, μ) *and* $\trianglelefteq$. ■

We will define the language R^w slightly different than Denkinger [Den17b, Definition 5.5, Definition 5.21] who used $R_{h,w}$, the domain of w in h, in the place we will use R^w. But, we will ensure that our definition is compatible by Theorem 44

Definition 42 (filter language)**.** Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG, $\widehat{\Delta}$ the set of generator fragments with respect to G, Δ the generator alphabet with respect to G, and $\sigma_1, \ldots, \sigma_k \in \Sigma$ for some $k \in \mathbb{N}$.

We call the deterministic finite state automaton

$$\mathcal{F}(G, \sigma_1 \ldots \sigma_k) = (\{0, \ldots, k\}, \widehat{\Delta}, 0, \{k\}, T)$$

the *filter automaton for* w *with respect to* G where

$$\begin{aligned} T = \big\{ & (l, \delta\langle_{\sigma_{l+1}}\rangle_{\sigma_{l+1}} \ldots \langle_{\sigma_{l'}}\rangle_{\sigma_{l'}} \delta', l') \\ & \mid l \in \{0, \ldots, k\}, l' \in \{l, \ldots, k\}, \\ & \delta, \delta' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\}) \colon \delta\langle_{\sigma_{l+1}}\rangle_{\sigma_{l+1}} \ldots \langle_{\sigma_{l'}}\rangle_{\sigma_{l'}} \delta' \in \widehat{\Delta}\big\}. \end{aligned}$$

Also, we call $\mathrm{L}(\mathcal{F}(G, \sigma_1 \ldots \sigma_k))$ the *filter language for* w *with respect to* G.

If G is clear from the context, we denote $\mathcal{F}(G, w)$ by $\mathcal{F}^{(w)}$ and $\mathrm{L}(\mathcal{F}(G, w))$ by R^w for each $w \in \Sigma^*$. ■

Lemma 43. *(G, μ) be a W-weighted non-deleting MCFG over some alphabet Σ, $\trianglelefteq$ a partial order on W, and $w \in \Sigma$. Moreover, let R^w the filter language of w with respect to G, $\widehat{\Delta}$ the set of generator fragments of G, and h the homomorphism with respect to (G, μ) and $\trianglelefteq$.*

Then,

$$R^w = \widehat{\Delta}^* \cap R_{h,w}.$$

Proof. Let (G, μ) be a W-weighted non-deleting MCFG over some alphabet Σ, $\trianglelefteq$ a partial order on W, and $w \in \Sigma$. Moreover, let R^w the filter language of w and $\mathcal{F}^{(w)} = (Q, \widehat{\Delta}, q_{init}, Q_F, T)$ the filter automaton of w with respect to G, $\widehat{\Delta}$ the set of generator fragments of G and Δ the generator alphabet of G.

$$v \in \widehat{\Delta}^* \cap R_{h^{(\Delta,\mu,\trianglelefteq)},w}$$

$$
\begin{aligned}
&\iff \exists n, l(1), \dots, l(n) \in \mathbb{N}, \sigma_{1,1}, \dots, \sigma_{n,l(n)} \in \Sigma, \\
&\qquad \delta_1, \delta_1', \dots, \delta_n, \delta_n' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\}) : \\
&\qquad \delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1', \dots, \delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n' \in \widehat{\Delta} \\
&\qquad \wedge v = \delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1' \dots \delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n' \\
&\qquad \wedge w = \sigma_{1,1} \dots \sigma_{n,l(n)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Definition 38, Definition 3)} \\
&\iff \exists n, l(1), \dots, l(n) \in \mathbb{N}, \sigma_{1,1}, \dots, \sigma_{n,l(n)} \in \Sigma, \qquad\qquad \text{(by Definition 42)} \\
&\qquad \delta_1, \delta_1', \dots, \delta_n, \delta_n' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\}) : \\
&\qquad v = \delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1' \dots \delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n' \\
&\qquad \wedge \big(0, \delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1', l(1)\big), \dots, \\
&\qquad\quad \big(l(n-1), \delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n', l(n)\big) \in T \\
&\qquad \wedge l(n) \in Q_F \\
&\iff \exists n, l(1), \dots, l(n) \in \mathbb{N}, \sigma_{1,1}, \dots, \sigma_{n,l(n)} \in \Sigma, \qquad\qquad \text{(by Definition 7)} \\
&\qquad \delta_1, \delta_1', \dots, \delta_n, \delta_n' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\}) : \\
&\qquad v = \delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1' \dots \delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n' \\
&\qquad \wedge 0\big(\delta_1 \langle_{\sigma_{1,1}} \rangle_{\sigma_{1,1}} \dots \langle_{\sigma_{1,l(1)}} \rangle_{\sigma_{1,l(1)}} \delta_1'\big) l(1) \dots \\
&\qquad\quad l(n-1)\big(\delta_n \langle_{\sigma_{n,1}} \rangle_{\sigma_{n,1}} \dots \langle_{\sigma_{n,l(n)}} \rangle_{\sigma_{n,l(n)}} \delta_n'\big) l(n) \in \mathrm{Runs}(\mathcal{F}^{(w)}) \\
&\iff v \in \mathrm{L}(F^{(w)}) \qquad\qquad \text{(by Definition 7)} \\
&\iff v \in R^w \qquad\qquad \text{(by Definition 42)}
\end{aligned}
$$

□

Theorem 44. *(G, μ) be a W-weighted non-deleting MCFG over some alphabet Σ, $\trianglelefteq$ a partial order on W, and $w \in \Sigma$. Moreover, let R^w the filter language of w with respect to G, R the generator language with respect to G and h the homomorphism with respect to (G, μ) and $\trianglelefteq$.*

Then,

$$R \cap R^w = R \cap R_{h,w}.$$

Proof.

$$
\begin{aligned}
R \cap R^w &= R \cap \widehat{\Delta}^* \cap R_{h,w} && \text{(by Lemma 43)} \\
&= R \cap R_{h,w} && \text{(by Definition 39, } R \subseteq \widehat{\Delta}^*\text{)}
\end{aligned}
$$

□

Example 45. We continue Example 40 with the construction of h as shown in Definition 41.

We observe, that $h(v)(\varepsilon) \neq 0$ for all $v \in \widehat{\Delta}$ that do not contain any brackets of form $\langle_\sigma$ for some $\sigma \in \Sigma$. We will denote the subset of $\widehat{\Delta}$ containing all these fragments by $\widehat{\Delta}^{(\varepsilon)}$. For all $v \in \widehat{\Delta}$ that contain such brackets, i.e. fragments of form $v = \delta \langle_{\sigma_1} \rangle_{\sigma_1} \dots \langle_{\sigma_k} \rangle_{\sigma_k} \delta'$ for some $\delta, \delta' \in (\Delta \cup \overline{\Delta}) \cap \{\langle_\sigma \mid \sigma \in \Sigma\}$ and $\sigma_1, \dots, \sigma_k \in \Sigma$, $h(v)(\sigma_1 \dots \sigma_k) \neq 0$.

Using $\widehat{\Delta}$, we can construct the filter automaton $\mathcal{F}^{(ac)}$ as shown in Figure 3.4. ■

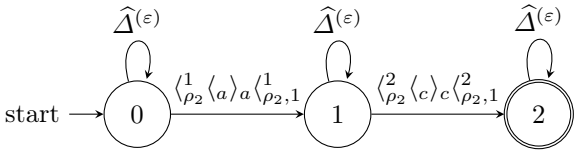


Figure 3.4: Filter automaton constructed in Example 45 using the grammar from Example 26. $\widehat{\Delta}^{(\varepsilon)}$ denotes the subset of fragments in $\widehat{\Delta}$ that do not contain brackets of form $\langle_\sigma$ for some $\sigma \in \Sigma$.

3.4 Candidate language

We recall the overview in the beginning of Chapter 3. In the sections Sections 3.2 and 3.3, we described two finite state automata, $\mathcal{A}$ and $\mathcal{F}^{(w)}$ that recognize the languages R and R^w, respectively. In this section, we construct a weighted finite state automaton that recognizes the language R^{local} using $\mathcal{A}$, $\mathcal{F}^{(w)}$ and the homomorphism h.

Definition 46. Let (G, μ) be a W-weighted grammar over an alphabet Σ, $\trianglelefteq$ a partial order on W, and $w \in \Sigma^*$. Moreover, let $\mathcal{A}$ be the generator automaton with respect to G, $\mathcal{F}^{(w)}$ the filter automaton for w with respect to G, and $\mathcal{A} \times \mathcal{F}^{(w)} = (Q, \widehat{\Delta}, q_{init}, Q_F, T)$ the product of both automata where $\widehat{\Delta}$ is the set of generator fragments with respect to G.

We define the function $wt\colon T \to W$ such that

$$wt\big(q, \delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_l}\rangle_{\sigma_l}\delta', q'\big) = h\big(\delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_l}\rangle_{\sigma_l}\delta'\big)\big(\sigma_1 \dots \sigma_l\big)$$

where $\delta, \delta' \in (\Delta \cup \overline{\Delta}) \setminus (\{\langle_\sigma \mid \sigma \in \Sigma\} \cup \{\rangle_\sigma \mid \sigma \in \Sigma\})$, $l \in \mathbb{N}$, $\sigma_1, \dots, \sigma_l \in \Sigma\colon \delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_l}\rangle_{\sigma_l}\delta' \in \widehat{\delta}$, and $q, q' \in Q$.

If (G, μ), $\trianglelefteq$ and $w \in \Sigma^*$ are clear from the context, we denote $\mathrm{L}(\mathcal{A} \times \mathcal{F}^{(w)}, wt)$ by R^{local} and call it the *w-candidate language with respect to* (G, μ). ■

This definition combines the intersection of R and R^w with the weight function according to h as it was introduced by Denkinger [Den17b, Definition 5.21]. In this definition, we emphasize the connection to the definitions by Hulden [Hul09, Section 3].

Example 47. We continue our example using the weighted MCFG from Example 26, the homomorphism h and the filter automaton $\mathcal{F}^{(ab)}$ from Example 45, and the generator automaton $\mathcal{A}$ from Example 40.

We can see the constructed automaton $(\mathcal{A} \times \mathcal{F}^{(ab)}, wt)$ in Figure 3.5.

■

3.5 Multiple Dyck language

Definition 48. Let $G = (N, T, P, S)$ be a non-deleting MCFG and Δ the generator alphabet with respect to G. We call the set

$$\begin{aligned}\mathfrak{P}(G) = &\left\{\left\{\langle_\sigma^{(1)}\right\} \mid \sigma \in \Sigma\right\} \\ &\cup \left\{\left\{\langle_p^{(j)} \mid j \in \mathrm{fanout}(p)\right\} \mid p \in P\right\} \\ &\cup \left\{\left\{\langle_{p,i}^{(j)} \mid j \in [\mathrm{fanout}_i(p)]\right\} \mid p \in P, i \in [\mathrm{rank}(p)]\right\}\end{aligned}$$

the *partition of Δ with respect to G*.

If G is clear from the context, we denote $\mathfrak{P}(G)$ by $\mathfrak{P}$ and $\mathrm{mD}(\Delta, \mathfrak{P})$ by D. We call $\mathrm{mD}(\Delta, \mathfrak{P})$ the *multiple Dyck language with respect to G*. ■

Example 49. Recall the grammar G from Example 26 and the generator set Δ from Example 40. The partition of Δ with respect ot G is the set

$$\begin{aligned}\mathfrak{P} = \Big\{ & \big\{\langle^{(1)}_{a}\big\}, \big\{\langle^{(1)}_{b}\big\}, \big\{\langle^{(1)}_{c}\big\}, \big\{\langle^{(1)}_{d}\big\} \\ & , \big\{\langle^{(1)}_{p_1}\big\}, \big\{\langle^{(1)}_{p_2}, \langle^{(2)}_{p_2}\big\}, \big\{\langle^{(1)}_{p_3}, \langle^{(2)}_{p_3}\big\}, \big\{\langle^{(1)}_{p_4}, \langle^{(2)}_{p_4}\big\}, \big\{\langle^{(1)}_{p_5}, \langle^{(2)}_{p_5}\big\} \\ & , \big\{\langle^{(1)}_{p_1,1}, \langle^{(2)}_{p_1,1}\big\}, \big\{\langle^{(1)}_{p_1,2}, \langle^{(2)}_{p_1,2}\big\}, \big\{\langle^{(1)}_{p_2,1}, \langle^{(2)}_{p_2,1}\big\}, \big\{\langle^{(1)}_{p_4,1}, \langle^{(2)}_{p_4,1}\big\} \\ & \Big\}\end{aligned}$$

The multiple Dyck language over Δ with respect to G, $\mathrm{mD}(\Delta, \mathfrak{P})$, contains the word

$$\langle^{(1)}_{p_1}\ \langle^{(1)}_{p_1,1}\langle^{(1)}_{p_3}\ \rangle^{(1)}_{p_3}\rangle^{(1)}_{p_1,1}\ \langle^{(1)}_{p_1,2}\langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\rangle^{(1)}_{p_1,2}\ \langle^{(2)}_{p_1,1}\langle^{(2)}_{p_3}\ \rangle^{(2)}_{p_3}\rangle^{(2)}_{p_1,1}\ \langle^{(2)}_{p_1,2}\langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5}\rangle^{(2)}_{p_1,2}\ \rangle^{(1)}_{p_1}$$

because

$$\begin{aligned}
& \langle^{(1)}_{p_1}\ \langle^{(1)}_{p_1,1}\langle^{(1)}_{p_3}\ \rangle^{(1)}_{p_3}\rangle^{(1)}_{p_1,1}\ \langle^{(1)}_{p_1,2}\langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\rangle^{(1)}_{p_1,2}\ \langle^{(2)}_{p_1,1}\langle^{(2)}_{p_3}\ \rangle^{(2)}_{p_3}\rangle^{(2)}_{p_1,1}\ \langle^{(2)}_{p_1,2}\langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5}\rangle^{(2)}_{p_1,2}\ \rangle^{(1)}_{p_1} \\
\equiv_{\Delta,\mathfrak{P}} \quad & \langle^{(1)}_{p_1,1}\langle^{(1)}_{p_3}\ \rangle^{(1)}_{p_3}\rangle^{(1)}_{p_1,1}\ \langle^{(1)}_{p_1,2}\langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\rangle^{(1)}_{p_1,2}\ \langle^{(2)}_{p_1,1}\langle^{(2)}_{p_3}\ \rangle^{(2)}_{p_3}\rangle^{(2)}_{p_1,1}\ \langle^{(2)}_{p_1,2}\langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5}\rangle^{(2)}_{p_1,2} \\
\equiv_{\Delta,\mathfrak{P}} \quad & \langle^{(1)}_{p_3}\ \rangle^{(1)}_{p_3}\ \ \langle^{(1)}_{p_1,2}\langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\rangle^{(1)}_{p_1,2}\ \ \langle^{(2)}_{p_3}\ \rangle^{(2)}_{p_3}\ \ \langle^{(2)}_{p_1,2}\langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5}\rangle^{(2)}_{p_1,2} \\
\equiv_{\Delta,\mathfrak{P}} \quad & \langle^{(1)}_{p_3}\ \rangle^{(1)}_{p_3}\ \ \langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\ \ \langle^{(2)}_{p_3}\ \rangle^{(2)}_{p_3}\ \ \langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5} \\
\equiv_{\Delta,\mathfrak{P}} \quad & \langle^{(1)}_{p_5}\ \rangle^{(1)}_{p_5}\ \ \langle^{(2)}_{p_5}\ \rangle^{(2)}_{p_5} \\
\equiv_{\Delta,\mathfrak{P}} \quad & \varepsilon.
\end{aligned}$$

■

In this section we will describe a variant of *tree-stack automata* that was introduced by Denkinger [Den16b, Definintion 1] as an instance of automata with data storage. We construct a tree-stack automaton that recognizes the multiple Dyck language D.

In addition to the recognition of Dyck languages, we need to cancel multiple pairs of matching brackets simultaneously. The general idea behind that construction is that we store each set of brackets that we need to cancel simultaneously in a single node of the tree stack.

Each node of the tree stack is a tuple of a closing bracket in $\overline{\Delta} \cup \{\varepsilon\}$ and a set of opening brackets in Δ. We intuitively consider the first component as the matching counterpart of the bracket we just read. The second component represents all remaining bracket pairs that need to be canceled at the same time as the read opening bracket. We can summarize the behavior of the automaton as follows:

- if we read an opening bracket δ in Δ, we move the tree-stack pointer up to a node whose second component contains δ, remove δ from the second component and store $\overline{\delta}$ in the first component to remember that we need to read it next, and
- if we read a closing bracket $\overline{\delta}$ in $\overline{\Delta}$, and the first component of the current node is $\overline{\delta}$, then we remove the first component of this node and move the tree-stack pointer to the parent node.

If the tree-stack only consists of nodes of the form $(\varepsilon, \varnothing)$, then the tree-stack automaton will accept.

Definition 50 (tree-stack)**.** Let A be a set and $@ \notin A$ a symbol. An A-tree-stack is a tuple (u, α) where $u \in \mathcal{U}_{\{@\} \cup A}$ is an unranked tree over $A \cup \{@\}$ and $\alpha \in \mathrm{pos}(u)$ is a position in u.

We usually use the symbol @ only used as the root symbol and in some leaves of u as placeholder. We denote the set of all A-tree-stacks by $\mathfrak{T}_A$. ■

Definition 51 (mD-tree-stack automaton)**.** Let Σ be an alphabet and $\mathfrak{P}$ a partition of Σ. An mD-tree-stack over Σ is a $((\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma))$-tree-stack. We define the following instructions for mD-tree-stacks over Σ for each $(u, \alpha) \in \mathfrak{T}_{(\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma)}$:

- $\mathrm{up}(u, \alpha) = \{(u, \alpha i) \mid i \in \mathbb{N} \colon \alpha i \in \mathrm{pos}(u)\} \cup \{\big(\mathrm{push}(u, \alpha, @), \mathrm{children}(u[\alpha]) + 1\big)\}$
- for each $\mathfrak{p} \in \mathcal{P}(\Sigma)$, $\mathrm{setempty}(\mathfrak{p})(u, \alpha) = \begin{cases} \left\{\left(u\big(\alpha/(\varepsilon, \mathfrak{p})\big), \alpha\right)\right\} & \text{if } u[\alpha] = @ \\ \{(u, \alpha)\} & \text{otherwise} \end{cases}$
- for each $i \in \mathbb{N}$ such that $\alpha i \in \mathrm{pos}(u)$, $\mathrm{down}(u, \alpha i) = \{(u, \alpha)\}$,
- for each $\overline{\sigma} \in \overline{\Sigma}$, $\mathrm{chkfst}(\overline{\sigma})(u, \alpha) = \begin{cases} \{(u(\alpha/(\varepsilon, \mathfrak{p})), \alpha)\} & \text{if } u(\alpha) = (\overline{\sigma}, \mathfrak{p}) \text{ for some } \mathfrak{p} \in \mathcal{P}(\Sigma) \\ \varnothing & \text{otherwise} \end{cases}$,
- for each $\sigma \in \Sigma$,

$$\mathrm{rmsnd}(\sigma)(u, \alpha) = \begin{cases} \left\{\big(u\big(\alpha/(\overline{\sigma}, \mathfrak{p} \setminus \{\sigma\})\big), \alpha\big)\right\} & \text{if } u(\alpha) = (\varepsilon, \mathfrak{p}) \text{ for some } \mathfrak{p} \in \\ & \mathcal{P}(\Sigma) \text{ s.t. } \sigma \in \mathfrak{p} \\ \varnothing & \text{otherwise.} \end{cases}$$

We define the *mD-tree-stack storage with respect to Σ and $\mathfrak{P}$* as the tuple

$$\mathrm{S}_{\mathrm{mD}}^{\Sigma, \mathfrak{P}} = (\mathfrak{T}_{((\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma))}, R, F, @)$$

where $F = \{(@(u_1, \ldots, u_k), \varepsilon) \mid k \in \mathbb{N}, u_1, \ldots, u_k \in \mathcal{U}_{\{(\varepsilon, \varnothing)\}}\}$ and

$$\begin{aligned} R = &\{\mathrm{up} \circ \mathrm{setempty}(\sigma, \mathfrak{P}[\sigma]) \circ \mathrm{rmsnd}(\sigma) \mid \sigma \in \Sigma\} \\ &\cup \{\mathrm{down} \circ \mathrm{chkfst}(\overline{\sigma}) \mid \overline{\sigma} \in \overline{\Sigma}\}. \end{aligned}$$

The *mD-tree-stack automaton with respect to Σ and $\mathfrak{P}$* is the $\mathrm{S}_{\mathrm{mD}}^{\Sigma, \mathfrak{P}}$-automaton $\mathcal{T}(\Sigma, \mathfrak{P}) = (\{q\}, \Sigma \cup \overline{\Sigma}, q, \{q\}, T)$ where

$$\begin{aligned} T = &\{(q, \sigma, \mathrm{up} \circ \mathrm{setempty}(\sigma, \mathfrak{P}[\sigma]) \circ \mathrm{rmsnd}(\sigma), q) \mid \sigma \in \Sigma\} \\ &\cup \{(q, \overline{\sigma}, \mathrm{down} \circ \mathrm{chkfst}(\overline{\sigma}), q) \mid \sigma \in \Sigma\}. \end{aligned}$$ ■

Theorem 52. *Let Σ be an alphabet and $\mathfrak{P}$ a partition of Σ. The $\mathrm{S}_{\mathrm{mD}}^{\Sigma, \mathfrak{P}}$-automaton $\mathcal{T}(\Sigma, \mathfrak{P})$ is not deterministic.*

Counter example. By Definition 6, we call an automaton deterministic if, for each state q and terminal symbol σ, there is at most one transition (q, σ, r, q') in the set of transitions where r is some instruction and q' another state, and $|r(c)| \geq 1$ for each storage configuration c. The automaton $\mathcal{T}(\Sigma, \mathfrak{P})$, constructed in Definition 51 for some alphabet Σ and some partition $\mathfrak{P}$ of Σ, is not deterministic in general. Since $\mathcal{T}(\Sigma, \mathfrak{P})$ is defined such that there is exactly one state and on transition for each symbol in $\Sigma \cup \overline{\Sigma}$, we will show that there are instructions that relate some tree-stacks to multiple other tree-stacks.

We can see in Definition 51 that there may be multiple tree-stacks in the image of up for certain tree-stacks. We construct a minimal example, such that the instruction $\mathrm{up} \circ \mathrm{setempty}(\sigma, \mathfrak{P}[\sigma]) \circ \mathrm{rmsnd}(\sigma)$ relates a tree-stack to multiple other tree-stacks.

Let $\Sigma = \{ \langle , [\}$ and $\mathfrak{P} = \big\{\{ \langle , [\}\big\}$, we denote $\overline{\langle}$ and $\overline{]}$ by $\rangle$ and $]$, respectively. We consider the tree-stack $(@(\varepsilon, \{ \langle \}), \varepsilon) \in \mathfrak{T}_{((\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma))}$ and the instruction $r = \mathrm{up} \circ \mathrm{setempty}(\langle , \mathfrak{P}[\langle]) \circ \mathrm{rmsnd}(\langle)$. We see that

$$\begin{aligned}
&\mathrm{up} \circ \mathrm{setempty}(\langle , \mathfrak{P}[\langle]) \circ \mathrm{rmsnd}(\langle)\big((@(\varepsilon, \{ \langle \}), \varepsilon)\big) \\
&= \mathrm{setempty}(\langle , \mathfrak{P}[\langle]) \circ \mathrm{rmsnd}(\langle)\Big(\big\{(@((\varepsilon, \{ \langle \}), 1), (@((\varepsilon, \{ \langle \}), @), 2)\big\}\Big) \\
&= \mathrm{rmsnd}(\langle)\Big(\big\{(@(\varepsilon, \{ \langle \}), 1), (@((\varepsilon, \{ \langle \}), (\varepsilon, \{ \langle , [\})), 2)\big\}\Big) \\
&= \big\{(@(\rangle , \varnothing), 1), (@((\varepsilon, \{ \langle \}), (\rangle , \{ [\})), 2)\big\}.
\end{aligned}$$

And thus that there are $u \in \mathfrak{T}_{(\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma)}$ where $|r(u)| > 1$. □

Theorem 53. *Let Σ be and alphabet and $\mathfrak{P}$ a partition of Σ. Then,*

$$\mathrm{L}(\mathcal{T}(\Sigma, \mathfrak{P})) = \mathrm{mD}(\Sigma, \mathfrak{P}).$$

Justification. Let Σ be and alphabet and $\mathfrak{P}$ a partition of Σ. By Definition 29, a word $w \in (\Delta \cup \overline{\Delta})^*$ is in $\mathrm{mD}(\Delta, \mathfrak{P})$ iff there are $k \in \mathbb{N}$, $\{\sigma_1, \ldots, \sigma_k\} \in \mathfrak{P}$, and $u_0, \ldots, u_k, v_1, \ldots, v_k \in \mathrm{D}(\Sigma)$ such that $u_0 \ldots u_k, v_1 \ldots v_k \in \mathrm{mD}(\Sigma, \mathfrak{P})$ and $w = u_0 \sigma_1 v_1 \overline{\sigma_1} u_1 \ldots \sigma_k v_k \overline{\sigma_k} u_k$. For now, we will assume that $u_0 \ldots u_k = \varepsilon$ and discuss the other cases later.

An accepting tree-stack in $\mathcal{T}(\Sigma, \mathfrak{P})$ is of the form (t, ε) where the root of t is @ and all other nodes are of the form $(\varepsilon, \varnothing)$.

In the construction of $\mathcal{T}(\Sigma, \mathfrak{P})$ in Definition 51, we can easily see that there is exactly one transition for each symbol $\sigma \in \Sigma$ and each symbol $\overline{\sigma} \in \overline{\Sigma}$ where the instruction in the transition for σ moves the tree-stack pointer up, and the instruction in the transition for $\overline{\sigma}$ moves it down. Since we start at the root node, we apply the tree-stack instructions in the transitions for $\sigma_1, \ldots, \sigma_k$ to a tree-stack pointing to the root node. We assume that, after processing the instruction in the transition for σ_1, the tree-stack pointer points to the node $(\overline{\sigma_1}, \{\sigma_2, \ldots, \sigma_k\})$ at the position 1 which is a child of the root node. In the following, the automaton processes the symbols in v_1. Since v_1 is a Dyck word, the stack pointer will only move in the subtree at 1, and point to position 1 when v_1 was processed. When we read the symbol $\overline{\sigma_1}$, we change the node at position 1 to $(\varepsilon, \{\sigma_2, \ldots, \sigma_k\})$ and move the stack pointer to ε.

These steps repeat for the remainder of the word. Since $v_1 \ldots v_k$ is in $\mathrm{mD}(\Sigma, \mathfrak{P})$, all child trees of the node at 1 consist only of nodes of the form $(\varepsilon, \varnothing)$, and the node at 1 is also of the form $(\varepsilon, \varnothing)$ after processing $\sigma_1, \overline{\sigma_1}, \ldots, \sigma_k$ and $\overline{\sigma_k}$. Thus, the tree-stack automaton will accept.

If $u_0 \ldots u_k \neq \varepsilon$, it is a word of the form $\sigma'_1 v'_1 \overline{\sigma'_1} \ldots \sigma'_{k'} v_{k'} \overline{\sigma'_{k'}}$ for some $k' \in \mathbb{N}$. Moreover, there is a partition $\mathfrak{B}$ of $[k']$ such that, for each $\{i_1, \ldots, i_{\hat{k}}\} \in \mathfrak{B}$, $\sigma'_{i_1} v'_{i_1} \overline{\sigma'_{i_1}} \ldots \sigma'_{i_{\hat{k}}} v'_{i_{\hat{k}}} \overline{\sigma'_{i_{\hat{k}}}}$ is in $\mathrm{mD}(\Sigma, \mathfrak{P})$ [Den17b, Obeservation 4.4]. If all symbols $\sigma'_1, \ldots, \sigma'_{k'}$ are distinct from the symbols $\sigma_1, \ldots, \sigma_k$, the tree-stack automaton will process each $u_0, \ldots, u_k$ in the same way as we described the recognition of w above. In that case, the root node will have multiple children. When we use the tree-stack instruction up upon reading a symbol σ in $\{\sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_{k'}\}$, it will move the stack pointer to a position whose node's second component contains σ.

If the symbols are not distinct, the tree-stack instruction up will nondeterministically decide for right child position. □

3.6 Bijection between bracket words and abstract syntax trees

If we recall Figure 3.1, we can see a function toderiv that assigns an abstract syntax tree of a non-deleting MCFG G to each bracket word in $R \cap D$. As previously discussed, we use the bracket words over $\Delta \cup \overline{\Delta}$ to encode abstract syntax trees of G. Intuitively, we add matching bracket pairs at the beginning and the end of each component of the composition representation, and around variables in the composition representation of a rule. Furthermore, each terminal in the composition representation in each rule is replaced by a pair of brackets. Each of those bracket pairs can be unambiguously identified by its indices. We consider each bracket word in $R \cap D$ as the yield of an abstract syntax tree in G with these modifications.

In this section, we describe the bijection toderiv with slight differences to its introduction by Denkinger [Den17b, Algorithm 1]. Although, we will first describe tobrackets, the inverse of toderiv, to be aware of the structure of words in $R \cap D$.

tobrackets reads a specified component of the composition representation of the grammar rule in the root of the given abstract syntax tree. We apply the modifications we discussed above to the component by adding brackets to the start and end of the current component of the composition representation, around all variables, and by substituting all terminals with bracket pairs. Lastly, we substitute each variable with the respective recursive call of tobrackets.

We define the function

$$\begin{aligned}\text{tobrackets}\Big(\big(A \to [u_{1,0}x_{i(1,1)}^{j(1,1)}u_{1,1}\dots u_{1,m_1},\dots,\dots u_{l,m_l}](A_1,\dots,A_k)\big)\big(t_1,\dots,t_k\big),c\Big)\\ = \langle_p^c\ \widetilde{u_{c,0}}\ \langle_{p,j(c,1)}^{i(c,1)}\ \cdot\ \text{tobrackets}\big(t_{i(c,1)},j(c,1)\big)\ \cdot\ \rangle_{p,j(c,1)}^{i(c,1)}\ \widetilde{u_{c,1}}\ \dots\ \widetilde{u_{c,m_c}}\ \rangle_p^c.\end{aligned}$$

for each $p(t_1,\dots,t_k) \in D^{(G)}$ where $p = A \to [u_{1,0}x_{i(1,1)}^{j(1,1)}u_{1,1}\dots u_{1,m_1},\dots,\dots u_{s,m_s}](A_1,\dots,A_k)$ is a rule in P for some $u_{1,0},\dots,u_{s,m_s} \in \Sigma^*$, $i(1,1),\dots,i(s,m_s) \in [k]$ and, for each $c \in [s]$ and $y \in m_c$, $j(c,y) \in \text{fanout}(A_{i(c,y)})$, and $x_{i(1,1)}^{j(1,1)},\dots,x_s^{m_s} \in X_{(\text{fanout}(p),\text{fanout}(A_1)\dots\text{fanout}(A_k))}$.

In toderiv, the inverse function of tobrackets, we will read applied grammar rules from brackets of the form $\langle_\rho^j$ for some grammar rule ρ and $j \in \text{fanout}(\rho)$. The structure of the resulting abstract syntax tree is determined by arrangement of opening and closing brackets of the form $\langle_{\rho,i}^j$ for some grammar rule ρ, $j \in \text{fanout}(\rho)$ and $i \in \text{rank}(\rho)$. Algorithm 1 shows an algorithmic approach for the inverse of the function Section 3.6.

For this algorithm, let $\bot\colon \mathbb{N}^* \to P$ be the partial function that is undefined for each element in $\mathbb{N}^*$. For each partial function $t\colon \mathbb{N}^* \to P$, we use the notion $t[\pi] \leftarrow \rho$ to modify t such that $t(\pi) = \rho$ for each $\pi \in \mathbb{N}^*$ and $\rho \in P$. Here, we use partial functions from $\mathbb{N}^*$ to P to represent abstract syntax trees of G such that the preimage is the set of positions and the image of the function is the set of labels of the abstract syntax tree.

Algorithm 1 The function toderiv reads an abstract syntax tree off a bracket word in $D \cap R$.

Input: generator alphabet Δ, $w \in D \cap R$
Output: an abstract syntax tree in $D^{(G)}$

1: **function** TODERIV(Δ, w)
2: $t \leftarrow \bot$
3: $\pi \leftarrow \varepsilon$
4: **for** $\delta \in w$ **do**
5: **if** $\delta \in \Delta$ and δ is of form $\langle_{\rho}^{(1)}$ for some $\rho \in P$ **then**
6: $t[\pi] \leftarrow \rho$
7: **else if** $\delta \in \Delta$ and δ is of form $\langle_{p,i}^{(j)}$ for some $i, j \in \mathbb{N}, \rho \in P$ **then**
8: $\pi \leftarrow \pi i$ // push child position to π
9: **else if** $\delta \in \overline{\Delta}$ and δ is of form $\rangle_{p,i}^{(j)}$ for some $i, j \in \mathbb{N}, p \in P$ **then**
10: $\pi i \leftarrow \pi$ // pop last child position from π
11: **end if**
12: **end for**
13: **return** t
14: **end function**

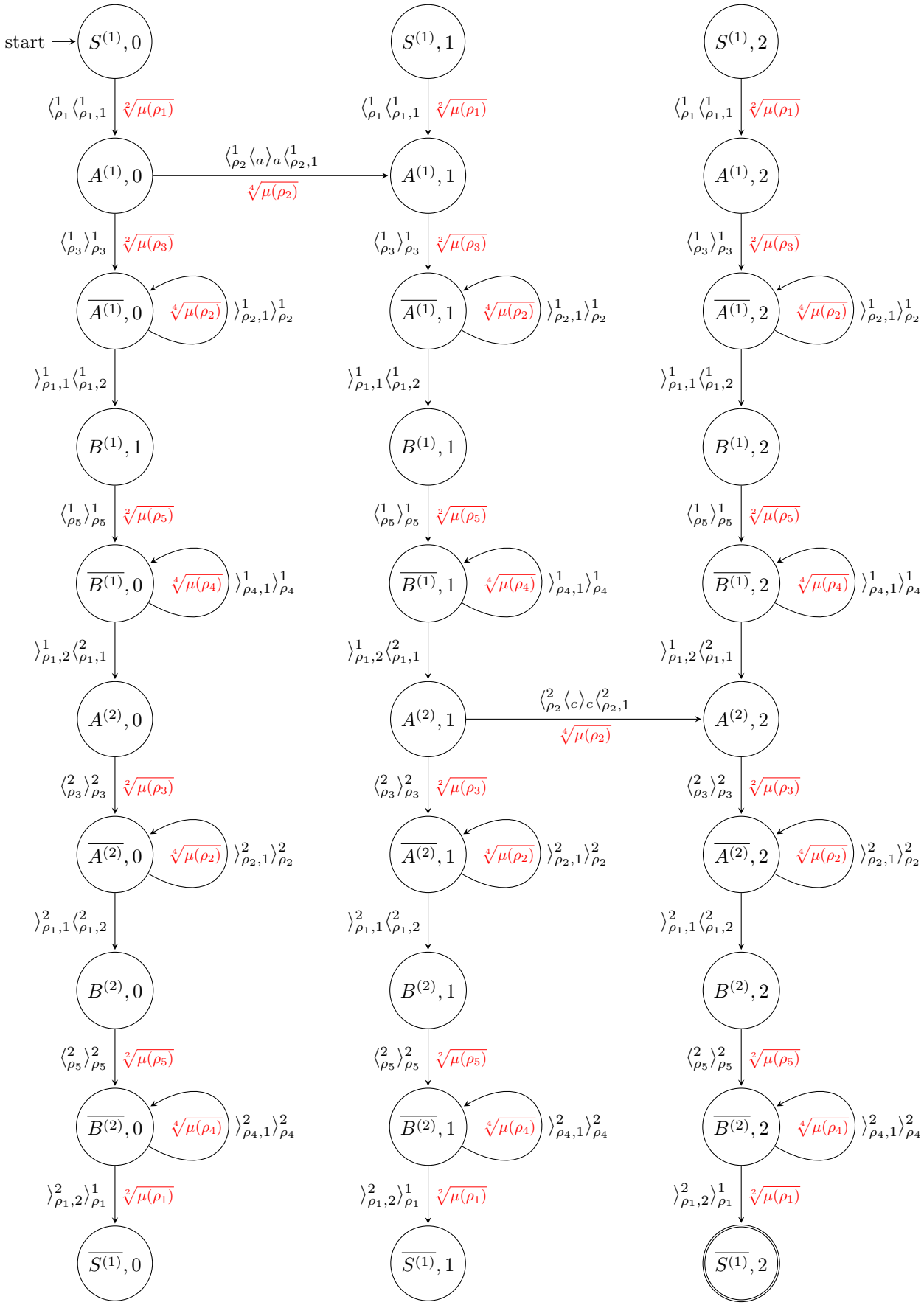


Figure 3.5: Product automaton $\mathcal{A}\times\mathcal{F}^{(ab)}$ with weights according to the monomial h. Transitions without annotated weight are weighted 1.

Chapter 4

Optimizations

Over the course of this chapter, we will discuss some optimizations for the parsing pipeline that was described in Chapter 3.

First, we will have a general look at the effectiveness of the *generator automaton* as it was described in Section 3.2. Due to the rough approximation of a multiple context-free language using a finite state automaton, it appears to be not suitable for our needs. We will introduce an alternative approach using a push-down automaton instead of the finite state automaton in Section 4.1.

Section 4.2 will deal with an alternative *filter automaton* whose set of transitions is likely to be smaller than the one described in Section 3.3.

In Section 4.3, we will introduce some heuristics utilized for the enumeration of words recognized by finite state automata and push-down automata. Since we decided to discontinue the implementation of finite state automata with the *OpenFst framework* [ARSSM07] (see Section 5.2.1), we needed to implement efficient representations for finite state automata and push-down automata. We focussed our attention specifically on the lazy evaluation of the enumeration of all words recognized by these automata. In order to enable an effective and efficient exploration of transitions, we used heuristics and a *beam search*.

Finally, Section 4.4 deals with an approach to eliminate the nondeterminism introduced by the up-instruction used in the tree-stack automaton described in Section 3.5.

4.1 Alternative generator automaton

Motivation. We recall the constructions outlined in Figures 3.1 and 3.2. The *generator automaton* $\mathcal{A}$ as it was described in Section 3.2, is constructed using the weighted non-deleting MCFG (G, p) over the alphabet Σ. Together with the *filter automaton* $\mathcal{F}^{(w)}$ and the weight assignment p, we construct $(\mathcal{A} \times \mathcal{F}^{(ab)}, wt)$, a weighted finite state automaton that recognizes candidates for encoded abstract syntax trees of G that yield some $w \in \Sigma^*$. Each of those candidates is checked for its membership in the multiple Dyck language D and decoded into an abstract syntax tree.

If a word recognized by $(\mathcal{A} \times \mathcal{F}^{(ab)}, wt)$ encodes an abstract syntax tree of G, its weight assigned by wt is the weight of the abstract syntax tree in (G, p). We use this behavior to enumerate all abstract syntax trees that yield some word $w \in \Sigma^*$ in best-first order.

During the implementation and testing of the parser, we noticed that, even for very small grammars, we have to enumerate numerous words of ordered(R^{local}) before we find the first word in D. For example, Figure 6.3 in Section 6.3.2 shows the number of words in supp(R^{local}) that were checked until the first

member of D was found. To tackle this problem, we decided to restrict $\mathrm{supp}(R^{local})$ such that it includes only Dyck words over Δ. In this section, we will introduce the context-free language $\widehat{R}$ that we will use as a replacement for R. We will define $\widehat{R}$ such that it only includes the Dyck words in R. Finally, we will show in Theorem 58 that we can replace R by $\widehat{R}$ in the Chomsky-Schützenberger representation of MCFG.

Definition 54 (generator PDA)**.** Let $G = (N, \Sigma, P, S)$ be an MCFG. The *push-down generator automaton with respect to* G is the deterministic Γ-PDA $\mathcal{K} = (Q, \widehat{\Delta}, S^{(1)}, \{\overline{S^{(1)}}\}, T)$ where

- $Q = \{A^{(i)} \mid A \in N, i \in \mathrm{fanout}(A)\} \cup \{\overline{A^{(i)}} \mid A \in N, i \in \mathrm{fanout}(A)\}$,
- Γ is the set $\{(\rho, i, j) \mid \rho \in P, i \in \mathrm{fanout}(\rho), j \in \mathrm{fanout}_i(\rho)\}$, and
- T is the smallest set such that, for each rule

$$\rho = A \to [u_{1,0} x_{i(1,1)}^{j(1,1)} u_{1,1} \dots x_{i(1,m_1)}^{j(1,m_1)} u_{1,m_1}, \dots, u_{s,0} x_{i(s,1)}^{j(s,1)} \dots u_{s,m_s}](A_1, \dots, A_k)$$

in P and $c \in [s]$,

- $\left(A^{(c)}, \langle_\rho^{(c)} \widetilde{u_{c,0}} \rangle_\rho^{(c)}, \mathrm{id}_\Gamma, \overline{A^{(c)}}\right) \in T$ only if $m_c = 0$
- $\left(A^{(c)}, \langle_\rho^{(c)} \widetilde{u_{c,0}} \langle_{\rho,i(c,1)}^{(j(c,1))}, r, A_{i(c,1)}^{(j(c,1))}\right) \in T$ only if $m_c > 0$
  where $r = \mathrm{push}(\rho, i(c,1), j(c,1))$
- $\left(\overline{A_{i(c,l)}^{(j(c,l))}}, \rangle_{\rho,i(c,l)}^{(j(c,l))} \widetilde{u_{c,l}} \langle_{\rho,i(c,l+1)}^{(j(c,l+1))}, r, A_{i(c,l+1)}^{(j(c,l+1))}\right) \in T$ for each $l \in [m_c - 1]$
  where $r = \mathrm{replace}((\rho, i(c,l), j(c,l)), (\rho, i(c,l+1), j(c,l+1)))$
- $\left(\overline{A_{i(c,m_c)}^{(j(c,m_c))}}, \rangle_{\rho,i(c,m_c)}^{(j(c,m_c))} \widetilde{u_{c,m_c}} \rangle_\rho^{(c)}, r, \varepsilon, \overline{A^{(c)}}\right) \in T$ only if $m_c > 0$
  where $r = \mathrm{pop}(\rho, i(c,m_c), j(i,m_c))$,

where $u_{1,0}, \dots, u_{s,m_s} \in \Sigma^*$, $i(1,1), \dots, i(s,m_s) \in [k]$ and, for each $c \in [s]$ and $y \in m_c$, $j(c,y) \in \mathrm{fanout}(A_{i(c,y)})$, and $x_{i(1,1)}^{j(1,2)}, \dots, x_s^{m_s} \in X_{(\mathrm{fanout}(\rho), \mathrm{fanout}(A_1) \dots \mathrm{fanout}(A_k))}$.

To emphasize the difference, we will call the generator automaton from Definition 39 the *generator FSA* and $\mathcal{K}$ the *generator PDA with respect to* G. Moreover, we call $\mathrm{L}(\mathcal{K})$ the *context-free generator language with respect to* G and if G is clear from the context, denote it by $\widehat{R}$. ■

As we can see, the construction of the push-down generator is closely related to the construction of the generator automaton described in Section 3.2. Specifically, the only difference is the introduction of push-down instructions in the transitions of the push-down generator.

Lemma 55. *Let G be an MCFG. The 0-approximation $[\mathcal{K}]_0$ of the generator PDA $\mathcal{K}$ is the generator automaton with respect to G.*

Proof idea. Intuitively, we obtain the 0-approximation of a push-down automaton by removing the instructions from the transitions. Since the construction of the generator PDA and the generator FSA are equal, except for the push-down instructions, we suppose that those two automata are equal. □

Corollary 56. *Let G be an MCFG, R the generator language with respect to G and $\widehat{R}$ the context-free generator language with respect to G. Then, $\widehat{R} \subseteq R$.*

Proof. This follows directly from Lemma 55 and Theorem 21. □

Using the push-down instructions, we are able to store context-free information. In particular, we choose the push-down instructions such that we close each opening bracket in Δ with the fitting closing bracket in $\overline{\Delta}$.

Lemma 57. *Let G be a non-deleting MCFG and Δ the generator alphabet of G. The language of the generator PDA with respect to G is the set of all Dyck words in the generator language with respect to G, i.e. $\widehat{R} = R \cap \mathrm{D}(\Delta)$.*

Proof idea. In Definitions 39 and 54, we can see that the constructions of the generator FSA and the construction of the generator PDA are very similar. In particular, the state transitions are equal as the only difference are the instructions in the definition of the generator PDA. In Corollary 56, we already noticed that the language of the generator PDA is a subset of the language of the generator FSA.

We observe in the construction of the generator PDA that each push-down symbol uniquely identifies a grammar rule and a variable in it. Since variables occur exactly once in each composition representation, we identify unique indices in the composition representation of the rule. We will distinguish three types of brackets and observe their order with respect to the canceling relation of Dyck languages.

- *Terminal brackets* of the form $\langle_\sigma$ and $\rangle_\sigma$, for some terminal symbol σ, are constructed such that the opening and the closing brackets are right next to each other.
- *Variable brackets* of the form $\langle^j_{\rho,i}$ and $\rangle^j_{\rho,i}$ for some rule ρ, $i \in \mathrm{rank}(\rho)$ and $j \in \mathrm{fanout}_i(\rho)$ are always at the end or the beginning of a transition. We choose the push-down instructions exactly to match the indices of these brackets. Thus, we ensure the correct placement with respect to the canceling rules.
- We can find the *component brackets* of the form $\langle^j_\rho$ and $\rangle^j_\rho$, for some grammar rule ρ amd $j \in \mathrm{fanout}(\rho)$, at the beginning of each transition that pushes a symbol for the first variable of the rule ρ. On the other side, we find the closing bracket in the transition that pops the push-down symbol for the last variable occurring in ρ. Thus, much like the variable brackets, we thereby ensure the correct placements of the opening and closing parts with respect to the canceling rule.

Hence, we suppose that the brackets in the language of the generator PDA are well balanced. □

Theorem 58. *Let G be a non-deleting MCFG, R the generator language with respect to G, $\widehat{R}$ the context-free generator language respect to G and D the multiple Dyck language with respect to G. Then,*

$$\widehat{R} \cap D = R \cap D.$$

Proof. Let G be a non-deleting MCFG, Δ the generator alphabet with respect to G, $\mathfrak{P}$ the partition of Δ with respect to G, R the generator language with respect to G, $\widehat{R}$ the context-free generator language with respect to G, and D the multiple Dyck language with respect to G. Then,

$$\begin{aligned}
\widehat{R} \cap D &= \widehat{R} \cap \mathrm{mD}(\Delta, \mathfrak{P}) && \text{(by Definition 54, Definition 48)} \\
&= R \cap \mathrm{D}(\Delta) \cap \mathrm{mD}(\Delta, \mathfrak{P}) && \text{(by Lemma 57)} \\
&= R \cap \mathrm{mD}(\Delta, \mathfrak{P}) && \text{(by Lemma 30)} \\
&= R \cap D && \text{(by Definition 48)}
\end{aligned}$$

□

Example 59. We continue Example 26 with the construction of a *push-down generator* as an alternative to Example 40. The structure of the resulting automaton will not change in comparison to Example 40, but the transitions will contain push-down instructions. We can see the constructed push-down generator in Figure 4.1. ■

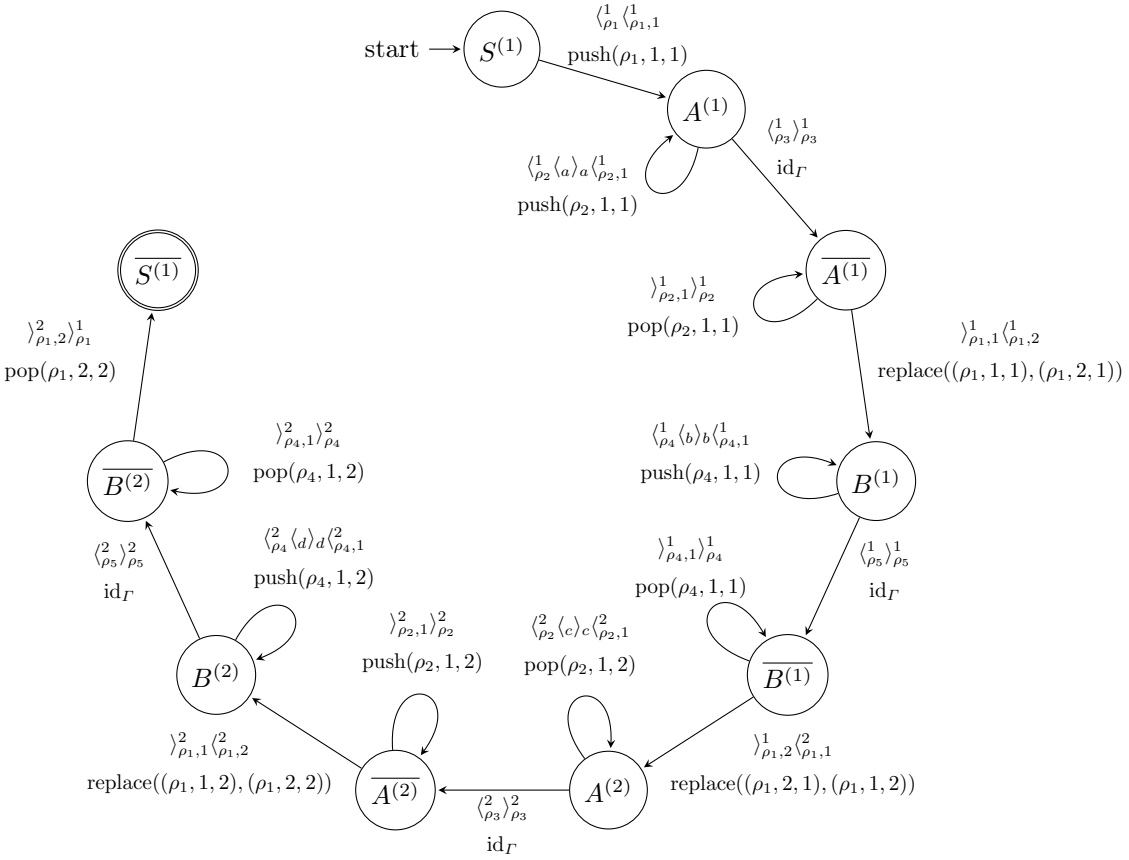


Figure 4.1: Push-down generator constructed from the non-deleting MCFG in Example 26.

4.2 Alternative filter automaton

Motivation. We recall the construction of the *filter automaton* as described in Section 3.3 and its utilization in the product construction with the *generator automaton* as outlined in Figure 3.2. We also recall the weighted alphabetic string homomorphism h that translates bracket words over $\Delta \cup \overline{\Delta}$ into words over Σ. The filter automaton $\mathcal{F}^{(w)}$ is a finite state automaton that, given a word $w \in \Sigma^*$, recognizes all bracket words in $\widehat{\Delta}^*$ that h translates into w.

In this section, we modify the filter automaton such that the set of transitions is as small as possible. We will use a definition of *productive grammar rules* to find a subset of grammar rules that is suitable to be applied in abstract syntax trees of a grammar to yield a specific word. This set of productive rules is then used to construct a filter automaton as described in Section 3.3. But since the set of productive rules with respect to a word is smaller or equal than the original set of rules, the set of transitions of the filter automaton will be smaller or equally sized as well.

If the set of transitions of this modified version of the filter automaton is smaller, we have to deal with less transitions during the product construction with the generator automaton. Moreover, the result of that product construction will have a smaller set of transitions.

Definition 60 (productive rules)**.** Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $w \in \Sigma^*$. We define $P_w \subseteq P$ as the smallest set such that, for each rule

$$\rho = A \to [u_{1,0} x_{i(1,1)}^{j(1,1)} u_{1,1} \dots x_{i(1,m_1)}^{j(1,m_1)} u_{1,m_1}, \dots, u_{s,0} x_{i(s,1)}^{j(s,1)} \dots u_{s,m_s}](A_1, \dots, A_k)$$

in P, ρ is in P_w if each sequence of terminal symbols $u_{1,0}, \dots, u_{s,m_s}$ is a subsequence of w and there is at least one rule with left-hand side A_i in P_w for each $i \in [k]$ where $u_{1,0}$, …, $u_{s,m_s} \in \Sigma^*$, $i(1,1)$, …, $i(s,m_s) \in [k]$ and, for each $\hat{s} \in [s]$ and $\hat{m} \in m_{\hat{s}}$, $j(\hat{m}, \hat{s}) \in \text{fanout}(A_{i(\hat{m},\hat{s})})$, and $x_{i(1,1)}^{j(1,1)}, \dots, x_{i(s,m_s)}^{j(s,m_s)}$ are variables. We call the set P_w the set of *w-productive rules.*

We abbreviate the tuple (N, Σ, P_w, S) by G_w. ■

Intuitively, there are two steps of filtering in this definition of productive rules with respect to some $w \in \Sigma^*$:

1. we remove each rule from P whose composition representation contains terminal sequences that are not in w, and after that,
2. we filter each rule by its *Boolean inside weight.* So, we only keep those rules that appear in any abstract syntax tree of the grammar after the first step of filtering.

Observation 61. *Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $w \in \Sigma^*$. The set of abstract syntax trees of G that yield w is equal to the set of abstract trees of G_w that yield w.*

Justification. We consider the non-deleting MCFG G and the word w. Since G is a non-deleting grammar, an abstract syntax tree of G that yields W can obviously not contain rules that includes terminal sequences that are not in w. Furthermore, an abstract syntax tree can not contain rules that contain right-hand side nonterminals which do not appear on the left-hand side of any rule. □

Definition 62 (inside filter automaton)**.** Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $w \in \Sigma^*$. We call the filter automaton for w with respect to G_w the *inside filter automaton for w with respect to G*, and if G is clear from the context, we denote it by $\widehat{\mathcal{F}^{(w)}}$, and $\mathrm{L}(\widehat{\mathcal{F}^{(w)}})$ by $\widehat{R^w}$. We also call $\widehat{R^w}$ the *inside filter language for w with respect to G.* ■

Lemma 63. *Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $w \in \Sigma^*$. Furthermore, we consider the filter automaton $\mathcal{F}^{(w)} = \left(Q, \widehat{\Delta}(G), q_0, Q_F, T\right)$ for w with respect to G, and the inside filter automaton $\widehat{\mathcal{F}^{(w)}} = \left(Q', \widehat{\Delta}(G_w), q_0, Q'_F, T'\right)$ for w with respect to G. If $P_w \subseteq P$ ($\subset$) then $\widehat{\Delta}(G_w) \subseteq \widehat{\Delta}(G)$ ($\subset$), $Q' = Q, Q'_F = Q_F$, and $T' \subseteq T$ ($\subset$).*

Proof. Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG, $w \in \Sigma^*$, $\mathcal{F}^{(w)} = \left(Q, \widehat{\Delta}(G), q_0, Q_F, T\right)$ the filter automaton with respect to G and w, and $\widehat{\mathcal{F}^{(w)}} = \left(Q', \widehat{\Delta}(G_w), q_0, Q'_F, T'\right)$ the inside filter automaton with respect to G and w. Furthermore, let $P_w \subset P$.

Each element in the set of generator fragments $\widehat{\Delta}(G)$ with respect to G is unambiguously constructed from one rule in P. Since P_w is a (proper) subset of P, $\widehat{\Delta}(G_w)$ is a (proper) subset of $\widehat{\Delta}(G)$. The same holds for the sets of transitions T and T'.

The sets Q and Q_F, and Q' and Q'_F are constructed from the set of nonterminals in G and G_w, respectively. Since G and G_w share the same set of nonterminals, the sets Q and Q', and Q_F and Q'_F are equal. □

Corollary 64. *Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG and $w \in \Sigma^*$. The filter language for w with respect to G_w ($\widehat{R^w}$) is a subset of the filter language for w with respect to G (R^w), i.e.*

$$\widehat{R^w} \subseteq R^w.$$

Proof idea. This follows directly from Corollary 64. Since both automata share the same states and the set of transitions of the inside filter automaton is a subset of the transitions of the filter automaton, the language of the inside filter automaton is a subset of the language of the filter automaton. □

Corollary 65. *Let $G = (N, \Sigma, P, S)$ be a non-deleting MCFG, $w \in \Sigma^*$, R the generator language with respect to G, D the multiple Dyck language with respect to G, R^w the filter language with respect to G and w, and $\widehat{R^w}$ the inside filter language with respect to G and w. Then*

$$R \cap \widehat{R^w} \cap D = R \cap R^w \cap D.$$

Proof idea. There is a bijection between $R \cap D$ an the set of abstract syntax trees of G [Den17b, Corollary 3.9]. Furthermore, there is a bijection between $R^{local} \cap D$ and the set of abstract syntax trees in G that yield w [Den17b, Theorem 5.22]. [1] Since $\widehat{R^w}$ is the filter language with respect to G_w, we suppose there is a bijection between $R \cap \widehat{R^w} \cap D$ and the set of abstract syntax trees of G_w that yield w. By Observation 61 $D^{(G)}(w) = D^{(G_w)}(w)$, so there is a bijection between $R \cap \widehat{R^w} \cap D$ and $R \cap R^w \cap D$.

Now, by Corollary 64, $\widehat{R^w} \subseteq R^w$, and thus $R \cap \widehat{R^w} \cap D \subseteq R \cap R^w \cap D$. Hence, we suppose these sets are equal. □

Definition 66 (modified candidate language)**.** Let $(W, +, \cdot, 0, 1)$ be a strong, commutative and factorizable bimonoid, (G, μ) be a W-weighted non-deleting MCFG over Σ, $w \in \Sigma^*$, $\mathcal{K}$ be the generator PDA with respect to G and $\widehat{\mathcal{F}^{(w)}}$ the inside filter automaton with respect to G. Let $\mathcal{K} \times \widehat{\mathcal{F}^{(w)}} = (Q, \widehat{\Delta}, q, Q_F, T)$ be the intersection of both automata. We define the weight assignment $wt \colon T \to W$ such that

$$wt(q, \delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_k}\rangle_{\sigma_k}\delta', r, q) = h(\delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_k}\rangle_{\sigma_k}\delta')(\sigma_1 \dots \sigma_k)$$

for some $\delta, \delta' \in \Delta, k \in \mathbb{N}, \sigma_1, \dots, \sigma_k \in \Sigma$ such that $(q, \delta\langle_{\sigma_1}\rangle_{\sigma_1} \dots \langle_{\sigma_k}\rangle_{\sigma_k}\delta', r, q) \in T$. We call the weighted language $\mathrm{L}(\mathcal{K} \times \widehat{\mathcal{F}^{(w)}}, wt)$ the *modified w-candidate language with respect to* (G, μ) and denote it by $\widehat{R^{local}}$ if G and w are clear from the context. ■

Observation 67. *Let $(W, +, \cdot, 0, 1)$ be a strong, commutative and factorizable bimonoid, (G, μ) be a W-weighted non-deleting MCFG over Σ, $w \in \Sigma$, D the multiple Dyck language with respect to G, R^{local} the w-candidate language with respect to (G, μ) and $\widehat{R^{local}}$ the modified w-candidate language with respect to (G, μ).*

For each $v \in \mathrm{supp}(\widehat{R^{local}})$,

$$\widehat{R^{local}}(v) = R^{local}(v).$$

Justification. We already pointed out that the structure of the generator PDA's transitions as constructed in Definition 54 is equal to the generator FSA as constructed in Definition 39. Now, in Definition 66, we use the same weight assignment as in Definition 46. Hence, we suppose that $\widehat{R^{local}}$ assigns the same weight to the word in its support as R^{local}. □

[1] This is not directly mentioned, but is necessary for this theorem.

4.3 Heuristics for weighted automata

We recall Figure 3.1 and Figure 3.2 to remind us of the use of the weighted language R^{local} and the weighted automaton $(\mathcal{A} \times \mathcal{F}^{(w)}, wt)$ that recognizes R^{local}. We will use $(\mathcal{A} \times \mathcal{F}^{(w)}, wt)$ to enumerate all words of R^{local} in best-first order.

We model the enumeration of words recognized by an automaton as a graph search. Intuitively, we traverse the graph represented by the transitions of the automaton and stop if we find final state. The concatenation of the symbols along the transitions we used to reach the final state is a recognized word. In the weighted case, the weight of this recognized word is the product of the transitions' weights. This enables us to solve this problem using algorithms like Dijkstra's search algorithm and A-star search.

To ensure an efficient and effective exploration, we will use *beam search*, a variant of the *A-star search* algorithm. Like Dijkstra's algorithm and A-star, beam search uses an *agenda* to store intermediate search results. The agenda is a priority queue that prioritizes each item using its weight and a heuristic. In each step, we remove the search result with lowest weight from the agenda, put its successors into the agenda and yield the current item. Beam search's unique feature is the limitation of the agenda to a fixed number of elements. Each time when we put items into the agenda, we will only keep the best items. Algorithm 2 shows the algorithm in more detail.

Algorithm 2 The beam search algorithm is an agenda-driven approach to explore a set of items.

Input: initial set of items I_{init}, set of target items I_F, successor relation *succ*, weight function μ, heuristic h, partial order $\trianglelefteq$, beam width b

Output: sequence of items $a_1, a_2, \ldots$ such that $\mu(a_i) \cdot h(a_i) \trianglelefteq \mu(a_{i+1}) \cdot h(a_{i+1})$ for each $i \in \mathbb{N}$

```
1:  function ITERATE(I_init, I_F, succ, h, b)
2:      Agenda ← Empty
3:      for all a in I_init do
4:          Agenda.insert_⊴(a, μ(a) · h(a))
5:      end for
6:      Agenda.limit_⊴(b)
7:      while Agenda is not empty do
8:          a ← Agenda.pop_best_⊴()
9:          for all a′ in succ(a) do
10:             Agenda.insert_⊴(a′, μ(a) · h(a))
11:         end for
12:         Agenda.limit_⊴(b)
13:         if a is in I_F then
14:             yield a
15:         end if
16:     end while
17: end function
```

We call each (intermediate) search result a *search item*. In our specific applications, e.g. for finding all runs of a weighted finite state automaton, a search item is simply a tuple of a state of the automaton and a word over its input alphabet. We interpret it as the state that we reached after following a sequence of transitions, and the sequence of all the symbols in these transitions. If we search for a specific set of items, we call this set *target items*. In automata, we can simply determine the set of target items by the acceptance criteria of an automaton. For example, a finite state automaton will accept if we reach one of its final states. Thus, the set of target items is the set of all search items with a final state.

We call the set of items we obtain from the expansion of an item the *successors of an item*. Let us consider a $[0, 1]$-weighted finite state automaton $((Q, \Sigma, q_0, Q_F, T), \mu)$ and the partial order $\geq$ as an example. For each search item $(q, w) \in Q \times \Sigma^*$ and each transition $(q, \sigma, q') \in T$, $(q', w\sigma)$ is a successor

of the search item (q, w). The weight of the successor item is the product of the weight of its predecessor and the used transition $\mu(q', w\sigma) = \mu(q, w) \cdot \mu(q, \sigma, q')$.

With this in mind, a heuristic is a function that estimates the additional weight from an item $(q, w) \in Q \times \Sigma^*$ to a target item $(q_F, v) \in Q_F \times \Sigma^*$ that can be reached from (q, w). We call a heuristic h *admissable* if, for every search item $(q, w) \in Q \times \Sigma^*$, the product of the transitions' weights that are used to reach any target item from (q, w) is less or equal to $h(q, w)$. In the case of automata, this means that the heuristic of (q, w) is greater than the weight of each run starting at q. Moreover, we call a heuristic *monotonic* if, for every search item $(q, w) \in Q \times \Sigma^*$, the heuristic for an item is greater or equal than the weight of the transition that is used to reach a successor item $(q', w\sigma)$ and the heuristic of $(q', w\sigma)$, $h(q, w) \geq \mu(q, \sigma, q') \cdot h(q', w\sigma)$.

In this section, we will describe heuristics for the exploration in weighted deterministic finite state automata and weighted deterministic push-down automata.

4.3.1 A heuristic for weighted finite state automata

Let $(\mathcal{A}, \mu)$ be a W-weighted finite state automaton with $\mathcal{A} = (Q, \Sigma, q_0, Q_F, T)$ and $\trianglelefteq$ is a partial order on W such that $p \trianglelefteq p \cdot p'$ for each $p, p' \in W$. As we already described, the set of search items we use to explore the words recognized by $\mathcal{A}$ is the set $Q \times \Sigma^*$ and our set of target items is $Q_F \times \Sigma^*$. We define the binary relation *succ* over $(Q \times \Sigma^*)$ such that

$$succ(q, w) = \{(q', w\sigma) \mid (q, \sigma, q') \in T\}$$

for each $q \in Q$ and $w \in \Sigma^*$. The heuristic for the exploration of weighted finite state automata is the least weight of a run starting with the state in a search item.

Definition 68 (heuristic for deterministic weighted fsa)**.** Let $(\mathcal{A}, \mu)$ be a W-weighted finite state automaton where $\mathcal{A} = (Q, \Sigma, q_0, Q_F, T)$, and $\trianglelefteq$ a partial order on W such that $w \trianglelefteq w \cdot w'$ for each $w, w' \in W$. The *$\trianglelefteq$-heuristic for the exploration of* $(\mathcal{A}, p)$ is the function $h \colon Q \times \Sigma^* \to W$ where

$$h \colon (q, w) \mapsto \min_{\trianglelefteq} \big\{\mu(r) \mid r \in \mathrm{Runs}(\mathcal{A}, q)\big\}. \qquad \blacksquare$$

Theorem 69. *Let $(\mathcal{A}, \mu)$ be a W-weighted FSA, and $\trianglelefteq$ a partial order on W such that $w \trianglelefteq w \cdot w'$ for each $w, w' \in W$. The $\trianglelefteq$-heuristic for the exploration of $(\mathcal{A}, \mu)$ is monotonic and admissable.*

Proof. The definition for our target items and our heuristic is closely related to the definition of runs of a weighted finite state automaton. It is easy to see that the heuristic is exactly the least weight needed for the exploration of a target item. Its monotonicity is as easy to see. Using the heuristic from above, we show that $h(q, w) \trianglelefteq \mu(q, \sigma, q') \cdot h(q', w\sigma)$ for every $(q, w) \in Q \times \Sigma^*, \sigma \in \Sigma$ and $(q', w\sigma) \in succ(q, w)$,

$$\begin{aligned}
\mu(q, \sigma, q') \cdot h(q', w\sigma) &= \mu(q, \sigma, q') \cdot \min_{\trianglelefteq} \big\{\mu(r) \mid r \in \mathrm{Runs}(\mathcal{A}, q')\big\} && \text{(by Definition 68)} \\
&\trianglerighteq \min_{\trianglelefteq} \big\{\mu(r) \mid r \in \mathrm{Runs}(\mathcal{A}, q)\big\} && \text{(by Corollary 11)} \\
&= h(q, w). && \text{(by Definition 68)}
\end{aligned}$$

□

4.3.2 A heuristic for weighted push-down automata

Let $(\mathcal{K}, \mu)$ be a W-weighted Γ-push-down automaton with $\mathcal{K} = (Q, \Sigma, q_0, Q_F, T)$. $Q \times \Gamma^* \times \Sigma^*$ is the set of search items for the exploration of $(\mathcal{K}, \mu)$ and $Q_F \times \{\varepsilon\} \times \Sigma^*$ is the set of target items. We define the binary relation over $Q \times \Gamma^* \times \Sigma^*$

$$succ(q, v, w) = \{(q', v', w\sigma) \mid (q, \sigma, r, q') \in T, v' \in r(v)\}$$

for each $q \in Q, v \in \Gamma^*$ and $w \in \Sigma$. In comparison to the heuristic for the exploration of weighted finite state automata, our heuristic for weighted finite push-down automata will consider the symbols of the push-down storage in addition to the state behavior. We know that an accepting configuration of each push-down automaton needs to have an empty push-down storage. We will approximate the weight of a run starting with the state and the push-down storage of a search item by computing the least weight for each symbol in our push-down storage of a run that is able to remove it from the push-down. We will use the greatest of these weight since we need to find a run that removes all of these push-down symbols.

Definition 70 (heuristic for deterministic weighted pda)**.** Let $(\mathcal{K}, \mu)$ be a W-weighted Γ-PDA where $\mathcal{K} = (Q, \Sigma, q_0, Q_F, T)$ and $\trianglelefteq$ is a partial ordered over W such that $w \trianglelefteq w \cdot w'$ for each $w, w' \in W$. The *$\trianglelefteq$-heuristic for the exploration of* $(\mathcal{K}, p)$ is the function $h\colon (Q \times \Gamma^* \times \Sigma^*) \to W$ where, for each $q \in Q, w \in \Sigma^*, k \in \mathbb{N}$ and $\gamma_1, \ldots, \gamma_k \in \Gamma$,

$$h\colon (q, \gamma_1 \ldots \gamma_k, w) \mapsto \begin{cases} \min\limits_{\trianglelefteq} \{\mu(r) \mid v \in \Gamma^*, r \in \mathrm{Runs}(\mathcal{K}, q, v)\} & \text{if } k = 0 \\ \max\limits_{\trianglelefteq} \Big\{ \min\limits_{\trianglelefteq} \{\mu(r) \mid v \in \Gamma^* : \gamma_i \in v, r \in \mathrm{Runs}(\mathcal{K}, q, v)\} \mid i \in [k] \Big\} & \text{otherwise} \end{cases}$$

■

Theorem 71. *Let $(\mathcal{K}, \mu)$ be an W-weighted PDA and $\trianglelefteq$ be a partial order on W such that $w \trianglelefteq w \cdot w'$ for each $w, w' \in W$. The $\trianglelefteq$-heuristic for the exploration of $(\mathcal{K}, \mu)$ is monotonic and admissable.*

Proof. **Admissability.** The heuristic for the exploration of weighted push-down automata is admissable if, for every search item, the heuristic is less or equal than the least weight of a run starting with the state and the push-down storage of the search item. Let h be the heuristic for the W-weighted push-down automaton $((Q, \Sigma, \Gamma, q_0, Q_F, T), \mu)$. For each $q \in Q, k \in \mathbb{N}, \gamma_1, \ldots, \gamma_k \in \Gamma^*$ and $w \in \Sigma^*$, we show that

$$\begin{aligned} h(q, \gamma_1 \ldots \gamma_k, w) &= \begin{cases} \min\limits_{\trianglelefteq} \{\mu(\theta) \mid v \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q, v)\} & \text{if } k = 0 \\ \max\limits_{\trianglelefteq} \Big\{ \min\limits_{\trianglelefteq} \{\mu(\theta) \mid v \in \Gamma^* : \gamma_i \in v, \theta \in \mathrm{Runs}(\mathcal{K}, q, v)\} \mid i \in [k] \Big\} & \text{otherwise} \end{cases} \\ &\trianglelefteq \begin{cases} \min\limits_{\trianglelefteq} \{\mu(\theta) \mid \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma_1 \ldots \gamma_k)\} & \text{if } k = 0 \\ \min\limits_{\trianglelefteq} \{\mu(\theta) \mid v \in \Gamma^* : \gamma_1, \ldots, \gamma_k \in v, \theta \in \mathrm{Runs}(\mathcal{K}, q, v)\} & \text{otherwise} \end{cases} \\ &\trianglelefteq \begin{cases} \min\limits_{\trianglelefteq} \{\mu(\theta) \mid \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma_1 \ldots \gamma_k)\} & \text{if } k = 0 \\ \min\limits_{\trianglelefteq} \{\mu(\theta) \mid \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma_1 \ldots \gamma_k)\} & \text{otherwise} \end{cases} \\ &= \min_{\trianglelefteq} \{\mu(\theta) \mid \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma_1 \ldots \gamma_k)\}. \end{aligned}$$

Monotonicity. The heuristic is monotonic if, for every search item (q, v, w) and each of its successors

$(q', v', w\sigma)$ holds $h(q, v, w)) \trianglelefteq \mu(q, \sigma, r, q') \cdot h(q', v', w\sigma)$ if $\tau = (q, \sigma, r, q') \in T$ is a transition such that $v' \in r(v)$. Because of the case distinction in the definition of h, we consider the following cases:

1. $v = v' = \varepsilon$.

$$\begin{aligned}
&\min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \\
&\trianglelefteq \mu(\tau) \cdot \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} && \text{(by Corollary 11)}
\end{aligned}$$

2. $v = \varepsilon$ and $v' \neq \varepsilon$, thus $r = \mathrm{push}(\gamma)$ and $v' = \gamma$ for some $\gamma \in \Gamma$.

$$\begin{aligned}
&\min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \\
&\trianglelefteq \mu(\tau) \cdot \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} && \text{(by Corollary 11)} \\
&\trianglelefteq \mu(\tau) \cdot \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \\
&= \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in v' \Big\} && (v' = \gamma)
\end{aligned}$$

3. $v \neq \varepsilon$ and $v' = \varepsilon$, thus $r = \mathrm{pop}(\gamma)$ and $v = \gamma$ for some $\gamma \in \Gamma$.

$$\begin{aligned}
&\max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in v \Big\} \\
&= \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} && (v = \gamma) \\
&\trianglelefteq \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma\bar{v})\} \\
&\trianglelefteq \min_{\trianglelefteq} \{\mu(\tau) \cdot \mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} && (\tau = (q, \sigma, r, q'), \text{ Definition 9}) \\
&= \mu(\tau) \cdot \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^*, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\}
\end{aligned}$$

4. $v \neq \varepsilon$ and $v' \neq \varepsilon$. We will further consider the two cases

   (a) $r = \mathrm{push}(\gamma)$ or $r = \mathrm{id}_{\Gamma *}$, thus $v' = \hat{\gamma}v$ where $\gamma \in \Gamma$ and $\hat{\gamma} \in \{\gamma, \varepsilon\}$

$$\begin{aligned}
&\max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in v \Big\} \\
&\trianglelefteq \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\tau) \cdot \mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in v \Big\} \\
&\qquad\qquad (\tau = (q, \sigma, r, q'), \text{ Definition 9}) \\
&= \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in v \Big\} \\
&\trianglelefteq \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in \hat{\gamma}v \Big\} \\
&= \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in v' \Big\} && (v' = \hat{\gamma}v)
\end{aligned}$$

   (b) $r = \mathrm{pop}(\gamma)$ or $r = \mathrm{replace}(\gamma, \bar{\gamma})$, thus there is some $x \in \Gamma^*$ such that $v = \gamma x$ and $v' = \hat{\gamma}x$ where $\gamma, \bar{\gamma} \in \Gamma$ and $\hat{\gamma} \in \{\gamma', \varepsilon\}$

$$\max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in v \Big\}$$

$$= \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in \gamma x \Big\} \qquad (v = \gamma x)$$

$$= \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v} \vee \gamma \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in x \Big\}$$

$$\trianglelefteq \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v} \wedge \gamma \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \bar{v})\} \mid \gamma' \in x \Big\}$$

$$\trianglelefteq \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q, \gamma\bar{v})\} \mid \gamma' \in x \Big\}$$

$$\trianglelefteq \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\tau) \cdot \mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in x \Big\}$$

$$(\tau = (q, \sigma, \mu, \mu', q'), \text{ Definition 9})$$

$$= \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in x \Big\}$$

$$\trianglelefteq \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in \hat{\gamma} x \Big\}$$

$$= \mu(\tau) \cdot \max_{\trianglelefteq} \Big\{ \min_{\trianglelefteq} \{\mu(\theta) \mid \bar{v} \in \Gamma^* : \gamma' \in \bar{v}, \theta \in \mathrm{Runs}(\mathcal{K}, q', \bar{v})\} \mid \gamma' \in v' \Big\} \qquad (v' = \hat{\gamma} x)$$

□

4.4 Sorted multiple Dyck languages

We recall the construction of the tree-stack automaton with respect to an alphabet Σ and a partition $\mathfrak{P}$ to recognize the multiple Dyck language $\mathrm{mD}(\Sigma, \mathfrak{P})$ described in Section 3.5. We introduced several instructions, among which was the instruction up which introduced nondeterminism into the constructed automaton since it could move the tree-stack pointer to any child position. Using this nondeterminism, we deal with the possibility that the same set of brackets in $\mathfrak{P}$ needs to be cancelled multiple times next to each other.

If we recall the structure of bracket words in $R \cap D$, each word represents an abstract syntax tree of an MCFG. Each rule in this tree introduces two *layers* of brackets into our word,

- an opening and closing bracket for each composition representation's component of a rule (*component brackets*), and
- opening and closing brackets for each terminal (*terminal brackets*) in the composition representation and around each variable (*variable brackets*).

This structure repeats recursively, the component brackets surround all terminal brackets and variable brackets associated to the component of the rule, and each pair of variable brackets surround exactly one pair of component brackets of the represented child derivation. We also note that we unambiguously associate each set of variable brackets in $\mathfrak{P}$ with the variables of one successor in one grammar rule. Thus, the only sets of brackets in $\mathfrak{P}$ that may be cancelled multiple times in the same layer are terminal brackets.

Denkinger [Den17b, Lemma 5.24] already suggested that these characteristics in the structure of our bracket words may enable us to recognize them more easily. In this section, we will exploit the observations above to modify the construction of the tree-stack automaton from Section 3.5 such that the constructed automaton is deterministic.

Definition 72 (sort-consistence)**.** Let (Σ, sort) be a sorted set and $\mathfrak{P}$ a partition of Σ. We call $\mathfrak{P}$ *sort*-consistent if, for each $k \in \mathbb{N}$, $\{\sigma_1, \ldots, \sigma_k\} \in \mathfrak{P} \Rightarrow \mathit{sort}(\sigma_1) = \ldots = \mathit{sort}(\sigma_k)$. ■

For each sorted set $(\Sigma, sort)$ and $sort$-consistent partition $\mathfrak{P}$ of Σ, we will use the abbreviation $sort(\{\sigma_1, \ldots, \sigma_k\}) = sort(\sigma_1) = \ldots = sort(\sigma_k)$ for each $k \in \mathbb{N}$ and $\{\sigma_1, \ldots, \sigma_k\} \in \mathfrak{P}$.

Definition 73 (sorted multiple Dyck languages)**.** Let $(\Sigma, sort)$ be a sorted alphabet and $\mathfrak{P}$ a $sort$-consistent partition of Σ. We define the *sorted multiple Dyck language over* $(\Sigma, sort)$ *with respect to* $\mathfrak{P}$ $\mathrm{smD}(\Sigma, sort, \mathfrak{P}) \subseteq (\Sigma \cup \overline{\Sigma})^*$ recursively

- $\varepsilon \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$
- $\sigma_1 v_1 \overline{\sigma_1} \sigma_2 v_2 \overline{\sigma_2} \ldots \sigma_n v_n \overline{\sigma_n} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$ for each $\sigma_1, \ldots, \sigma_n \in \Sigma$ and $v_1, \ldots, v_n \in \mathrm{D}(\Sigma)$ iff there is a partition $\mathfrak{B}$ of $[n]$ such that for each $\{m_1, \ldots, m_k\} \in \mathfrak{B}$ where $k \in [n]$ and $m_1 < \ldots < m_k$ holds
  - $\{\sigma_{m_1}, \ldots, \sigma_{m_k}\} \in \mathfrak{P}$, and
  - for each $\{m'_1, \ldots, m'_{k'}\} \in \mathfrak{B}$ with $k' \in [n]$ holds $sort(\{\sigma_{m_1}, \ldots, \sigma_{m_k}\}) = sort(\{\sigma_{m'_1}, \ldots, \sigma_{m'_{k'}}\}) \iff \{m_1, \ldots, m_k\} = \{m'_1, \ldots, m'_{k'}\}$, and
  - $v_{m_1} \ldots v_{m_k} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$. ■

In sorted multiple Dyck languages, we associate each partition cell with exactly one sort and restrict Dyck languages such that for each sort only one set of brackets can be cancelled in each layer. This definition is very similar to the definition of $\mathfrak{P}$-*simple multiple Dyck words* by Denkinger [Den17b, Chapter 4, page 28].

Theorem 74. *Let $(\Sigma, sort)$ be a sorted alphabet and $\mathfrak{P}$ a sort-consistent partition of Σ. The sorted multiple Dyck language* $\mathrm{smD}(\Sigma, sort, \mathfrak{P})$ *is a subset of the multiple Dyck language* $\mathrm{mD}(\Sigma, \mathfrak{P})$.

Proof. Let $(\Sigma, sort)$ be a sorted alphabet and $\mathfrak{P}$ be a $sort$-consistent partition of Σ. We prove by induction that $v \in \mathrm{smD}(\Sigma, sort, \mathfrak{P}) \implies v \in \mathrm{mD}(\Sigma, \mathfrak{P})$.

Induction base Let $v = \varepsilon$. Clearly, $v \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$ and $v \in \mathrm{mD}(\Sigma, \mathfrak{P})$.

Induction hypothesis Let $n \in \mathbb{N}$, $\sigma_1, \ldots, \sigma_n \in \Sigma$ and $v_1, \ldots, v_n \in \mathrm{D}(\Sigma)$. We fix the sorted multiple Dyck word $\sigma_1 v_1 \overline{\sigma_1} \sigma_2 v_2 \overline{\sigma_2} \ldots \sigma_n v_n \overline{\sigma_n} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$. By Definition 73, there is a partition $\mathfrak{B}$ of $[n]$ such that, for all $\{m_1, \ldots, m_k\} \in \mathfrak{B}$ where $k \in [n], m_1 < \ldots < m_k$,

- $\{\sigma_{m_1}, \ldots, \sigma_{m_k}\} \in \mathfrak{P}$ and
- $v_{m_1} \ldots v_{m_k} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P})$ and
- for all $k' \in [n]$, $\{m'_1, \ldots, m'_{k'}\} \in \mathfrak{B}$: $sort(\{\sigma_{m'_1}, \ldots, \sigma_{m'_{k'}}\}) = sort(\{\sigma_{m_1}, \ldots, \sigma_{m_k}\}) \iff \{m'_1, \ldots, m'_{k'}\} = \{m_1, \ldots, m_k\}$.

We fix $\mathfrak{B}$ and, for all $k \in [n]$ and $\{m_1, \ldots, m_k\} \in \mathfrak{B}$, we chose $\sigma_1 v_1 \overline{\sigma_1} \sigma_2 v_2 \overline{\sigma_2} \ldots \sigma_n v_n \overline{\sigma_n}$ such that $v_{m_1} \ldots v_{m_k} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P}) \implies v_{m_1} \ldots v_{m_k} \in \mathrm{mD}(\Sigma, \mathfrak{P})$.

Induction step

$$
\begin{aligned}
&\sigma_1 v_1 \overline{\sigma_1} \sigma_2 v_2 \overline{\sigma_2} \ldots \sigma_n v_n \overline{\sigma_n} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P}) \\
\implies &\forall k \in [n], \{m_1, \ldots, m_k\} \in \mathfrak{B} \text{ such that } m_1 < \ldots < m_k\colon \\
&\{\sigma_{m_1}, \ldots, \sigma_{m_k}\} \in \mathfrak{P} \wedge v_{m_1} \ldots v_{m_k} \in \mathrm{smD}(\Sigma, sort, \mathfrak{P}) \\
&\wedge \forall k' \in [n], \{m'_1, \ldots, m'_{k'}\} \in \mathfrak{B}\colon sort(\{\sigma_{m'_1}, \ldots, \sigma_{m'_{k'}}\}) = sort(\{\sigma_{m_1}, \ldots, \sigma_{m_k}\}) \\
&\quad \iff \{m'_1, \ldots, m'_{k'}\} = \{m_1, \ldots, m_k\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Definition 73)}
\end{aligned}
$$

$$\begin{aligned}
&\implies \forall k \in [n], \{m_1, \ldots, m_k\} \in \mathfrak{B} \text{ such that } m_1 < \ldots < m_k: && \text{(induction hypothesis)}\\
&\quad \{\sigma_{m_1}, \ldots, \sigma_{m_k}\} \in \mathfrak{P} \wedge v_{m_1} \ldots v_{m_k} \in \mathrm{mD}(\Sigma, \mathfrak{P})\\
&\implies \forall k \in [n], \{m_1, \ldots, m_k\} \in \mathfrak{B} \text{ such that } m_1 < \ldots < m_k: && \text{(by Definition 29)}\\
&\quad \sigma_{m_1} v_{m_1} \overline{\sigma_{m_1}} \ldots \sigma_{m_k} v_{m_k} \overline{\sigma_{m_k}} \in \mathrm{mD}(\Sigma, \mathfrak{P})\\
&\implies \sigma_1 v_1 \overline{\sigma_1} \sigma_2 v_2 \overline{\sigma_2} \ldots \sigma_n v_n \overline{\sigma_n} \in \mathrm{mD}(\Sigma, \mathfrak{P}) && \text{([Den17b, Observation 4.4])}
\end{aligned}$$

□

Definition 75 (smD-tree-stack automaton)**.** Let $(\Sigma, sort)$ be an $\mathbb{N}$-sorted alphabet and $\mathfrak{P}$ a *sort*-consistent partition of Σ. For each $i \in \mathbb{N}$, we define the binary relation up_i where for each $(u, \alpha) \in \mathfrak{T}_{(\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma)}$,

$$\mathrm{up}_i(u, \alpha) = \begin{cases} \{(u, \alpha i)\} & \text{if } i \leq \mathrm{children}(u[\alpha]) \\ \{(push(u, \alpha, @^{i - \mathrm{children}(u[\alpha])}, \alpha i))\} & \text{otherwise.} \end{cases}$$

With the instructions from Definition 51 (setempty, rmsnd, chkfst, and down), we define the *smD tree-stack storage*

$$\mathrm{S}_{\mathrm{smD}}^{\Sigma, sort, \mathfrak{P}} = (\mathfrak{T}_{((\overline{\Sigma} \cup \{\varepsilon\}) \times \mathcal{P}(\Sigma))}, R, F, @)$$

where $F = \{(@(u_1, \ldots, u_k), \varepsilon) \mid k \in \mathbb{N}, u_1, \ldots, u_k \in \mathcal{U}_{\{(\varepsilon, \varnothing)\}}\}$ and

$$R = \{\mathrm{up}_{sort(\sigma)} \circ \mathrm{setempty}(\sigma, \mathfrak{P}[\sigma]) \circ \mathrm{rmsnd}(\sigma), \mathrm{down} \circ \mathrm{chkfst}(\overline{\sigma}) \mid \sigma \in \Sigma\}.$$

The *smD-tree-stack automaton with respect to* Σ *and* $\mathfrak{P}$ is the $\mathrm{S}_{\mathrm{smD}}^{\Sigma, sort, \mathfrak{P}}$-automaton $\mathcal{T}'(\Sigma, sort, \mathfrak{P}) = (\{q\}, \Sigma \cup \overline{\Sigma}, q, \{q\}, T)$ where

$$\begin{aligned} T = &\{(q, \sigma, \mathrm{up}_{sort(\sigma)} \circ \mathrm{setempty}(\sigma, \mathfrak{P}[\sigma]) \circ \mathrm{rmsnd}(\sigma), q) \mid \sigma \in \Sigma\} \\ &\cup \{(q, \overline{\sigma}, \mathrm{down} \circ \mathrm{chkfst}(\overline{\sigma}), q) \mid \overline{\sigma} \in \overline{\Sigma}\}. \end{aligned}$$

■

Theorem 76. *Let* $(\Sigma, sort)$ *be an* $\mathbb{N}$*-sorted alphabet and* $\mathfrak{P}$ *a sort-consistent partition of* Σ*. The smD-tree-stack automaton* $\mathcal{T}'(\Sigma, sort, \mathfrak{P})$ *with respect to* Σ *and* $\mathfrak{P}$ *is a deterministic* $\mathrm{S}_{\mathrm{smD}}^{\Sigma, sort, \mathfrak{P}}$*-automaton.*

Proof. Let $(\Sigma, sort)$ be an $\mathbb{N}$-sorted alphabet and $\mathfrak{P}$ a *sort*-consistent partition of Σ. From the construction of the transitions in $\mathcal{T}'(\Sigma, \mathfrak{P})$ in Definition 75, it is clear that there is exactly one state and one transition for each symbol in $\Sigma \cup \overline{\Sigma}$. Furthermore, all used instructions that were introduced in Definitions 51 and 75 are defined such that there is at most one element in the image of the instruction for each tree-stack. Thus, the compositions of the instructions used in the transitions of $\mathcal{T}'(\Sigma, sort, \mathfrak{P})$ will also relate each tree-stack to at most one other tree-stack. □

Theorem 77. *Let* $(\Sigma, sort)$ *be an* $\mathbb{N}$*-sorted alphabet and* $\mathfrak{P}$ *a sort-consistent partition of* Σ*. The language of the* $\mathrm{S}_{\mathrm{smD}}^{\Sigma, sort, \mathfrak{P}}$*-automaton* $\mathcal{T}'(\Sigma, sort, \mathfrak{P})$ *is the sorted multiple Dyck language over* Σ *with respect to* $\mathfrak{P}$*,* $\mathrm{L}(\mathcal{T}'(\Sigma, sort, \mathfrak{P})) = \mathrm{smD}(\Sigma, sort, \mathfrak{P})$*.*

Definition 78 (sorted multiple Dyck language with respect to an MCFG)**.** Let $G = (N, \Sigma, P, S)$ be non-deleting MCFG, Δ the generator alphabet with respect to G, and $\mathfrak{P}$ the partition of Δ with respect to G. We define the alphabet $\Delta' \subseteq \Delta$ by $\Delta' = \Delta \setminus \{\langle_\sigma \mid \sigma \in \Sigma\}$, and the function $sort \colon \Delta \to \mathbb{N}$ for each

$\delta \in \Delta'$ by

$$sort(\delta) = \begin{cases} 1 & \text{if } \delta \text{ is of the form } \langle_\rho^j \text{ for some } \rho \in P, j \in \text{fanout}(\rho) \\ i & \text{if } \delta \text{ is of the form } \langle_{\rho,i}^j \text{ for some } \rho \in P, i \in \text{rank}(\rho), j \in \text{fanout}_i(\rho). \end{cases}$$

We call $\text{smD}(\Delta', sort, \mathfrak{P} \setminus \{\{\langle_\sigma\} \mid \sigma \in \Sigma\})$ the *sorted multiple Dyck language with respect to* G and denote it by $\widehat{D}$ if G is clear from the context.

Furthermore, we define the alphabetic homomorphism $\text{rmterm}\colon (\Delta \cup \overline{\Delta})^* \to (\Delta' \cup \overline{\Delta'})^*$ where, for each $n \in \mathbb{N}$ and $\delta_1, \ldots, \delta_n \in \Delta$, $\text{rmterm}(\delta_1 \ldots \delta_n) = h_{\text{term}}(\delta_1) \cdot \ldots \cdot h_{\text{term}}(\delta_n)$ and

$$h_{\text{term}}(\delta) = \begin{cases} \varepsilon & \text{if } \delta \text{ is of the form } \langle_\sigma \text{ or } \rangle_\sigma \text{ for some } \sigma \in \Sigma \\ \delta & \text{otherwise.} \quad \blacksquare \end{cases}$$

Theorem 79. *Let $G = (N, \Sigma, P, S)$ be non-deleting MCFG, R the generator language with respect to G, D the multiple Dyck language with respect to G, and $\widehat{D}$ the sorted multiple Dyck language with respect to G.*

Then,

$$v \in R \cap D \iff \text{rmterm}(v) \in \text{rmterm}(R) \cap \widehat{D}.$$

Proof idea. Denkinger [Den17b, Lemma 5.24] proved a very similar statement. As previously mentioned, the definition of $\mathfrak{P}$-simplicity [Den17b, Chapter 4, page 28] is very similar to our definition of sorted multiple Dyck languages. But it includes an exception for reoccurring symbols $\sigma \in \Sigma$ where $|\mathfrak{P}[\sigma]| = 1$. We omit this exception, but exclude these symbols using the homomorphism rmterm. Thus, we assume that

$$v \in R \cap D \implies \text{rmterm}(v) \in \widehat{D} \tag{4.1}$$

holds.

Part 1: "$\Longrightarrow$"

$$\begin{aligned} & v \in R \cap D \\ \iff & v \in R \cap D \wedge \text{rmterm}(v) \in \text{rmterm}(R) \\ \implies & \text{rmterm}(v) \in \widehat{D} \wedge \text{rmterm}(v) \in \text{rmterm}(R) && \text{(by Eq. (4.1))} \\ \iff & \text{rmterm}(v) \in \widehat{D} \cap \text{rmterm}(R) \end{aligned}$$

Part 2: "$\Longleftarrow$"

As already mentioned in Section 3.2, the first and the last symbol in each sequence of $\widehat{\Delta}$ is unique in all sequences of $\widehat{\Delta}$. Thus, rmterms is a bijection from R to $\text{rmterm}(R)$. Furthermore, rmterm only removes brackets of the form $\langle_\sigma$ and $\rangle_\sigma$ for some terminal symbol σ. Since those brackets occur only next to each other in R, we suppose

$$v \in R \wedge \text{rmterm}(v) \in D \implies v \in R \wedge v \in D \tag{4.2}$$

$$\begin{aligned} & \text{rmterm}(v) \in \widehat{D} \cap \text{rmterm}(R) \\ \implies & \text{rmterm}(v) \in D \cap \text{rmterm}(R) \\ \iff & \text{rmterm}(v) \in D \wedge \text{rmterm}(v) \in \text{rmterm}(R) \\ \iff & \text{rmterm}(v) \in D \wedge v \in R && \text{(rmterm is a bijection)} \end{aligned}$$

$$\implies v \in D \wedge v \in R \qquad \text{(by Eq. (4.2))}$$
$$\iff v \in D \cap R \qquad \square$$

By Observation 67 and Theorem 79, we are able to alter the parsing algorithm by Denkinger [Den17b, Definition 5.21] to

$$\mathrm{parse}_{k,\trianglelefteq}(G, \mu, w) = \mathrm{take}(k) \circ \mathrm{map}(\mathrm{toderiv}) \circ \mathrm{filter}(\widehat{D}) \circ \mathrm{map}(\mathrm{rmterm}) \circ \mathrm{ordered}_{\trianglelefteq}(\widehat{R^{local}}).$$

Chapter 5

Implementation

In this chapter, we will have a close look on the most important parts of the implementation of our approach for *k-best Chomsky-Schützenberger parsing for weighted multiple context-free grammars.* [1]

We implemented our parser in *Rust*, an imperative systems programming language. It features algebraic data types, type parameters, and type classes that are called *traits*. As usual in imperative languages, we implement *lazy evaluation* with iterators. In Rust, we achieve this by implementing the `Iterator` trait.

In this chapter we will use the following type parameters by convention:

- `T` for terminal symbols,
- `NT` for nonterminal symbols,
- `W` for weights, and
- `Q` for states.

Very often, we will use integerized versions of data structures to speed up comparisons and keep a small memory footprint. Thus, we use `HashIntegerisers` to store a bijection of a data type to integers.

A significant amount of the constructions and definitions described in Chapters 3 and 4 only depend on a weighted grammar. So, we can prepare those constructions before knowing the word we want to parse. Specifically, this applies to the generator automaton that recognizes R, the tree-stack automaton that recognizes the multiple Dyck language D, and the homomorphism h. We store these constructions in a data type that can be persistently saved in a binary file using Rust's *serde framework*.

In this chapter, we will start with our implementation of a weighted graph search in Section 5.1 where we will introduce the data structure `Search` that implements the `Iterator` trait to enumerate all nodes in a weighted graph. Since we will need it in a variety of different scenarios, it is the first part of the implementation we are inspecting. Section 5.2 deals with the implementation of *deterministic finite state automata* and *deterministic push-down automata*. After that, we will have a look on the implementation of our generator automata in Section 5.3, and our filter automata in Section 5.4. We will wrap up our implementations with the *C-S-parsing module* in Section 5.5.

[1] The whole implementation of our parser in Rustomata is available at `https://github.com/truprecht/rustomata`.

5.1 Search

For many tasks, including word generation using automata or the calculation of heuristics, we need to implement an instance of a *graph search*. To avoid repeating implementations for that purposes, we decided to implement a generic interface for these tasks.

General workflow. A `Search` is an iterator that yields the lowest-weighted element of its `Agenda` according to `I`'s implementation of the `Ord` trait and puts weighted successor elements of it into the same `Agenda`. The set of successors for each element is determined by a function that is stored in each `Search`. We use the type parameters

- `A` for the type of the agenda,
- `I` for the type of search items, and
- `F` for the type of the closure that yields the successors of each item.

```
1  /// Implements a `Search` in a graph that is defined by a set of elements and a
   ↪  successor function.
2  pub enum Search<A, I, F>
3  where
4      A: Agenda<Item = I>,
5      F: FnMut(&I) -> Vec<I>,
6  {
7      /// Yield elements twice if there are multiple paths to them.
8      All(A, F),
9      /// Stores the set of all expanded elements to not yield them multiple times.
10     Uniques(A, F, BTreeSet<I>),
11 }
```

Implementation details. A `Search` has two variants, `Search::All` and `Search::Uniques`. In contrast to `Search::All`, `Search::Uniques` will save all visited items of type `I` in a set such that they are visited at most once. We will be able to switch between these two variants using the method `uniques` that adds an empty set to the contents of an `Seach::All` variant and puts it into a new `Search::Uniques`. And vice versa, there is the `all` method that just drops the set and converts a `Search::Uniques` into a `Search::All` variant.

The implementation of the `Iterator` trait is shown in the following listing. If the `self` is the `Search::All` variant (lines 10 to 18), we dequeue the best-weighted item from our agenda (line 11) and enqueue all successor items in our agenda (lines 12 to 14). After that, we yield the item we just dequeued (line 15). If `self` is a `Search::Uniques` additionally insert it in our set (line 22) and filter the successor items according to the set (line 23).

```
1  impl<A, I, F> Iterator for Search<A, I, F>
2  where
3      I: Clone + Ord,
4      A: Agenda<Item = I>,
5      F: FnMut(&I) -> Vec<I>,
```

```
{
    type Item = I;
    fn next(&mut self) -> Option<Self::Item> {
        match *self {
            Search::All(ref mut agenda, ref mut succ) => {
                if let Some(item) = Agenda::dequeue(agenda) {
                    for succ_item in (succ)(&item) {
                        agenda.enqueue(succ_item);
                    }
                    return Some(item);
                }
                None
            }

            Search::Uniques(ref mut agenda, ref mut succ, ref mut found) => {
                while let Some(item) = Agenda::dequeue(agenda) {
                    if found.insert(item.clone()) {
                        for succ_item in (succ)(&item).into_iter().filter(|i|
↪   !found.contains(i)) {
                            agenda.enqueue(succ_item);
                        }
                        return Some(item);
                    }
                }
                None
            }
        }
    }
}
```

Currently, there are two constructors for `Search`. The first one, `Search::weighted`, initializes a `Search::All` object with a `PriorityQueue` as agenda. The `PriorityQueue` is constructed from a sequence of items `init` (lines 5 and 8) and a weight assignment `weight` that is passed to the queue (line 5).

```
pub fn weighted<C>(init: C, successors: F, weight: Box<Fn(&I) -> W + 'a>) -> Self
where
    C: IntoIterator<Item = I>,
{
    let mut agenda = PriorityQueue::new(Capacity::Infinite, weight);

    for item in init {
        agenda.enqueue(item);
    }

    Search::All(agenda, successors)
```

```
}
```

The second constructor, `Search::unweighted`, initializes a `Search::All` object with a push-down storage (implemented as a `Vec`) as agenda. Therefore it omits the weight assignment and just collects the `Vec` from the iterator over all initial items (line 5).

```
pub fn unweighted<C>(init: C, successors: F) -> Self
where
    C: IntoIterator<Item = I>,
{
    Search::All(init.into_iter().collect(), successors)
}
```

The `PriorityQueue` data structure implements a limitation of its contents to a maximum number of elements. This enables us an easy way to use beam search as an approximation method. If the `Search` as initialized using the `weighted` constructor then we just need to pass a specified capacity to the `PriorityQueue`.

```
/// Switch to beam search with a beam `width`.
pub fn beam(mut self, width: Capacity) -> Self {
    if let Capacity::Limit(b) = width {
        match &mut self {
            &mut Search::All(ref mut a, _) | &mut Search::Uniques(ref mut a, _, _) =>
↪   {
                a.set_capacity(b);
            }
        }
    }

    self
}
```

5.2 Automata

For the construction of the automata described in Sections 3.2, 3.3, 4.1 and 4.2, we provide implementations for weighted deterministic finite state automata and weighted deterministic push-down automata in *Rustomata*. Both implementations are optimized with respect to the best-first enumeration of words in their language. Furthermore, we supply fast methods to intersect both automata with unweighted finite state automata.

5.2.1 Implementation of an OpenFst interface in Rust

OpenFst is an open-source framework for weighted finite state transducers introduced by Allauzen, Riley, Schalkwyk, Skut, and Mohri [ARSSM07]. To enable an efficient implementation of the construction, storage, intersection and evaluation of finite state automata, we planned to use an interface of OpenFst for the generator and filter automata.

Interfacing C++ and Rust. Rust offers a *foreign function interface* to the programming language *C*. Therefore, we can access functions and data structures with a C-compatible interface in Rust. Since OpenFst is a library written in *C++*, it offers no compatible interface, e.g. it uses templates and classes which are not available in C. For example, a variety of OpenFst's functions use *templates* and *inheritance hierarchies* to implement an interface that is independent of the weight and the specific implementation of automata.

Although, C++ is able to offer a C-compatible function interface if no C-incompatible features are used in all function headers. Therefore, we implemented the library *OpenFsa* as an intermediate layer to a subset of OpenFst's functions for selected configurations (e.g. specific weights and implementations) of finite state automata.

Lack of a generative iterator. An important part of the implementation is the *best-first* enumeration of words recognized by an automaton. Since we did not find any functionality in OpenFst to iterate lazily over the set of all that words or runs of an automaton, we implemented a work-around that exploits the functions to generate the n-best runs (for some chosen $n \in \mathbb{N}$) of an automaton and the weighted product construction of automata.

In detail, a best-first iterator over *n-batches* of words recognized by a weighted finite state automaton $\mathcal{A}$ consists of $\mathcal{A}$ and an automaton $\mathcal{A}^{(n)}$ that contains the n-best runs of $\mathcal{A}$. $\mathcal{A}^{(n)}$ is of a specific structure:

- the transitions in $\mathcal{A}^{(n)}$ are acyclic,
- there is at most one transition leaving each non-initial state of $\mathcal{A}^{(n)}$, and
- there are exactly $\max\{n, |\mathrm{Runs}(\mathcal{A}^{(n)})|\}$ transitions leaving its initial state. Thus, each of them represents exactly one run of $\mathcal{A}$. Additionally, they are ordered by the weight of the run they represent.

Because of these properties, it is fairly easy to read off the runs of (and hence also the words recognized by) $\mathcal{A}^{(n)}$. In each step, we can yield all words recognized by $\mathcal{A}^{(n)}$ and modify $\mathcal{A}$ using $\mathcal{A}^{(n)}$ such that it does not contain the runs of $\mathcal{A}^{(n)}$ for the next iteration. [2] Since this modifications requires a determinization of $\mathcal{A}^{(n)}$ and a product construction of $\mathcal{A}$ with an automaton that recognizes the inverse language of $\mathcal{A}^{(n)}$, the performance of this implementation is not acceptable.

Drop of OpenFsa. Although we spent a lot of work for the *OpenFsa* library in Rust, we decided to drop the implementation from this project. Besides the missing generative functionality, we also decided to use push-down automata as it was already described in Section 4.1, and therefore, switched to an own implementation in Rustomata.

Still, *OpenFsa* provides some functionality of weighted finite state automata in Rust. Therefore, we published it as an open-source library. [3]

5.2.2 Finite state automata

With the constructions we described in Chapters 3 and 4, we have two basic use-cases for finite state automata:

- the product construction with an unweighted finite state automaton, and

[2] Since this implementation was for the use of deterministic finite state automata, each run corresponds to exactly one word that is recognized by the automaton.

[3] The source of the Rust library *OpenFsa* can be found at `https://github.com/truprecht/openfsa`.

- the enumeration of all words in the language of an FSA.

We implemented a the data type `FiniteAutomaton` in Rustomata that satisfies both needs and implements Rustomata's `Automaton` trait.

Implementation details. A `FiniteAutomaton` is a data structure for weighted deterministic finite state automata with integer states. We store the set of transitions for each automaton in a list of `IntMaps` (line 9). The index of each list entry corresponds to the state that the transitions leaves. Each entry of the `IntMap` at that index represents one transition reading its key as integerized symbol. The value of the entry contains the integerized target state and the weight of the transition.

```
1  /// A deterministic finite state automaton.
2  #[derive(Clone, Debug)]
3  pub struct FiniteAutomaton<T, W>
4  where
5      T: Eq + Hash,
6  {
7      pub initial: usize,
8      pub finals: Vec<usize>,
9      pub arcs: Vec<IntMap<(usize, W)>>,
10     pub labels: Rc<HashIntegeriser<T>>,
11 }
```

The method

```
1  fn intersect(&self, filter: FiniteAutomaton<T, ()>) -> Self { ... }
```

of `FiniteAutomaton` implements the product construction with unweighted deterministic finite state automata. In the following listing, we show the part of the function that enumerates the set of transitions of the product FSA. We use a `Search` to limit the set of states and the set of transitions to those elements that are reachable from the initial state of the product.

```
Search::unweighted(
    initial_arcs,
    |&(_, _, _, sto, oto, _)| {
        let mut successors = Vec::new();
        for (label, &(to, weight)) in self.arcs.get(sto).unwrap_or(&IntMap::default())
↪   {
            if let Some(&(to_, _)) = other.arcs.get(oto).and_then(|m| m.get(label)) {
                successors.push(
                    (sto, oto, *label, to, to_, weight)
                );
            }
    }
    successors
}).uniques()
```

The method

```rust
fn generate<'a>(self, beam: Capacity) -> Box<Iterator<Item = Vec<T>> + 'a> { ... }
```

returns an iterator over all words recognized by a weighted deterministic finite state automaton. As we can see in the next listing, it is implemented as a search as well. Starting with the initial state and the empty word, we explore each search item by finding all transitions that leave its state (line 5). Each of these transitions adds a new search item with the target state of the transition and the symbol of the transition appended to the word of the search item (lines 6 to 8). Furthermore, we compute the weight of the new item as the product old item's weight and the weight of the transition (line 8). In the end, we just need to filter for search items with a final state (line 16) and yield their contained word.

```rust
Search::weighted(
    initial_agenda,
    move |&(q, ref word, weight)| {
        let mut successors = Vec::new();
        for (label, &(to, w)) in arcs.get(q).unwrap_or(&IntMap::default()) {
            let mut word_: Vec<usize> = word.clone();
            word_.push(label.clone());
            successors.push((to, word_, weight * w))
        }
        successors
    },
    Box::new(move |&(ref q, _, w)| {
        (*heuristics.get(q).unwrap_or(&LogDomain::zero()) * w).pow(-1.0)
    }),
).beam(beamwidth)
    .filter(move |&(ref q, _, _)| finals.contains(q))
```

5.2.3 Push-down automata

As finite state automata, we use weighted deterministic push-down automata for the construction of a generator automaton as described in Section 3.2. Thus, we will cover very similar use cases as the implementation of finite state automata, namely the product construction of weighted deterministic push-down automata with unweighted finite state automata and the best-first enumeration of all words recognized by a weighted deterministic push-down automaton.

Implementation details. As we can see in the following, the actual definition of `PushDownAutomaton` is very similar to the definition of `FiniteAutomaton`.

```rust
pub struct PushDownAutomaton<T, W>
where
    T: Hash + Eq + Clone,
{
    pub initial: usize,
    pub finals: Vec<usize>,
    pub arcs: Vec<IntMap<(usize, W, PushDownInstruction<usize>)>>,
    pub labels: Rc<HashIntegeriser<T>>,
}
```

The difference between both is the `PushDownInstruction` type that declares the instruction that is applied to the push-down for each transition of the automaton. The next listing shows the implementation of a `PushDownInstruction`'s application to a push-down implemented as a `Vec`. If we need to pop an element (lines 13 an 15), we split the last element off the `Vec` and check if it is the right one. After we are sure that the method does not fail, we clone the push-down. If we push an element (lines 8 and 15), we simply use the `push` method of `Vec` to insert it as the last element in the cloned push-down (lines 10 and 18).

```
1  fn apply(&self, pd: &Vec<S>) -> Option<Vec<S>> {
2      match *self {
3          PushDownInstruction::Nothing => Some(pd.clone()),
4          PushDownInstruction::Add(s) => {
5              let mut pd_ = pd.clone();
6              pd_.push(s);
7              Some(pd_)
8          }
9          PushDownInstruction::Remove(s) => pd.split_last()
10             .and_then(|(&s_, f)| if s_ == s { Some(f.to_vec()) } else { None }),
11         PushDownInstruction::Replace(s, s_) => pd.split_last().and_then(|(&vs, f)| {
12             if vs == s {
13                 let mut pd_ = f.to_vec();
14                 pd_.push(s_);
15                 Some(pd_)
16             } else {
17                 None
18             }
19         }),
20     }
21 }
```

Since the definition and the core functionality of `PushDownAutomaton` is very similar to the one of `FiniteAutomaton`, the implementation of the product construction with unweighted deterministic finite state automata and the enumeration of recognized words is very similar as well. Thus, we omit further explanation.

5.2.4 Tree-stack automata

Rustomata already carries a variety of different structures that implement automata with generic capabilities to recognize words of a language. Specifically, it includes a data structure for tree-stack automata that is closely related to the one we use. We use it as the foundation of our specific implementation for tree-stack automata that recognize multiple Dyck languages.

General workflow. We introduce the structure `MultipleDyckAutomaton`[4] that is constructed from a partition of elements of type `T`. To recognize words, we implement the existing interface of the `Automaton`

[4] We chose this name to emphasize the difference to the existing `TreeStackAutomaton` in Rustomata.

trait in Rustomata. With an implementation of this trait, Rustomata offers the function `recognise` to recognize words of the language of an automaton.

Implementation. Before we inspect the definitions of `MultipleDyckAutomaton`, we will have a look on the structure `MultipleDyckInstruction` that implements the tree-stack instructions that were described in Sections 3.5 and 4.4. There are three variants for this type, `Up`, `UpAt` and `Down`. `Up` and `Down` are the instruction used in the construction of tree stack automata that recognize multiple Dyck languages, and `UpAt` and `Down` are the instructions in the construction of the tree-stack automaton that recognizes sorted multiple Dyck languages.

```
pub enum MultipleDyckInstruction<T: Ord> {
    /// Nondeterministic `Up` for multiple Dyck languages.
    /// Returns a tree stack for each child with a set that contains the symbol and
    /// a tree stack with a freshly pushed child node.
    Up(T, BTreeSet<T>),
    /// Moves up if the child at the specified position
    /// * is vacant, or
    /// * contains the symbol in its set.
    UpAt(usize, T, BTreeSet<T>),
    /// Moves doum.
    Down(T),
}
```

`MultipleDyckInstruction` implements the `Instruction` trait of Rustomata. Its `apply` method is the application of the corresponding tree-stack instruction to a tree-stack. To handle nondeterministic tree-stack instructions (i.e. instructions that relate one tree stack to more than one tree-stacks), it will return multiple results.

```
impl<T: Clone + Ord> Instruction for MultipleDyckInstruction<T> {
    type Storage = TreeStack<MDTreeElem<T>>;

    fn apply(&self, ts: TreeStack<MDTreeElem<T>>) -> Vec<TreeStack<MDTreeElem<T>>> {
    ↪ ... }
}
```

The definition of the data structure `MultipleDyckAutomaton` itself consists of a set of `Transitions`, a structure provided by Rustomata. Each `Transition` holds a sequence of terminals, a weight and a `MultipleDyckInstruction`. The `MultipleDyckAutomaton` implements Rustomata's `Automaton` trait that defines associated types for its storage, instructions on its storage, initial and accepting elements of its storage, and a way to access the transitions. Implementing this trait enables us to use the function `recognise` that enumerates all runs in an `Automaton`.

Lastly, we wrapped the `MultipleDyckAutomaton` into the `MultipleDyckLanguage` data structure to abstract from its representation as an automaton. This data structure has two constructors,

- `new` that uses a partition $\mathfrak{P}$ to construct a `MultipleDyckAutomaton` that recognizes the multiple Dyck language over $\bigcup_{\mathfrak{p}\in\mathfrak{P}} \mathfrak{p}$ with respect $\mathfrak{P}$, and

- `sorted` that uses a partition $\mathfrak{P}$ and function f to construct a `MultipleDyckAutomaton` that recognizes the sorted multiple Dyck language over $(\bigcup_{\mathfrak{p}\in\mathfrak{P}} \mathfrak{p}), f)$ with respect to $\mathfrak{P}$.

Moreover it has the method `recognize` that returns true iff the stored automaton recognizes a given word. The three functions and the definition of `MultipleDyckLanguage` are shown in the following listing.

```
pub struct MultipleDyckLanguage<T: Ord + Clone>(MultipleDyckAutomaton<T>);

impl<'a, T: Clone + Eq + Ord> MultipleDyckLanguage<T> {
    /// Represents a multiple Dyck language with respect to
    /// a partition `p` = {p_1, ..., p_k} of an implicit alphabet p1 + ... + pk.
    pub fn new(p: Partition<T>) -> Self {
        MultipleDyckLanguage(MultipleDyckAutomaton::new(p))
    }

    /// Represents a sorted multiple Dyck language with respect to a
    /// partition `p` = {p_1, ..., p_k} of an implicit sorted alphabet (p1 + ... + pk,
    ↪ `sort`).
    pub fn sorted<F>(p: Partition<T>, sort: F) -> Self
    where
        F: Fn(&T) -> usize,
    {
        MultipleDyckLanguage(MultipleDyckAutomaton::sorted(p, sort))
    }

    /// Returns true iff `word` is a member of the multiple Dyck language.
    pub fn recognize(&self, word: &[Bracket<T>]) -> bool {
        let &MultipleDyckLanguage(ref mda) = self;
        let word_ = word.to_owned();
        let mut b = recognise(mda, word_);
        b.next().map(|item| item.0)
    }
}
```

5.3 Generator automata

In this section, we will describe the implementation of the different types of *generator automata* that were described in Sections 3.2 and 4.1. The major difficulty in this area is that we need to handle finite state automata and push-down automata the same way. For that purpose, we introduce the `GeneratorAutomaton` trait. It defines functions for the product construction with finite state automata and the enumeration of all words in the language of that automaton.

```
pub trait GeneratorAutomaton<T>
where
    T: Hash + Clone + Eq
```

```
{
    /// Size of the automaton (the number of transitions) for debugging purposes.
    fn size(&self) -> usize;

    /// Returns a pointer to the internal `Integeriser` that is used to integerize
    ↪ `BracketFragements`.
    fn get_integeriser(&self) -> Rc<HashIntegeriser<T>>;

    /// Constructs the intersection of the automaton with an unweighted finite state
    ↪ automaton.
    fn intersect(&self, other: FiniteAutomaton<T, ()>) -> Self
    where Self: Sized;

    /// Returns an `Iterator` that enumerates words recognized by this automaton.
    /// If `beam` is a `Capacity::Limit`, it uses beam search.
    fn generate<'a>(self, beam: Capacity) -> Box<Iterator<Item=Vec<T>> + 'a>
    where T: 'a;
}
```

Furthermore, there is a `GeneratorStrategy` trait that declares functions for the actual construction of a `GeneratorAutomaton`. It has an associated type for the specific type of the `GeneratorAutomaton`, i.e. if it is a finite state automaton or a push-down automaton.

Each data structure that implements the `GeneratorStrategy` trait may be used to extract a structure that implements `GeneratorAutomaton` from a non-deleting MCFG. In our implementation, we provide three data structures that implement the *GeneratorStrategy* trait,

- `FiniteGenerator` that constructs a `FiniteAutomaton` from a non-deleting MCFG as described in Section 3.2,
- `PushDownGenerator` that constructs a `PushDownAutomaton` from a non-deleting MCFG as described in Section 4.1, and
- `ApproxGenerator` that constructs the d-approximation of generator PDA automaton for some $d \in \mathbb{N}$.

Since each automaton constructed by one of these implementation is a `GeneratorAutomaton`, we are able to construct the product with a unweighted `FiniteAutomaton` and to enumerate each element of its language.

```
/// A `GeneratorStrategy` is a method to create a `GeneratorAutomaton` with respect to
↪ an LCFRS.
pub trait GeneratorStrategy<T>
where
    T: Clone + Hash + Eq
{
    type Generator: GeneratorAutomaton<BracketFragment<T>> + Serialize + for<'de>
↪ Deserialize<'de>;

```

```
    fn create_generator_automaton<'a, N, R>(&self, grammar_rules: R, initial: N,
    ↪ integeriser: &Integeriser<Item=PMCFGRule<N, T, LogDomain<f64>>>) ->
    ↪ Self::Generator
    where
        N: Hash + Ord + Clone + 'a,
        R: IntoIterator<Item=&'a PMCFGRule<N, T, LogDomain<f64>>>,
        T: 'a;
}
```

5.4 Filter automata

Similar to the `GeneratorStrategy` trait described in Section 5.3, there is a `FilterAutomaton` trait that contains the definition of two functions,

- a function `new` that uses an non-deleting MCFG to construct an object of the structure, and
- a function `fsa` that constructs a *FiniteAutomaton* from the structure that implements the trait and a sequence of terminals.

So, the data structures contain all information that is needed to construct the filter automaton (or inside filter automaton) using the `fsa` method. We can see the definition of this trait in the following listing with the type parameters

- `R` for an `Iterator` over the rules of the given grammar, and
- `I` for an `Integeriser` for the rules of the grammar (we use it to synchronize the integerization with other automata constructions).

```
pub trait FilterAutomaton<'a, T: 'a>
where
    T: Hash + Eq + Clone,
{
    /// Constructor.
    fn new<N: 'a, W: 'a, I, R>(
        grammar_rules: R,
        grammar: &I,
        reference_automaton: &GeneratorAutomaton<BracketFragment<T>>,
    ) -> Self
    where
        N: Hash + Eq + Clone,
        W: Eq + Clone,
        R: Iterator<Item=&'a PMCFGRule<N, T, W>>,
        I: Integeriser<Item=PMCFGRule<N, T, W>>;

    /// Extracts an unweighted finite state automaton from the object.
    fn fsa(
        &self,
```

```
20          word: &[T],
21          reference_automaton: &GeneratorAutomaton<BracketFragment<T>>,
22      ) -> FiniteAutomaton<BracketFragment<T>, ()>;
23  }
```

We provide two data structures that implement `FilterAutomaton`,

- `NaiveFilterAutomaton` that constructs a filter automaton as described in Section 3.3, and
- `InsideFilterAutomaton` that constructs the filter automaton from the set of productive grammar rules as described in Section 4.2.

Both structures also implement the `Serialize` trait of the serde framework which allows us to store them persistently.

We inspect the implementation of the `NaiveFilterAutomaton` a little further. It contains a `HashMap` that maps each terminal symbol σ in the alphabet of the grammar to the set of generator fragments that contain the bracket $\langle_\sigma$ in their second position (line 5). [5] Moreover, we store the list of all other terminals that occur in this generator fragment along with it. Also, each `NaiveFilterAutomaton` contains a list of generator fragments that does not contain any brackets with terminals of the grammar (line 6).

```
1   pub struct NaiveFilterAutomaton<T>
2   where
3       T: Eq + Hash,
4   {
5       brackets_with: HashMap<T, Vec<(Vec<T>, usize)>>,
6       epsilon_brackets: Vec<usize>,
7   }
```

The `fsa` method of the `NaiveFilterAutomaton` is shown in the following listing and implements the construction of the filter automaton as explained in Section 3.3. For each position in the word we want to parse (line 7), we find the set of generator fragments that are stored in our map for the symbol in that position (line 8). For each of those generator fragments, we compare the terminals that occur in this fragment with the following terminals in our word (line 9). If both sequences are equal, we construct a transition from the position before the first symbol to the position after the last symbol in the subsequence of the word (lines 11 to 15). Lastly, we construct loop transitions for generator fragments that do not contain any brackets with terminal symbols (lines 19 to 22 and 24 to 27).

```
1   fn fsa(
2       &self,
3       word: &[T],
4       reference: &GeneratorAutomaton<BracketFragment<T>>,
5   ) -> FiniteAutomaton<BracketFragment<T>, ()> {
6       let mut arcs = Vec::new();
7       for i in 0..(word.len()) {
8           for &(ref terminals, i_brackets) in
    ↪   self.brackets_with.get(&word[i]).unwrap_or(&Vec::new()) {
```

[5] Keep in mind that each generator fragment in $\widehat{\Delta}$ is of the form $\delta\langle_{\sigma_1}\rangle_{\sigma_1}\ldots\langle_{\sigma_k}\rangle_{\sigma_k}\delta'$ for some $\delta, \delta' \in \Delta$ and $\sigma_1, \ldots, \sigma_k \in \Sigma$ for some $k \in \mathbb{N}$.

```
            if word[i..(i + terminals.len())] == terminals[..] {
                arcs.push(
                    Transition {
                        instruction: StateInstruction(i, i+terminals.len()),
                        word: vec![i_brackets],
                        weight: ()
                    }
                );
            }
        }
        arcs.extend(self.epsilon_brackets.iter().map(
            |brackets|
            Transition{ instruction: StateInstruction(i, i), word: vec![*brackets],
↪   weight: ()}
        ));
    }
    arcs.extend(self.epsilon_brackets.iter().map(
        |brackets|
        Transition{ instruction: StateInstruction(word.len(), word.len()), word:
↪   vec![*brackets], weight: ()}
    ));
    FiniteAutomaton::from_integerized(
        arcs,
        0,
        vec![word.len()],
        Rc::clone(&reference.get_integeriser()),
    )
}
```

5.5 Chomsky-Schützenberger parsing

As mentioned in the beginning of Chapter 5, we added the `cs_parsing` module to Rustomata that wraps the previously described structures.

One of the data structures defined in `cs_parsing` is `CSRepresentation`. We use it to store a `GeneratorAutomaton`, a `FilterAutomaton`, and a sorted `MultipleDyckLanguage` that were extracted from a non-deleting MCFG. This data structure implements the `Serialize` and `Deserialize` trait that are provided by the serde framework and enable us to persistently store it in a file. Thus, we do not need to extract these constructions multiple times if we parse with the same grammar.

```
pub struct CSRepresentation<N, T, F, S>
where
    N: Ord + Hash + Clone,
    T: Ord + Hash + Clone,
    F: for<'a> FilterAutomaton<'a, T> + Serialize,
    S: GeneratorStrategy<T>,
```

```
{
    generator: S::Generator,
    rules: HashIntegeriser<PMCFGRule<N, T, LogDomain<f64>>>,
    filter: F,
    checker: MultipleDyckLanguage<BracketContent<T>>
}
```

This data structure also implements the method `generate` that, given a word w, returns an `Iterator` over the abstract syntax trees of the grammar that yield w.

```
pub fn generate(&self, word: &[T], beam: Capacity) -> CSGenerator<T, N> {
    let f = self.filter.fsa(word, &self.generator);
    let g = self.generator.intersect(f).generate(beam);

    CSGenerator {
        candidates: g,
        rules: &self.rules,
        checker: &self.checker
    }
}
```

For this purpose, we implemented the data structure `CSGenerator`. It stores an `Iterator` over the words recognized by the intersection of a `GeneratorAutomaton` and a `FilterAutomaton` (line 6), and a pointer to a sorted `MultipleDyckLanguage` (line 8).

```
pub struct CSGenerator<'a, T, N>
where
    T: 'a + PartialEq + Hash + Clone + Eq + Ord,
    N: 'a + Hash + Eq,
{
    candidates: Box<Iterator<Item = Vec<BracketFragment<T>>> + 'a>,
    rules: &'a HashIntegeriser<PMCFGRule<N, T, LogDomain<f64>>>,
    checker: &'a MultipleDyckLanguage<BracketContent<T>>
}
```

`CSGenerator` implements the `Iterator` trait by iterating over the stored words (line 14) and filtering for those words that are in the sorted `MultipleDyckLanguage` (line 19). After that, these filtered words are transformed into abstract syntax trees of our grammar (line 20).

```
impl<'a, N, T> Iterator for CSGenerator<'a, T, N>
where
    T: PartialEq + Hash + Clone + Eq + Ord,
    N: Hash + Eq + Clone,
{
    type Item = Derivation<'a, N, T>;

    fn next(&mut self) -> Option<Derivation<'a, N, T>> {
```

```
9           let &mut CSGenerator {
10              ref mut candidates,
11              rules,
12              checker
13          } = self;
14          for fragments in candidates {
15              let candidate: Vec<Delta<T>> =
    ↪ BracketFragment::concat(fragments).into_iter().filter(|b| match *b {
16                  Bracket::Open(BracketContent::Terminal(_)) |
    ↪ Bracket::Close(BracketContent::Terminal(_)) => false,
17                  _ => true
18              }).collect();
19              if checker.recognize(&candidate) {
20                  if let Some(derivation) = from_brackets(rules, candidate) {
21                      return Some(derivation);
22                  }
23              }
24          }
25
26          None
27      }
28  }
```

Chapter 6

Evaluation

This section deals with the evaluation of the implemented parser. In general, we will inspect the parser in terms of its accuracy and its parse time.

For the accuracy, we will use a variant of the *Parseval* measure that was introduced by Black et al. [Bla+91] to compare parse trees to the trees in the test corpus. We will describe it in Section 6.1 and use its implementation for discontinuous parse trees by Cranenburgh, Scha, and Bod [CSB16]. To obtain the parse time, we measure the time in the parser itself. So, we are able to start the measurement after all files are processed to collect unaltered time spans.

Section 6.2 will characterize the setup of our experiments.

And finally, Section 6.3 will show the results of the described experiments in comparison to two different published parsers, namely the *Grammatical Framework* by Ranta [Ran11] and *rparse* by Kallmeyer and Maier [KM13].

6.1 Metrics

To compare the effectiveness of our parsing approach, we compare the proposed parse tree (i.e. the result of our implementation) for each sentence in a test corpus with the gold parse tree for that sentence in the test corpus. We report the *labeled recall, precision and F1-score* as introduced by Collins [Col97, Section 4, page 21] and implemented in *Disco-Dop* [CSB16].

The three scores are based on the *Parseval* measure that was introduced by Black et al. [Bla+91]. It matches each *subtree* in a *parse tree* with a subtree in the corresponding *gold tree* by its *span*. We can then compute the *labeled precision* and *labeled recall* for each constituent as follows:

- *true positives* are all constituent-span pairs that match in the gold tree and in the candidate tree with the same label,
- *false positives* are all constituent-span pairs in the candidate parse tree that do not match with any span in the gold parse tree or have a match with a different label,
- *false negatives* are all constituent-span pairs in the gold parse tree that do not match with any span in the candidate parse tree or have a match with a different label, and finally,
- the *precision* is the number of true positives divided by the number of false positives and true positives and

- the *recall* is the number of true positives divided by the number of false negatives and true positives.

In the end, we may compute the *F1 score* as the harmonic mean of precision and recall.

Let us describe the matching and computation of the two scores by an example.

Example 80. Consider the two parse trees in Figure 6.1.

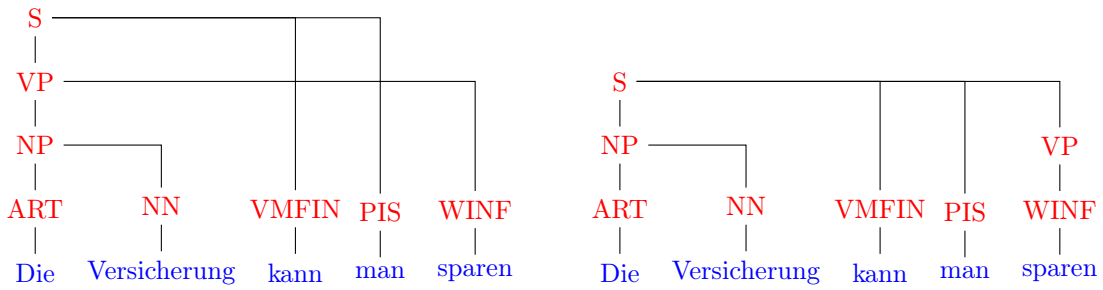


Figure 6.1: Two parse trees for the same german sentence [CSB16].

For each constituent, we can read off the span, i.e. the list of leaves from left to right below the node of the constituent as shown in Table 6.1. In this case, we do not consider the POS tags for the words in our sentences.

Table 6.1: Constituents and spans of the two parse trees shown in Figure 6.1.

first parse tree		second parse tree	
constituent	discontinuous span	constituent	discontinuous span
S	Die Versicherung kann man sparen	S	Die Versicherung kann man sparen
VP	Die Versicherung sparen	VP	sparen
NP	Die Versicherung	NP	Die Versicherung

As we can see in Table 6.1, we can match the span of the first and third constituent in both parse trees and we can not match the span of constituent of the first parse tree with any span in the second parse tree. Since the first and third constituents of the gold and candidate parse tree are also the same, we consider these two matches as *true positives*. Since the span of the second constituent of the candidate parse tree matches with no span of any constituent in the gold parse tree, we consider it as a *false positive*. And vice versa, the second span of the gold tree does not match with any span in the candidate parse tree. Thus it is a *false negative*.

Thus, in our example, the labeled recall and labeled precision are both $\frac{2}{3}$. We determine the F1 score with respect to our results as $\frac{2 \cdot \frac{2}{3} \cdot \frac{2}{3}}{\frac{2}{3}+\frac{2}{3}} = \frac{2}{3}$. ■

6.2 Experimental setup

For the evaluation of our implemented parser, we will use grammars (training sets) and sentences with their gold parse trees (test sets) extracted from the *NeGra* corpus [SUBK98]. All used grammars were extracted from subsets of NeGra using the implementation in *Vanda-Haskell* [DBD18] by Mielke [Mie15].

Due to our low expectations of the performance of the implemented parser, we limit the extracted training corpora to 250, 500, 750, 1 000, 1 250 and 1 500 parse trees from the NeGra corpus. In detail, we extract the training and test splits as follows:

Table 6.2: Overview over the size of all used corpus splits. We counted the sentences that contain up to 5, between 6 and 10, and more than 10 tokens separately.

size of the training set				size of the test set			
total	$\|w\| \leq 5$	$5 < \|w\| \leq 10$	$10 < \|w\|$	total	$\|w\| \leq 5$	$5 < \|w\| \leq 10$	$10 < \|w\|$
250	134	82	34	100	84	16	0
500	263	169	68	176	132	34	10
750	409	240	101	235	167	53	15
1 000	572	306	122	233	156	60	17
1 250	750	359	141	234	152	68	14
1 500	867	468	165	195	137	44	14

- we consider only those sentences (and their parse trees) that
  - contain less than 16 tokens, and
  - only consist of tokens that occur at least twice in NeGra,
- we choose a fixed amount of training parse trees from this filtered set,
- all remaining sentences (and gold parse trees) that contain only tokens that occur in the training sentences are the test set.

So, we ensure that the terminal symbols from the grammar extracted the training set contains all terminal symbols in the sentences of the corresponding test set. The sizes of all training and test splits are shown in Table 6.2.

For the *evaluation of beam search* in the enumeration of words in the language R^{local}, we use the training corpus that contains 500 parse trees and, for each sentence in the test set, measure the time that is spent enumerating words in R^w until the first parse tree was found. Furthermore, we report the amount of parses where at least one parse tree was found within 15 seconds. We repeat this experiment for the beam widths of 10, 100, 1 000 and 10 000. With the results that are discussed in Section 6.3.1, we decided to fix the beam width for all following experiments to 1 000.

We *compare the two generator automata* (generator FSA and generator PDA) using the training corpus that contains 500 sentences and its respective test corpus. We report the parse time (including the computation of the heuristics, the product constructions and the generation of parse candidates) and the rate of sentences where at least one parse tree was found within 15 seconds. If a parse tree was found, we also report the number of words in R^w that were enumerated until the first element in D was found.

Very similarly, we *compare the two filter automata* (filter automaton and inside filter automaton). We report the number of transitions of the constructed automata (filter automata and intersection automata), the time that was needed to construct the intersection and the parse time.

Since the performance of the implementation seemed to very dependent on the size of the used grammar in early tests (we were barely able to parse sentences with grammars extracted from 200 parse trees of NeGra), we also compare the parse time of our parser with grammars of different sizes. For each pair of training and test corpus as described above, we report the distribution of parse times and the ratio of sentences for that we were able to find a parse tree within 15 second. Moreover, we will compare the size of the constructed automata and the time needed to intersect the respective generator automaton with the filter automaton.

Lastly, we compare our implementation with two published parsers, *rparse* [KM13] and *Grammatical Framework* [Ran11]. We will use the largest training set of our corpus splits to parse the sentences of the test set with a grammar extracted from the training set. For our parser, we will extract a probabilistic non-deleting MCFG using Vanda-Haskell as for all other setups. Both other parsers use a probabilistic LCFRS as well, although we extract it using rparse in a representation that can be used by both other parsers. Using these grammars, we parse the sentences of the training set and report parse times and labeled recall, precision and f-scores with respect to the gold parse trees as explained in Section 6.1.

6.3 Results

6.3.1 Beam search

In Figure 6.2, we can see results for the evaluation of beam search used for the enumeration of words in R^{local}. As we described in Section 4.3, we introduced some heuristics for the exploration of the graph structure spanned by the set of transitions of finite state automata and push-down automata. With this heuristics, we were able to implement beam search by limiting the elements in the search agenda to a certain amount of items (*beam width*).

In the four plots for the ratio of sentences we were able to parse, we can see a positive effect with greater beam widths. As we are not able to parse sentences longer than eight tokens with a beam width of ten, the parser finds more results with a beam width of 100, and even more with 1000. We suppose that the results using the latter two beam widths show no difference at all.

Surprisingly, the results in the parse time in the second set of plots shows almost no difference for all four tested beam widths. We can only observe the absence of some data points in the first two plots since we were not able to find parses for some sentences.

With the result shown in these plots, we decided to continue our experiments with a beam width of 1000.

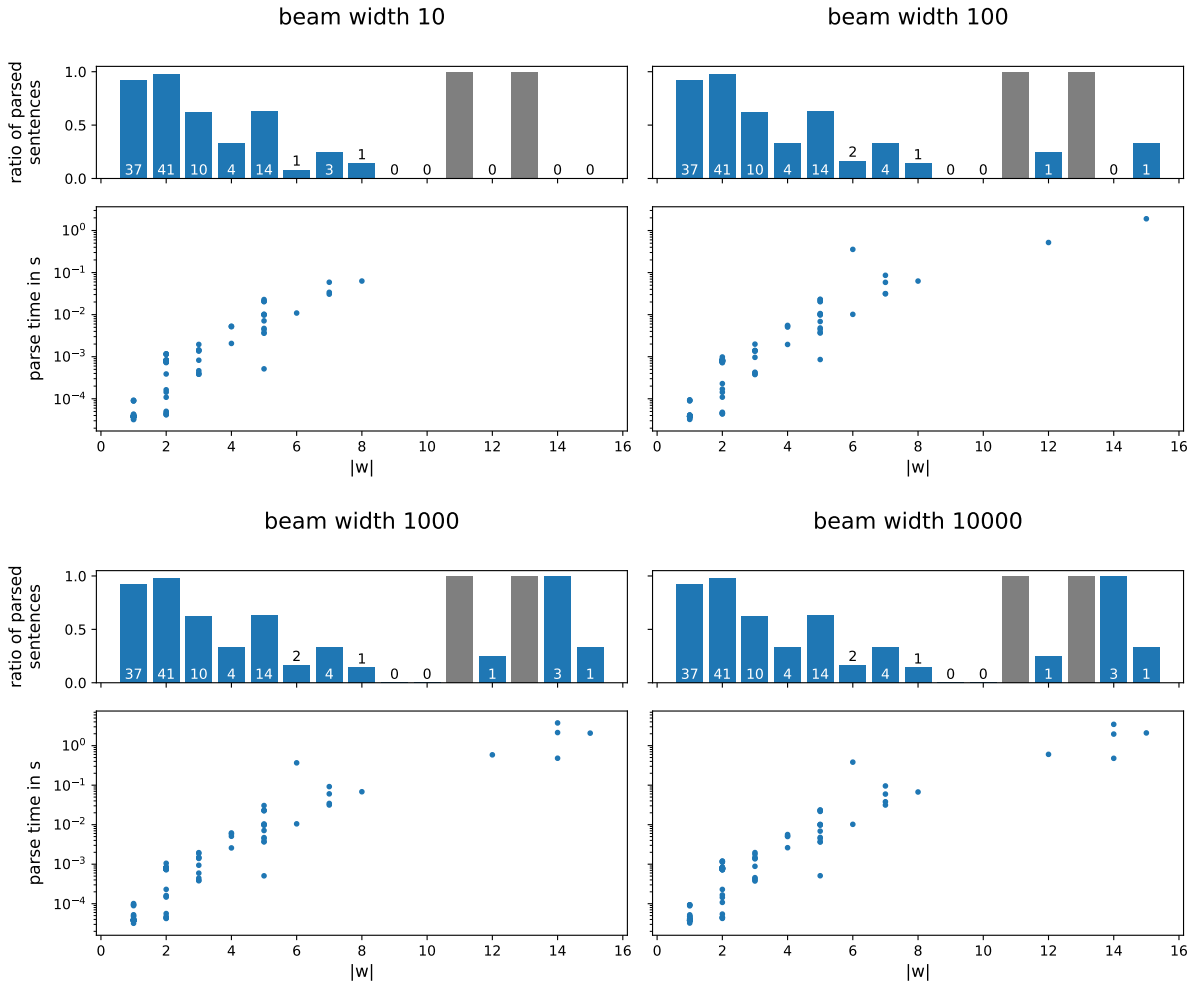


Figure 6.2: Comparison of different beam widths. We enumerated elements of R^{local} with beam search and different beam widths. The plots show the ratio of parsed sentences and the distribution of parse times for each beam width an each sentence length. Gray bars in the upper plots indicate the absence of sentences of the specific length in the test set.

6.3.2 Generator automata

We show the results of our experiments using the two described versions of the generator automaton in Figure 6.3. It shows

- the ratio of sentences for that a parse tree was found within 15 seconds
- the number of candidates in R^{local} we enumerated until we found the first element of D, and
- the parse time

for each generator automaton and length of the parsed sentences. Both versions used the same training and test corpora.

We can see that the version with the push-down automaton is able to find more parse trees than the version with the finite state automaton. Especially with larger sentences, the first version is not able to find parse trees within the time limit. We can see a possible reason for that in the second set of plots. With larger sentences, we apparently need to enumerate more words in R^w until the first multiple Dyck word in found. If we compare this plot with the third plot, we suppose that this amount of enumerated words is closely related to the parse time.

The second version of the generator automaton on the other hand, does not need to enumerate as large amounts of candidates as the first version. In our test set, we only needed to check at most three candidates until we found the first multiple Dyck word. Comparing this plot with the parse times in the third set of plots, we can not spot an obvious connection between those two distributions. If we compare the third set of plots of the two generators, we can see that the parse times for sentences up to a length of four tokens seems to be very similar.

Summarizing, we suppose that the generator PDA that we introduced in Section 4.1 is superior to the generator FSA that was described in Section 3.2.

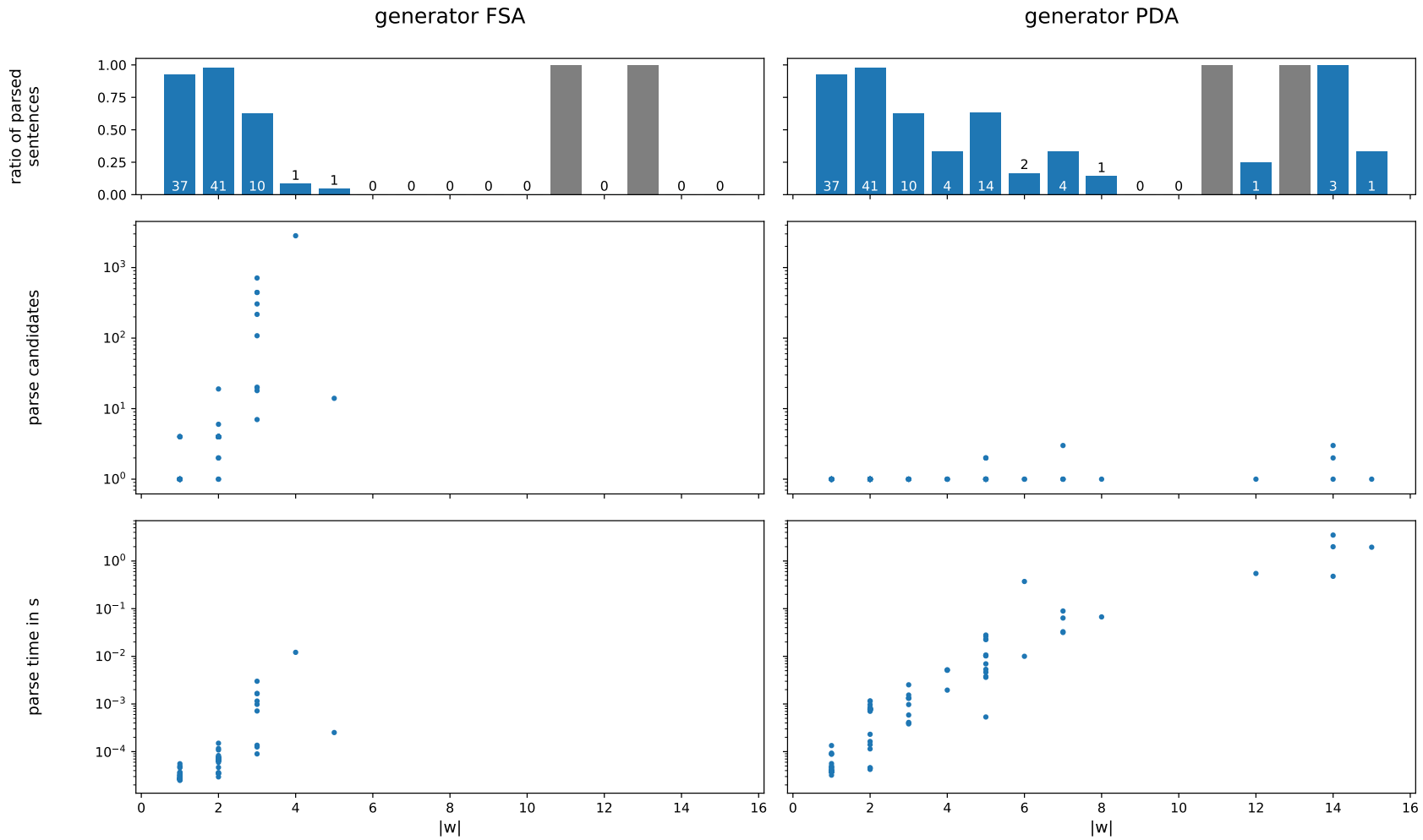


Figure 6.3: Comparison of both versions of the *generator automaton*. The plots show the ratio of sentences for which a parse tree was found, the number of candidates in R^{local} that were enumerated until the first word in $R^{local} \cap D$ was found, and the time needed to parse the sentence. The gray bars show sentence lengths for which no sentences were available in the test set.

6.3.3 Filter automata

In Figure 6.4, we can see the comparison of results using the filter automaton as it was described in Section 3.3 and the inside filter automaton as introduced in Section 4.2. For both versions of the filter automaton, we see four plots that show

- the ratio of the sentences for which we were able to find a parse tree,
- the size of the set of transitions of the respective filter automaton,
- the time needed to intersect this filter automaton with the generator PDA, and
- the size of the set of transitions of the resulting product of the filter automaton with the generator PDA

for both automata and for each sentence size. Both versions used the same training and test corpora as it was described in Section 6.2.

In the first set of plots, we can see that we were able to parse more sentences using the inside filter automaton. Although there seems to be no difference when we try to parse sentences up to eight tokens, we were only able to parse sentences of lengths larger than eight with the inside filter automaton.

In the next set of plots, we compare the size of the set of transitions in both constructed automata. We can see that the number of transitions in the first version of the filter automaton appears to grow linearly with the length of the sentence we want to parse. Since we construct the set of transitions with a repeating set of loops for each terminal symbol in the word we want to parse, this is plausible. The next two plots for the first version of the filter automaton show a very similar distribution of values. As the construction of the intersection of both automata consists mostly of the enumeration of transitions, this behavior is conceivable as well.

If we compare these three plots with the ones for the inside filter automaton, we can see that the set of transitions in the constructed inside filter automaton is smaller for each word in our test set. Similar to the filter automaton, the distribution of values in the latter three plots look very similar. We suppose that the size of the filter automaton has a very strong influence on the size of the intersection automaton and the time needed for its construction. Lastly, the last two sets of plots show that the time needed for the construction and the size of the set of transitions of the product automaton is always smaller using the inside filter automaton.

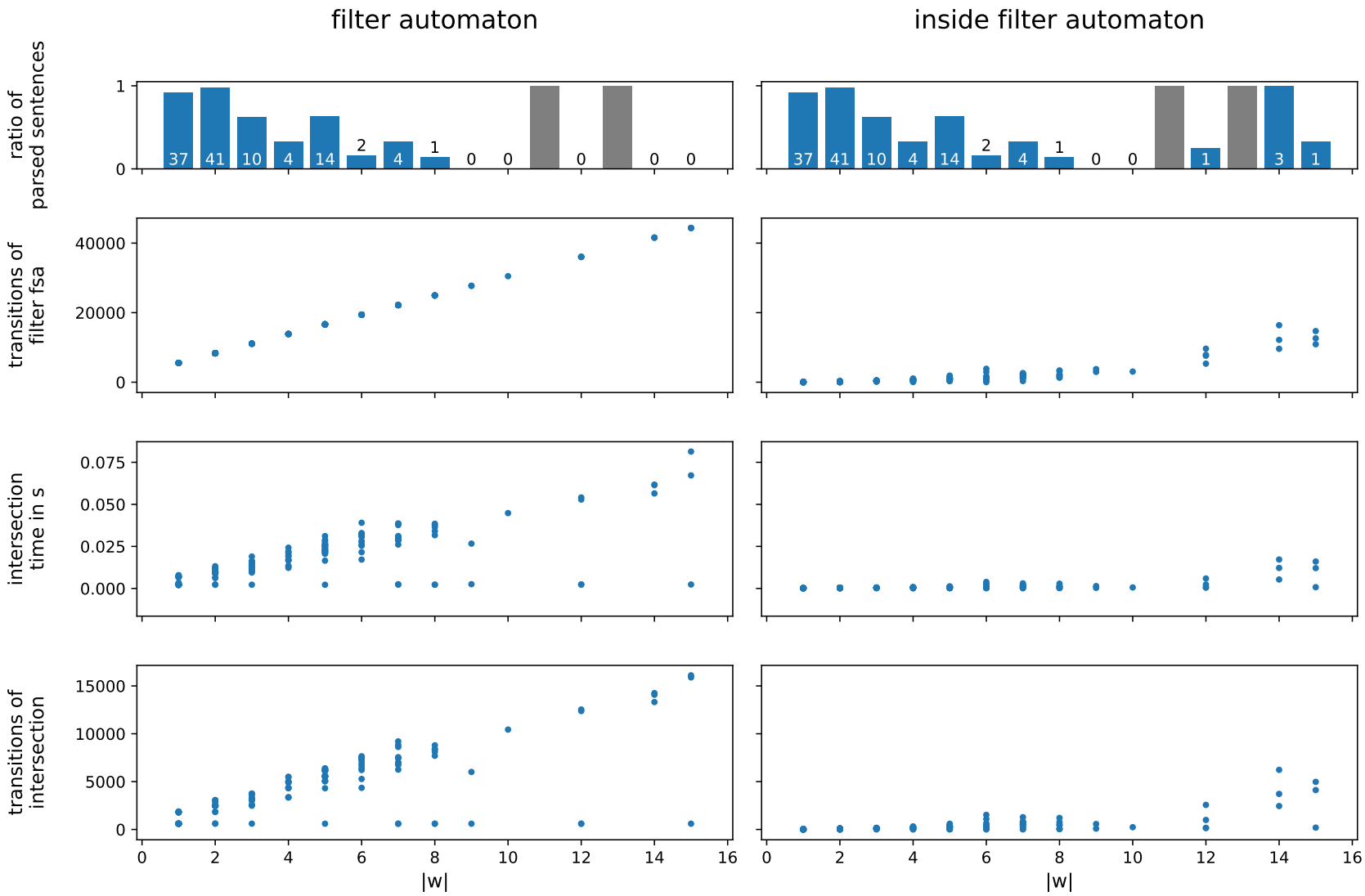


Figure 6.4: Comparison of the *filter automaton* to the *inside filter automaton*. The plots show the ratio of sentences for which a parse tree was found, the number of transitions of the filter automaton, the time needed to intersect the filter automaton with the generator automaton, and the transitions of the intersection automaton. Gray bars in the first plot indicate the unavailability of sentences of the specific length in the test corpus.

6.3.4 Grammar sizes

The plots in Figures 6.5 and 6.6 show the results of our parser with different splits of the NeGra corpus. We choose six fixed sizes for the training set, ranging from 250 to 1 500 parse trees, and extracted the parse trees of the training and test sets as explained in Section 6.2. As mentioned earlier, we did this tests because we had issues parsing sentences of the NeGra corpus with grammars extracted from more than 500 parse trees.

Figure 6.5 contains plots for the first half of our tests, i.e. for grammars extracted from 250, 500 and 750 parse trees in the NeGra corpus. For each of those training sets, we show the ratio of sentences we were able to parse, and the distribution of parse times for each sentence length in the test set. Contrary to our expectations, the first set of plots shows that the ratio of sentences from the test set that we are able to parse within 15 seconds does not seem to decrease using larger training sets. If anything, we suppose that these plots show better results of the parser with larger training corpora. In the second set of plots that show the parse time for each parsed sentence length. We can see very similar patterns in each plot. In particular, the growth of parse time with the sentence length seems to behave very similar with all three corpus splits. If we compare the specific values, we see only little increase with larger training corpora.

The results in the plots in Figure 6.6 seem to continue the trends, but not as clear as in the previous plots. We suppose that the parser is able to generalize better over the structure of parse trees with larger training corpora while preserving comparable parse times.

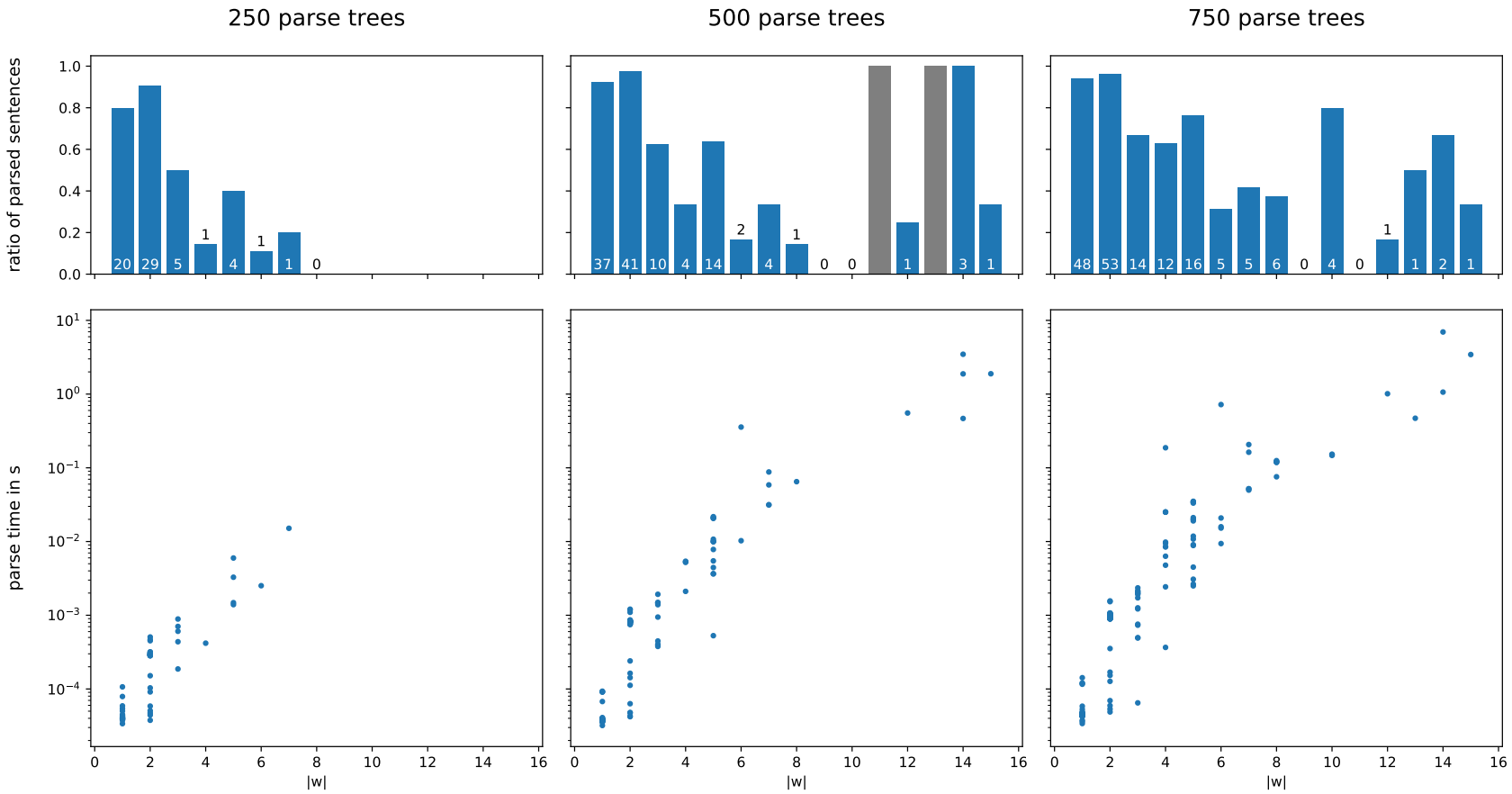


Figure 6.5: First part of the results with different training set and test set splits. We use training sets assembled from 250, 500, and 750 parse trees in NeGra and report the ratio of sentences that we are able to parse within 15 seconds and the parse time for each sentence length in the test set. Gray bars in the upper plots indicate the absence of sentences of the specific length in the test set.

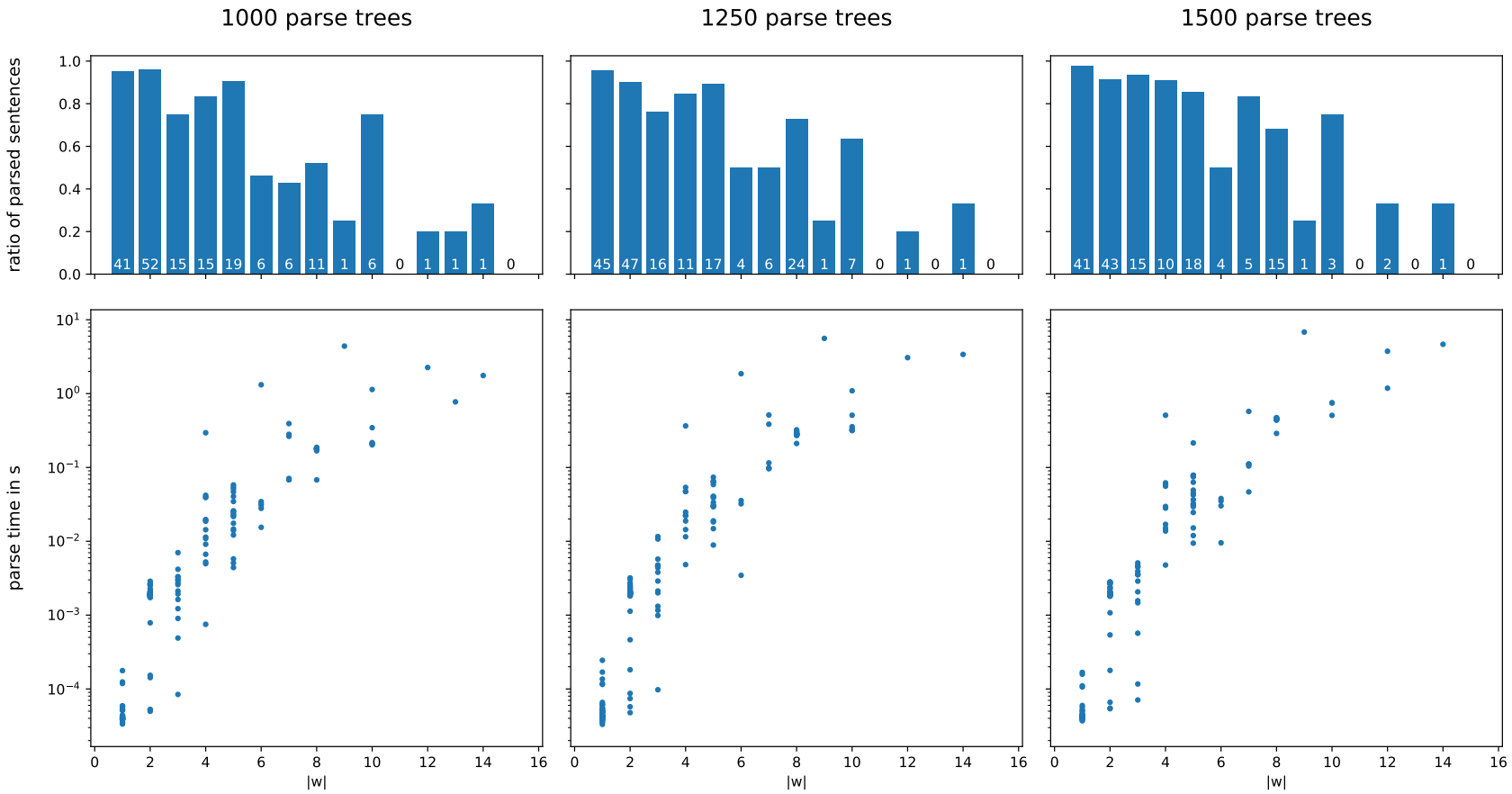


Figure 6.6: Second part of the results using different training set and test set splits of NeGra. We use training sets assembled from 100, 1 250, and 1 500 parse trees in NeGra and report the ratio of sentences that we are able to parse within 15 seconds and the parse time for each sentence length in the test set. Gray bars in the upper plots indicate the absence of sentences of the specific length in the test set.

6.3.5 Comparison with other parsers

We used our parser (*Rustomata*), *rparse* and *Grammatical Framework* to parse the sentences in the test set with the grammar extracted from the largest training set of our splits. So, there are 1 500 parse trees in the training set and 195 sentences in the test set with their corresponding gold parse trees.

Before we continue with the results of the experiments, we point out some limitations of rparse and Grammatical Framework that we noticed during the preparations of these experiments.

1. rparse can only parse sentences if we supply the POS-tags for each word. So, for the best results, we annotate the words in the sentences of the test set parsed by rparse with the POS-tags of the gold parse trees.

2. We suppose that Grammatical Framework is not able to parse punctuation characters, like dots and commas, using the grammar extracted with rparse. We did not find any way but to filter the test for the characters that Grammatical Framework could not handle. As a result of this, we use shorter sentences (and even less sentences) for the test of Grammatical Framework than for both other parsers.

3. Additionally to the previous point, we were only able to partially parse some sentences of the remaining test set using Grammatical Framework. Some of the parse trees it produced did not contain all of the words in the sentence that we tried to parse. We omitted these trees as the implementation of the evaluation metrics in Disco-Dop could not handle those cases. Moreover, we will not consider these trees as a parse of the sentences, i.e. they will not count towards the sentences that we were able to parse using Grammatical Framework in the third plot of Figure 6.7.

The ratio of sentences we were able to parse and the distribution of parse times for each parser and sentence length is shown in Figure 6.7. We can easily see that rparse outperforms our parser easily in terms of the amount of sentences in the test set that it was able to parse. On the other side, we were able to parse more sentences using our parser than with Grammatical Framework.

In the last plot, we can see that our parser is able to parse very small sentences faster than both other parsers. Although, the parse time of our parse grows much faster with respect to the sentence length than both other parsers. Both outperform our parser easily for sentences that are larger than four tokens with respect to the parse time.

Table 6.3 shows the qualitative results of our tests. rparse shows a very high accuracy for its predicted of POS-tags. Since we provided those as described earlier, we expect these predictions to be perfect. It is not exactly one, however since it could not parse all sentences. For both other parsers, we suppose to relate the accuracy of POS-tags to the ratio of parsed sentences as well.

All three parsers show similar f-scores in the interval between 0.75 and 0.86. Our parser shows similarities to Grammatical Framework, i.e. both parsers are very precise, even more precise than rparse, but seem to recall less constituents than rparse.

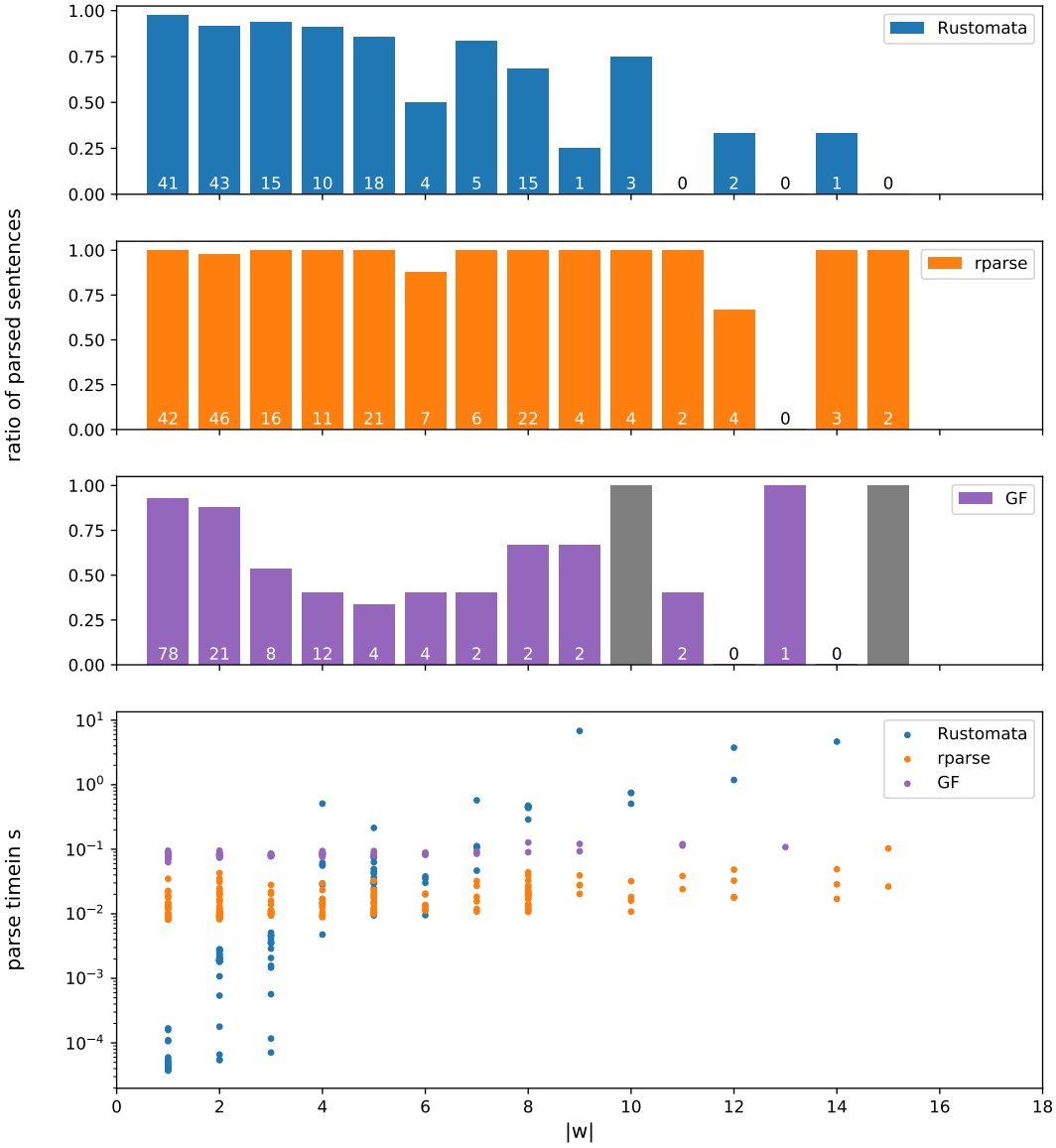


Figure 6.7: Comparison of our parser implemented in *Rustomata* to *rparse* and *Grammatical Framework*. The first three plots show the ratio of sentences we were able to parse. The last plot shows the distribution of parse times for each parser and each sentence length. Gray bars in the upper plots indicate the absence of sentences of the specific length in the test set.

Table 6.3: Comparison of our parser implemented in *Rustomata* to *rparse* and *Grammatical Framework*. We show the labeled recall, precision and f-scores for constituents in the parses of the sentences in the test set with respect to their gold parse trees. Furthermore, we report the quality of the predicted POS-tags using the accuracy with respect to the POS-tags in the gold parse tree. There are 177 sentences and parse trees in the test set with 418 constituents in total.

parser	Rustomata	rparse	Grammatical Framework
accuracy of POS-tags	0.6487	0.9473	0.4466
number of predicted constituents	306	416	239
labeled precision for all constituents	0.9608	0.8654	0.9414
labeled recall for all constituents	0.7033	0.8612	0.6285
labeled f-score for all constituents	0.8122	0.8633	0.7538

Chapter 7

Conclusion

In this thesis, we introduced an implementation of a k-best Chomsky-Schützenberger parser for weighted multiple context-free grammars. We started by formalizing the foundations of this approach and described multiple context-free grammars, different instances of automata with data storage and multiple Dyck languages and went on with an overview over the k-best Chomsky-Schützenberger parsing algorithm that was introduced by Denkinger [Den17b].

We introduced four different kinds of optimizations for the parsing approach that helped us to improve the performance of our implementation. After that, we described our implementation of the parser in great detail. With it, we extended Rustomata with the implementations of most of our described formalisms.

Our implementation was originally planned to use OpenFst, a library that provides fast and efficient functions for finite state automata. Unfortunately, we had to abandon this idea because a central part of our application now uses push-down automata instead of finite state automata. Since Rustomata, and the part we contributed to it, is very modular, we can replace our own implementation of push-down automata and finite state automata in favor of promising alternative libraries.

We evaluated our parser in terms of its parsing time, and its recall and precision of constituents using the NeGra corpus. Firstly, we performed experiments using different setups of our parser to evaluate the improvements due to our optimizations. Secondly, we used two other parsers under similar conditions to compare them to our implementation in Rustomata with respect to their parsing time and accuracy. The results show that our parser performs not as well as state-of-the-art parsers, but we suppose it is very accurate. We are very confident to find further optimizations to improve it.

List of Algorithms

List of Figures

List of Tables

Bibliography

[ARSSM07] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. "OpenFst: A General and Efficient Weighted Finite-State Transducer LIbrary". In: *Proceedings of the Twelfth International Conference on Implementation and Application of Automata*. 2007.

[Bla+91] Ezra Black et al. "A procedure for quantitatively comparing the syntactic coverage of English grammars". In: *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*. 1991.

[Col97] Michael Collins. "Three generative, lexicalised models for statistical parsing". In: *Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 1997, pp. 16–23.

[CS63] Noam Chomsky and Marcel P Schützenberger. "The algebraic theory of context-free languages". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 35. Elsevier, 1963, pp. 118–161.

[CSB16] Andreas van Cranenburgh, Remko Scha, and Rens Bod. "Data-Oriented Parsing with discontinuous constituents and function tags". In: *Journal of Language Modelling* 4.1 (2016), pp. 57–111. URL: http://dx.doi.org/10.15398/jlm.v4i1.100.

[DBD18] Toni Dietze, Matthias Buechse, and Tobias Denkinger. *Vanda-Haskell*. 2018. URL: https://github.com/tud-fop/vanda-haskell.

[Den16a] Tobias Denkinger. "A Chomsky-Schützenberger representation for weighted multiple context-free languages". In: *arXiv preprint arXiv:1606.03982* (2016).

[Den16b] Tobias Denkinger. "An automata characterisation for multiple context-free languages". In: *CoRR* abs/1606.02975 (2016). arXiv: 1606.02975. URL: http://arxiv.org/abs/1606.02975.

[Den17a] Tobias Denkinger. "Approximation of weighted automata with storage". In: *Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20-22 September 2017*. 2017, pp. 91–105. DOI: 10.4204/EPTCS.256.7. URL: https://doi.org/10.4204/EPTCS.256.7.

[Den17b] Tobias Denkinger. "Chomsky-Schützenberger parsing for weighted multiple context-free languages". In: *Journal of Language Modelling* (2017). DOI: 10.4204/EPTCS.256.7.

[Den17c] Tobias Denkinger. *Rustomata*. 2017. URL: https://github.com/tud-fop/rustomata.

[Hul09] Mans Hulden. "Parsing CFGs and PCFGs with a Chomsky-Schützenberger representation". In: *Language and Technology Conference*. Springer. 2009, pp. 151–160.

[KM13] Laura Kallmeyer and Wolfgang Maier. “Data-driven parsing using probabilistic linear context-free rewriting systems”. In: *Computational Linguistics* 39.1 (2013), pp. 87–119.

[Mie15] Sebastian Mielke. “Extracting and Binarizing probabilistic linear context-free rewriting systems”. MA thesis. TU Dresden, 2015.

[Ran11] Aarne Ranta. *Grammatical framework: Programming with multilingual grammars*. CSLI Publications, Center for the Study of Language and Information, 2011.

[SMFK91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. “On multiple context-free grammars”. In: *Theoretical Computer Science* 88.2 (1991), pp. 191–229.

[SUBK98] Wojciech Skut, Hans Uszkoreit, Thorsten Brants, and Brigitte Krenn. “A Linguistically Interpreted Corpus of German Newspaper Text”. In: *Proceedings of the 10th European Summer School in Logic, Language and Information (ESSLLI’98). Workshop on Recent Advances in Corpus Annotation, August 17-28*. Saarbrücken, Germany, 1998.

[YKS10] Ryo Yoshinaka, Yuichi Kaji, and Hiroyuki Seki. “Chomsky-Schützenberger-type characterization of multiple context-free languages”. In: *Language and Automata Theory and Applications* (2010), pp. 596–607.