
ADVAINECD
OFRNAL LAAGGE
TNOHY THOHY

Advanced Formal Language Theory

Ryan Cotterell, Anej Svete, Alexandra Butoi, Andreas Opedal, Franz Nowak

Wednesday 17th May, 2023

Contents

1 Context-free Languages **5**

1.1 Context-Free Grammars . . . 5
1.1.1 Context-free Grammars . . . 5
1.1.2 Derivation Sets and Ambiguity . . . 8
1.1.3 Two Views of CFGs . . . 9
1.1.4 Closure Properties of CFGs . . . 10
1.1.5 Semiring-weighted CFGs . . . 14
1.1.6 Treesums in WCFGs . . . 15
1.1.7 A Dynamic Program for Acyclic WCFGs . . . 17
1.1.8 Fixed-Point Iteration . . . 20
1.2 Newton's Method . . . 23
23section*.17
1.3 Grammar Transforms . . . 30
1.3.1 Introduction to Grammar Transforms . . . 30
1.3.2 Chomsky Normal Form . . . 32
1.3.3 Weighted Conversion to Chomsky Normal Form . . . 33
1.3.4 Left-Corner Transform . . . 45
1.3.5 Speculation . . . 51
1.4 The Parsing Problem . . . 53
1.4.1 Introducing Parsing . . . 53
1.4.2 The CKY Algorithm . . . 55
1.4.3 Bar-Hillel Parsing . . . 56
1.4.4 Agenda-based Parsing . . . 57
1.4.5 Earley's Algorithm . . . 59
1.5 Pushdown Automata . . . 62
1.5.1 Pushdown Automata . . . 62
1.5.2 Weighted Pushdown Automata . . . 64
1.5.3 Subclasses of WPDAs . . . 65
1.5.4 PDA-CFG Conversions . . . 66
1.5.5 Determinism . . . 69
1.5.6 Stack Language & PPDAs . . . 71
1.5.7 Shift-reduce parsing . . . 72
1.5.8 Computing Allsums . . . 75
1.6 PDA Transforms . . . 77
1.6.1 Binarization . . . 79
1.6.2 Nullary Removal . . . 79
1.6.3 Unary Removal . . . 82

1.7 WPDA Parsing 85
1.7.1 Lang's Algorithm 85
1.7.2 Butoi et al. (2022) 85

Chapter 1

Context-free Languages

1.1 Context-Free Grammars

Now we are going a rung higher on the ladder of the Chomsky hierarchy. We are going to consider **context-free languages**, which is a more general class of languages than the regular languages that are described by finite-state automata. In general, as FSAs cannot recognize context-free languages, we must instead resort to more general automata, known as **pushdown automata**. Before giving a formal treatment of pushdown automata, however, we will discuss an arguably more natural formalism for generating the context-free languages—**context-free grammars**.[1]

1.1.1 Context-free Grammars

We start by giving a formal definition of context-free grammars and introducing some notation.

Definition 1.1.1 (Context-free Grammar)**.** *A **context-free grammar** (CFG) is a 4-tuple* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ *where* $\mathcal{N}$ *is a non-empty set of non-terminal symbols,* Σ *is a set of terminal symbols (called alphabet) with* $\mathcal{N} \cap \Sigma = \emptyset$*,* $\mathrm{S} \in \mathcal{N}$ *is a designated start non-terminal symbol*[2] *and* $\mathcal{P}$ *is a set of production rules, where each rule* $p \in \mathcal{P}$ *is a 2-tuple* $(\mathrm{A}, \boldsymbol{\alpha})$*, with* $\mathrm{A} \in \mathcal{N}$ *and* $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$*, that we write as* $\mathrm{A} \to \boldsymbol{\alpha}$*.*

CFGs allow us to generate strings $\boldsymbol{y} \in \Sigma^*$ by applying production rules on its non-terminals. We apply a production rule $\mathrm{A} \to \boldsymbol{\alpha}$ to $\mathrm{A} \in \mathcal{N}$ in a rule p by taking A on the right-hand side of p and replacing it with $\boldsymbol{\alpha}$.[3] Formally:

Definition 1.1.2 (Rule Application)**.** *A production rule* $\mathrm{B} \to \boldsymbol{\beta}, \boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^*$*, is **applicable** to* B *in a rule* p*, if* p *takes the form*

$$\mathrm{A} \to \boldsymbol{\alpha}\,\mathrm{B}\,\boldsymbol{\gamma}, \quad \boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*$$

*The **result of applying*** $\mathrm{B} \to \boldsymbol{\beta}$ *to* $\boldsymbol{\alpha}\,\mathrm{B}\,\boldsymbol{\gamma}$ *is* $\boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma}$*.*

[1] You might wonder what non context-free grammars are. A more general class is that of context-sensitive grammars, in which a production rule may be surrounded by a left and right context. They are still however a set of restricted cases of general grammars, which are grammars that can emulate Turing machines. We will consider mildly context-sensitive grammars in later chapters.

[2] As is the case for initial states in FSAs, multiple start symbols could be possible. However we consider only one for the sake of simplicity.

[3] We say that X is on the right-hand side of a rule p if p takes the form $p = (\mathrm{A} \to \boldsymbol{\alpha}\,\mathrm{X}\boldsymbol{\gamma})$, where $\boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*$. We refer to A as the *left-hand side* or *head* of the production rule $\mathrm{A} \to \boldsymbol{\alpha}$, and $\boldsymbol{\alpha}$ as the *right-hand side* or *body* of the production rule.

In order to generate a string we start by applying some rule $\mathrm{S} \to \boldsymbol{\alpha} \in \mathcal{P}$ to S, then apply another production rule to a new non-terminal in $\boldsymbol{\alpha}$,[4] and follow this procedure until all non-terminal symbols have been transformed into terminal symbols. This procedure is called a **derivation**. The resulting string, its **yield**, is the string obtained by concatenating all terminal symbols read from left to right. Note that there are several orderings of which we can apply production rules if there are more than one non-terminals on the right-hand side of any production rule. We usually consider one of two such orderings: applying a production rule to the left-most non-terminal in each step, or applying a production rule to the right-most non-terminal in each step. Derivation is formalized below.

Definition 1.1.3 (Derivation)**.** *A **derivation** in a grammar $\mathfrak{G}$ is a sequence $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m$, where $\boldsymbol{\alpha}_1 \in \mathcal{N}$,$\boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_{m-1} \in (\mathcal{N} \cup \Sigma)^*$ and $\boldsymbol{\alpha}_m \in \Sigma^*$, in which each $\boldsymbol{\alpha}_{i+1}$ is formed by applying a production rule in $\mathcal{P}$ to $\boldsymbol{\alpha}_i$. We say that a derivation is a **left-most derivation** if for each $\boldsymbol{\alpha}_i$, a production rule is applied to its left-most non-terminal to form $\boldsymbol{\alpha}_{i+1}$. We say that a derivation is a **right-most derivation** if for each $\boldsymbol{\alpha}_i$, a production rule is applied to its right-most non-terminal to form $\boldsymbol{\alpha}_{i+1}$.*

We define the following relation: $\mathrm{A} \Rightarrow \boldsymbol{\beta}$ iff $\exists p \in \mathcal{P}$ such that $p = (\mathrm{A} \to \boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma})$, $\boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*, \boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^* \setminus \{\varepsilon\}$. The special case $\mathrm{A} \Rightarrow \varepsilon$ holds iff $\mathrm{A} \to \varepsilon$. Further, the reflexive transitive closure of the $\Rightarrow$ relation is denoted as $\overset{*}{\Rightarrow}$. We say that $\boldsymbol{\beta}$ is derived from A if $\mathrm{A} \overset{*}{\Rightarrow} \boldsymbol{\beta}$. In other words, A derives $\boldsymbol{\beta}$ if we can apply a finite sequence of production rules to generate $\boldsymbol{\beta}$ starting from A.

Following the general definition introduced in **??**, the (context-free) language of a CFG $\mathfrak{G}$ is defined as the set of all strings $\boldsymbol{y} \in \Sigma^*$ that can be derived from the start symbol S of $\mathfrak{G}$, or alternatively, the set of all yields possible from derivations in $\mathfrak{G}$ that start with S. We denote the language generated by $\mathfrak{G}$ as $L(\mathfrak{G})$.

Definition 1.1.4 (Language of a Grammar)**.** *The **language** of a context-free grammar $\mathfrak{G}$ is*

$$L(\mathfrak{G}) = \{\boldsymbol{y} \in \Sigma^* \mid \mathrm{S} \overset{*}{\Rightarrow} \boldsymbol{y}\} \tag{1.1}$$

Note that the same set of production rules applied in a different order may generate different strings. Hence, we cannot represent derivations simply as multisets over production rules. Instead, we represent derivations with **derivation trees** (also known as parse trees). In a derivation tree $\boldsymbol{t}$, the start symbol S is the root node, each internal node is labeled by a non-terminal symbol and each leaf node is labeled by either a terminal symbol or ε. The internal nodes in the tree correspond to production rules, for which the node is the left-hand side non-terminal symbol and the children are the symbols on the right-hand side in the same order. We denote the yield of a tree $\boldsymbol{t}$ by $\boldsymbol{y}(\boldsymbol{t})$. A tree rooted at any non-terminal other than S, but with the same characteristics as a derivation tree otherwise, is called a **derivation subtree**.

Example 1.1.1 (Nominal Phrases)**.** *CFGs are often used to model natural languages. Terminals would then correspond to words in the natural language, strings would be text sequences and non-terminals would be abstractions over words. As an example, consider a grammar $\mathfrak{G}$ that can generate a couple of nominal phrases. We let $\mathcal{N} = \{\mathrm{Adj}, \mathrm{Det}, \mathrm{N}, \mathrm{Nominal}, \mathrm{NP}\}$, $\Sigma = \{$a, big, female, giraffe, male, tall, the$\}$, $\mathrm{S} = \mathrm{Nominal}$ and define the following production rules:*

[4] We write $\mathrm{X} \in \boldsymbol{\alpha}$, which formally means a substring of $\boldsymbol{\alpha}$ with length 1. Unless otherwise stated, X can be either a non-terminal or a terminal.

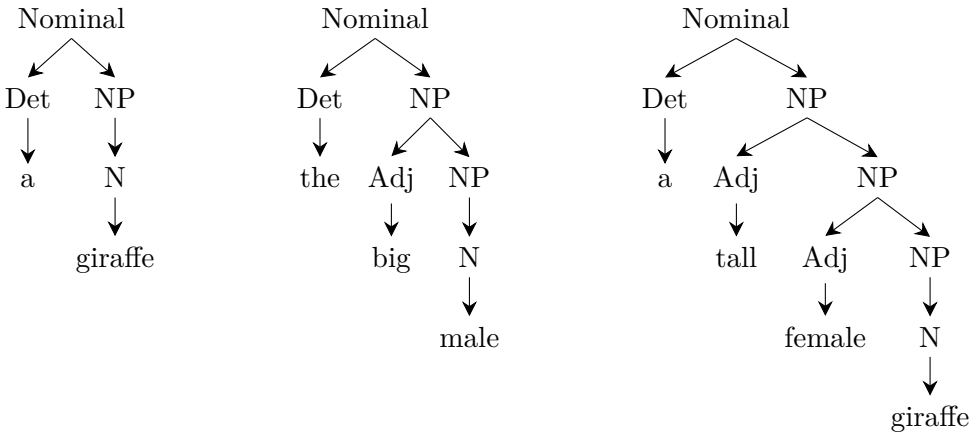


Figure 1.1: Derivation trees for natural language nominal phrases.

$$
\begin{aligned}
&\text{Nominal} \to \text{Det NP}\\
&\text{NP} \to \text{N} \mid \text{Adj NP}\\
&\text{Det} \to \textit{a} \mid \textit{the}\\
&\text{N} \to \textit{female} \mid \textit{giraffe} \mid \textit{male}\\
&\text{Adj} \to \textit{big} \mid \textit{female} \mid \textit{male} \mid \textit{tall}
\end{aligned}
$$

See Fig. 1.1 for a few examples of derivation trees in this grammar. Consider the tree with yield $\boldsymbol{y} =$ *a giraffe. Its left-most derivation is* $(\text{Nominal}, \text{Det NP}, a\ \text{NP}, a\ \text{N}, a\ \textit{giraffe})$*. Its right-most derivation is* $(\text{Nominal}, \text{Det NP}, \text{Det N}, \text{Det}\ \textit{giraffe}, a\ \textit{giraffe})$*. Hence, we note that the mapping from derivations to derivation trees follows a many-to-one relationship – it is non-injective.*

Example 1.1.2 (Recognizing $a^n b^n$)**.** *The language* $L = \{a^n b^n \mid n \in \mathbb{N}\}$ *is not regular. However, we can show that it is context-free with a simple grammar* $\mathfrak{G}$ *with* $\mathcal{N} = \{\text{A}\}$*,* $\Sigma = \{a, b\}$*,* $\text{S} = \text{A}$*,* $\mathcal{P} = \{\text{A} \to a\,\text{A}\,b, \text{A} \to \varepsilon\}$*.*

Proof. We show that $L = L(\mathfrak{G})$ in two steps: (i) showing that $L \subseteq L(\mathfrak{G})$ and (ii) showing that $L(\mathfrak{G}) \subseteq L$. Define $\boldsymbol{y}_n = a^n b^n$.

(i) We first need to show that each $\boldsymbol{y} \in L$ can be generated by $\mathfrak{G}$, which we do by induction.

Base case $(n = 0)$ We have that $\boldsymbol{y}_0 = \varepsilon$, which is generated by $\boldsymbol{t} = (\text{A} \to \varepsilon)$.

Inductive step $(n > 1)$ We have that $\boldsymbol{y}_n$ is generated by $\boldsymbol{t} = \underbrace{(\text{A} \to a\text{A}b) \cdots (\text{A} \to a\text{A}b)}_{n \text{ times}}(\text{A} \to \varepsilon)$. It is then easy to see that $\boldsymbol{y}_{n+1}$ is generated by the derivation we get by replacing the last rule $(\text{A} \to \varepsilon)$ with $(\text{A} \to a\text{A}b)(\text{A} \to \varepsilon)$. See Fig. 1.2 for an illustration.

(ii) Next, we show that for each derivation $\boldsymbol{t}$ in $\mathfrak{G}$, we have that $\boldsymbol{y}(\boldsymbol{t}) \in L$.

Base case $(\boldsymbol{t} = (\text{A} \to \varepsilon))$ It is trivial to see that the derivation $\boldsymbol{t} = (\text{A} \to \varepsilon)$ yields $\boldsymbol{y}(\boldsymbol{t}) = \varepsilon$.

Inductive step Now observe that $\mathcal{P}$ only contains two production rules and one non-terminal. Starting with X, we can either apply $\text{A} \to a\text{A}b$ to get one new non-terminal A, or apply $\text{A} \to \varepsilon$ to terminate the process. Hence, if we fix the length of the sequence of production rules there is no

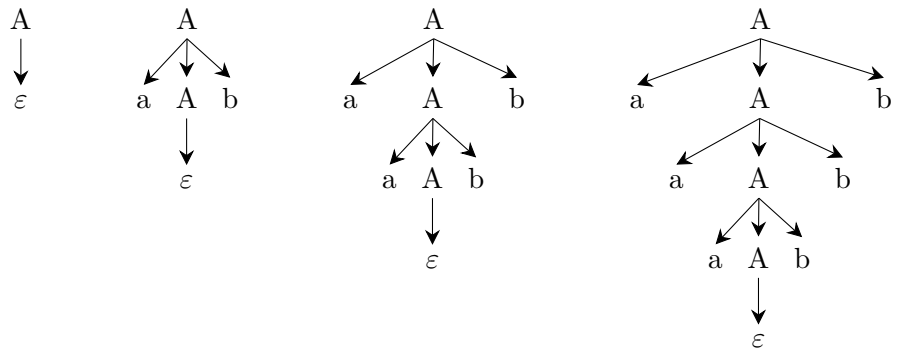


Figure 1.2: A sequence of derivation trees for the strings in $\{a^n b^n \mid n = 0, 1, 2, 3\}$.

ambiguity in which string that will be generated. Thus, by induction, we conclude that if we have a derivation tree given by $\underbrace{(\mathrm{A} \to a\mathrm{A}b), \ldots, (\mathrm{A} \to a\mathrm{A}b)}_{n \text{ times}}, (\mathrm{A} \to \varepsilon)$ generating $a^n b^n$, the derivation tree given by $\underbrace{(\mathrm{A} \to a\mathrm{A}b), \ldots, (\mathrm{A} \to a\mathrm{A}b)}_{n+1 \text{ times}}, (\mathrm{A} \to \varepsilon)$ will generate $a^{n+1} b^{n+1}$.

□

1.1.2 Derivation Sets and Ambiguity

We may have multiple derivation trees for a given string. Imagine, for instance, that we add a non-terminal B and rules $\mathrm{A} \to \mathrm{B}, \mathrm{B} \to \varepsilon$ to the grammar in Example 1.1.2. The empty string ε may then be derived either by $(\mathrm{A} \to \varepsilon)$, or $(\mathrm{A} \to \mathrm{B}), (\mathrm{B} \to \varepsilon)$, corresponding to two separate derivation trees. The set of these two trees comprise what we call the **derivation set** of ε. We denote a derivation set of a string $\boldsymbol{y}$, generated by the grammar $\mathfrak{G}$, as $\mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$.

We say that a grammar is **unambiguous** if for every string that can be generated by the grammar, there is only one associated derivation tree. We define it formally using our newly introduced notation.

Definition 1.1.5 (Unambiguity)**.** *A grammar $\mathfrak{G}$ is **unambiguous** if $\forall \boldsymbol{y} \in L(\mathfrak{G})$, $|\mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})| = 1$.*

The converse holds for **ambiguous** grammars.

Definition 1.1.6 (Ambiguity)**.** *A grammar $\mathfrak{G}$ is **ambiguous** if $\exists \boldsymbol{y} \in L(\mathfrak{G})$ such that $|\mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})| > 1$.*

We may also define ambiguity in terms of derivations. A grammar $\mathfrak{G}$ is ambiguous if there exists a string in $L(\mathfrak{G})$ that has more than one left-most (or right-most) derivation. Conversely, a grammar is unambiguous if every string in its language has a unique left-most and right-most derivation.

As in the FSA case, a subset of unambiguous context-free grammars are **deterministic context-free grammars**. They are context-free grammars that can be derived from deterministic pushdown automata, which we will see later in this chapter.

Example 1.1.3 (Ambiguous and Unambiguous Grammars)**.** *Consider the context-free (and regular) language $L = \Sigma^*$ with $\Sigma = \{x\}$. We can generate L with ambiguous CFGs, e.g. with a grammar $\mathfrak{G}$ having*

$$\mathcal{N} = \{\mathrm{A}\}, \mathrm{S} = \mathrm{A} \text{ and } \mathcal{P} = \{\mathrm{A} \to x\mathrm{A}, \mathrm{A} \to \mathrm{A}x, \mathrm{A} \to \varepsilon\}.$$

Note that this grammar is ambiguous since the trees $\boldsymbol{t}_1 = (\mathrm{A} \to x\mathrm{A}), (\mathrm{A} \to \varepsilon)$ *and* $\boldsymbol{t}_2 = (\mathrm{A} \to \mathrm{A}x), (\mathrm{A} \to \varepsilon)$ *both yield the string* x*. We can, however, generate the same language with a grammar that is unambiguous, by removing one of the non-terminal production rules:*

$$\mathcal{N} = \{\mathrm{A}\}, \mathrm{S} = \mathrm{A} \text{ and } \mathcal{P} = \{\mathrm{A} \to x\mathrm{A}, \mathrm{A} \to \varepsilon\}.$$

As seen in the above example, a given language could be generated by both ambiguous and unambiguous grammars. This leads us to the notion of ambiguity over *languages*.

Definition 1.1.7 (Inherent Ambiguity)**.** *A context-free language* L *is* ***inherently ambiguous*** *if every CFG that generates* L *is ambiguous.*

Given a context-free language, the problem of determining whether it is inherently ambiguous is actually undecidable, i.e. there exists no algorithm to solve the problem.[5] This follows as a direct consequence of Greibach's theorem (Greibach, 1963).

Theorem 1.1.1 (Greibach's theorem)**.** *Let* $\mathcal{C}$ *be a class of languages that is closed under concatenation with regular languages and union. It is further undecidable for any* $L \in \mathcal{C}$ *whether* $L = \Sigma^*$ *for any sufficiently large fixed* Σ*. Let* P *be a non-trivial property that is true for all regular languages and preserves quotient with a single terminal (meaning, if* L *has property* P*, then so does* $L_{/a} = \{w \mid wa \in L\}$*). Then* P *is undecidable for* $\mathcal{C}$*.*

This definition follows the one laid out in Hopcroft and Ullman (1979). We refer to Hopcroft and Ullman (1979) for a proof of Thm. 1.1.1 and its application for the undecidability of inherently ambiguous context-free languages.

Example 1.1.4. *We can, however, show for some specific context-free languages that they are inherently ambiguous. Consider for instance the language defined by*

$$L = \{a^n b^n c^m d^m \mid n > 0, m > 0\} \cup \{a^n b^m c^m d^d \mid n > 0, m > 0\} \tag{1.2}$$

The language L *is context-free and inherently ambiguous, as shown in Hopcroft et al. (2006).*

Further, we introduce the notion of the set of derivations in a grammar and for a given non-terminal. The **derivation set of a grammar**, $\mathcal{D}_{\mathfrak{G}}$, is the set of all derivations possible under the grammar. More formally, it can be defined as the union over the derivation set for the strings in its language,

$$\mathcal{D}_{\mathfrak{G}} \stackrel{\text{def}}{=} \bigcup_{\boldsymbol{y}' \in L(\mathfrak{G})} \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y}') \tag{1.3}$$

The **derivation set of a non-terminal** $\mathrm{A} \in \mathcal{N}$ in $\mathfrak{G}$, denoted $\mathcal{D}_{\mathfrak{G}}(\mathrm{A})$, is defined as the set of derivation subtrees with root node A. Note that $\mathcal{D}_{\mathfrak{G}}$ could be defined as $\mathcal{D}_{\mathfrak{G}}(\mathrm{S})$. For a terminal symbol $a \in \Sigma$, we trivially define the derivation set $\mathcal{D}_{\mathfrak{G}}(a)$ to be empty.[6]

1.1.3 Two Views of CFGs

As should be apparent from the discussion above, there are two ways to view a CFG. One way is to consider what strings $\boldsymbol{y} \in \Sigma^*$ can be generated by the CFG. As previously mentioned, the set of all such strings comprise what is known as a context-free language. This view leads us to the notion of pushdown automata. In line with this view, we consider two CFGs to be equal if they generate the same language. This is *weak* equivalence, defined specifically for CFGs below.

[5] Another undecidable problem is to determine whether two CFGs generate the same language.

[6] Empty derivation sets for terminal symbols is defined solely for ease of notation later.

Definition 1.1.8 (Weak Equivalence)**.** *Two CFGs $\mathfrak{G}$ and $\mathfrak{G}'$ are **weakly equivalent** if they generate the same set of strings, i.e., if we have $L(\mathfrak{G}) = L(\mathfrak{G}')$.*

The second view considers a CFG as a device for generating a set of trees. This view is useful as well, as in the study of natural languages we are often interested in describing sets of trees, for instance, in order to represent the syntactic structure of sentences. This view leads us to the notion of a finite-state tree automata (Thatcher, 1967), which is discussed in a later chapter. Note that a string can be generated by multiple different trees, while a given tree only yields one unique string. This view suggests two CFGs to be equal if they generate the same set of derivation trees. This is the CFG operationalization of *strong* equivalence, for which we give a formal definition below.

Definition 1.1.9 (Strong Equivalence)**.** *Two CFGs $\mathfrak{G}$ and $\mathfrak{G}'$ are **strongly equivalent** if they generate the same set of derivation trees, i.e., if we have $\mathcal{D}_{\mathfrak{G}} = \mathcal{D}_{\mathfrak{G}'}$.*

Proposition 1.1.1. *Strong equivalence implies weak equivalence.*

Proof. If $\mathcal{D}_{\mathfrak{G}} = \mathcal{D}_{\mathfrak{G}'}$ it follows, since each derivation corresponds to a single unique string, that $\forall \boldsymbol{y} \in L(\mathfrak{G}), \exists \boldsymbol{y}' \in L(\mathfrak{G}')$ such that $\mathcal{D}_{\mathfrak{G}}(\boldsymbol{y}) = \mathcal{D}_{\mathfrak{G}'}(\boldsymbol{y}')$ and $\boldsymbol{y} = \boldsymbol{y}'$. By symmetry, the converse also holds. Hence, $L(\mathfrak{G}) = L(\mathfrak{G}')$. □

The two grammars presented in Example 1.1.3 are weakly equivalent, but they are not strongly equivalent.

1.1.4 Closure Properties of CFGs

Like regular languages, context-free languages are closed under union, concatenation and the Kleene Closure. This means that, given context-free grammars $\mathfrak{G}$ and $\mathfrak{G}'$, we have that $L(\mathfrak{G}) \cup L(\mathfrak{G}')$, $L(\mathfrak{G}) \circ L(\mathfrak{G}')$ and $L(\mathfrak{G})^*$ are all context-free. These properties can be shown constructively using the grammars, i.e., if we can construct a CFG that generates the language that results from the operation, then the resulting language is by definition context-free. We show closure under union below, and leave as exercises to show the other two properties.

Proposition 1.1.2 (Closure under Union)**.** *Context-free languages are closed under union.*

Proof. Let $\mathfrak{G}_1(L_1) = (\mathcal{N}_1, \Sigma_1, \mathrm{S}_1, \mathcal{P}_1)$ and $\mathfrak{G}_2(L_2) = (\mathcal{N}_2, \Sigma_2, \mathrm{S}_2, \mathcal{P}_2)$. We can then construct a CFG, $\mathfrak{G}_\cup = (\mathcal{N}_\cup, \Sigma_\cup, \mathrm{S}, \mathcal{P}_\cup)$, with:

$$\begin{aligned}
\mathcal{N}_\cup &= \mathcal{N}_1 \cup \mathcal{N}_2 \cup \{\mathrm{S}\} \\
\Sigma_\cup &= \Sigma_1 \cup \Sigma_2 \\
\mathcal{P}_\cup &= \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{\mathrm{S} \to \mathrm{S}_1, \mathrm{S} \to \mathrm{S}_2\},
\end{aligned}$$

where we note that $L(\mathfrak{G}_\cup) = L(\mathfrak{G}_1) \cup L(\mathfrak{G}_2)$. Hence, we conclude that context-free languages are indeed closed under the union operator. □

Context-free languages are also closed under **homomorphism** and **inverse homomorphism**. Homomorphism is a particular string substitution, where each terminal is replaced by some string in L, i.e. $b := f(a), \forall a \in \Sigma$ with $f : \Sigma \to L$. The homomorphic image of f under Σ, is the set of all such substitutions

$$f(\Sigma) = \{f(a) \mid a \in \Sigma\} \tag{1.4}$$

The inverse homomorphic image of L is defined as

$$f^{-1}(L) = \{a \mid f(a) \in L\} \tag{1.5}$$

Non-closure of CFGs

Context-free languages are not, however, closed under intersection and complement, in contrast to regular languages. We show by contradiction that CFGs are not closed under intersection, and for that we require the pumping lemma (Kreowski, 1979).

Theorem 1.1.2 (Pumping Lemma for Context-free Languages)**.** *If a language L is context-free, there exists an integer $p \geq 1$ (called "pumping length") such that $\forall \boldsymbol{y} \in L$ with $|\boldsymbol{y}| \geq p$, $\boldsymbol{y}$ can be written as*

$$\boldsymbol{y} = uvwxy,$$

where u, v, w, x, y are substrings of $\boldsymbol{y}$ such that

- $|vx| \geq 1$
- $|vwx| \leq p$
- $uv^n wx^n y \in L, \quad \forall n \geq 0$

The pumping lemma states a property for all strings in the context-free language with a length above the pumping length. Informally, for these strings $\boldsymbol{y} = uvwxy$, we require that the subsequence vx is non-empty but that the concatenation with w, vwx, is at most the pumping length, and that repeating v and x the same number of times in $\boldsymbol{y}$ produces a new string that is still in the same language. The pumping lemma gets its name from this process of repeating, or "pumping up", v and x.

We do not prove the pumping lemma here, but the proof relies on the fact that we can write any context-free grammar in Chomsky normal form, which is a grammar transformation we will discuss in depth later in this chapter. The pumping lemma is a necessary, but not sufficient, condition for a language to be context-free. That makes it suitable for showing that some specific language is *not* context-free.

Proposition 1.1.3 (Non-closure under intersection)**.** *Context-free languages are not closed under intersection.*

Proof. Consider the two languages $L_1 = \{a^n b^n c^m \mid m > 0, n > 0\}$ and $L_2 = \{a^m b^n c^n \mid m > 0, n > 0\}$. We know that L_1 is context-free since it is generated by the following CFG:

$$\mathcal{N} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$$
$$\mathcal{P} = \{\mathrm{S} \to \mathrm{AB}, \mathrm{A} \to a\mathrm{A}b, \mathrm{A} \to ab, \mathrm{B} \to c\mathrm{B}, \mathrm{B} \to c\}.$$

Also L_2 is context-free, as it can be generated e.g. by:

$$\mathcal{N} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$$
$$\mathcal{P} = \{\mathrm{S} \to \mathrm{AAB}, \mathrm{A} \to a\mathrm{A}, \mathrm{A} \to a, \mathrm{B} \to b\mathrm{B}c, \mathrm{B} \to bc\}.$$

It is easy to see that if we intersect the two languages, we get

$$L_\cap = L_1 \cap L_2 = \{a^m b^m c^m \mid m > 0\}.$$

This language is however not context-free. This can be shown with the pumping lemma as follows. Assume that $L = \{a^m b^m c^m \mid m > 0\}$ is a context-free language. Let $\boldsymbol{y} = a^p b^p c^p$ for some large pumping length p, which is in L. The pumping lemma lets us write $\boldsymbol{y} = uvwxy$ with the conditions mentioned in Thm. 1.1.2. Since $|vwx| \leq p$ must hold, we observe that vwx can not contain more than two symbols (in $\Sigma = \{a, b, c\}$). This gives us five possibilities:

- $vwx = a^i, \quad 1 \leq i \leq p$
- $vwx = a^i b^j, \quad 2 \leq i + j \leq p$
- $vwx = b^i, \quad 1 \leq i \leq p$
- $vwx = b^i c^j, \quad 2 \leq i + j \leq p$
- $vwx = c^i, \quad 1 \leq i \leq p$

We see that for all the above possibilities, unless $n = 1$, $uv^n wx^n y$ does not follow the form $a^m b^m c^m$ and is hence not in L, which contradicts the pumping lemma. □

We also show the non-closure of context-free languages under complement, using the two closure properties we have previously shown.

Proposition 1.1.4 (Non-closure under complement)**.** *Context-free languages are not closed under complement.*

Proof. Assume that context-free languages are closed under complement and let L_1 and L_2 be two such languages. By assumption, the complement languages $\overline{L_1}$ and $\overline{L_2}$ are thus context-free. Since context-free languages are closed under union, the language $\overline{\overline{L_1} \cup \overline{L_2}}$ must be context-free. However, we have that

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2,$$

and we know that context-free languages are not closed under intersection, which contradicts our assumption. □

Closure under Intersection with FSAs

Although intersecting a context-free language with another context-free language will generally not yield a new context-free language, they are in fact closed under intersection with *regular languages*. So, if we take $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ and $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ we can construct a new context-free grammar $\mathfrak{G}_\cap = (\mathcal{N}_\cap, \Sigma, \mathrm{S}, \mathcal{P}_\cap)$ such that $L(\mathfrak{G}_\cap) = L(\mathfrak{G}) \cap L(\mathcal{A})$. Bar-Hillel et al. (1961) showed this property using the following construction.[7]

Transform 1.1.1 (Intersecting with FSA)**.** *Intersecting* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ *with* $\mathcal{A} = (Q, \Sigma, I, F, \delta)$*, yields* $\mathfrak{G}_\cap = (\mathcal{N}_\cap, \Sigma, \mathrm{S}, \mathcal{P}_\cap)$ *where:*[8]

- *For each* $X \to X_1 X_2 ... X_n$ *with* $n \in \mathbb{N}$ *in* $\mathcal{P}$*, add* $[q_0, X, q_n] \to [q_0, X_1, q_1][q_1, X_2, q_2] \dots [q_{n-1}, X_n, q_n]$ *to* $\mathcal{P}_\cap$ *for all* $q_0, \dots, q_n \in Q$*;*
- *For each* $[q_I, \mathrm{S}, q_F] \to [q_I, X_1, q_1][q_1, X_2, q_2] \dots [q_{n-1}, X_n, q_F]$ *in* $\mathcal{P}_\cap$ *with* $q_I \in I, q_F \in F$*, add* $\mathrm{S} \to [q_I, \mathrm{S}, q_F]$ *to* $\mathcal{P}_\cap$*;*
- *For each* $X \to \varepsilon$ *in* $\mathcal{P}$*, add* $[q, X, q] \to \varepsilon$ *to* $\mathcal{P}_\cap$ *for all* $q \in Q$*;*
- *For each* $(q_1, a, q_2) \in \delta$*, add* $[q_1, a, q_2] \to a$ *to* $\mathcal{P}_\cap$*;*

[7] In the below, the "[]" bracketed units denote non-terminals.

[8] This construction actually fails in the case when the FSA has ε-arcs. This was recently discussed and corrected by Pasti et al. (2022). However, we provide the original construction here for the sake of simplicity.

Note that all rules in $\mathfrak{G}_\cap$ are constructed either out of a rule in $\mathfrak{G}$ or a transition in $\mathcal{A}$. It can be seen that for every derivation yielding a string in $\mathfrak{G}_\cap$, we have a unique derivation in $\mathfrak{G}$ yielding the same string and a unique set of transitions in $\mathcal{A}$ recognizing the same string. Intuitively, this follows since all rules in $\mathcal{P}$, both terminal and non-terminal, will translate to non-terminal rules in $\mathcal{P}_\cap$. All non-terminal rules in $\mathcal{P}_\cap$ stem from $\mathcal{A}$, and hence a full derivation sequence with a yield requires that the yield string is accepted by $\mathcal{A}$.

We can also see that if a string can be derived by $\mathfrak{G}$ but is not recognized by $\mathcal{A}$, it also cannot be derived by $\mathfrak{G}_\cap$ due to missing terminal rules. The converse also holds.

This construction leads to an algorithm that runs in $\mathcal{O}(|Q|^{n_{\max}+1})$, where $n_{\max}$ is the length of the longest right-hand side in $\mathcal{P}$. Besides the intimidating runtime, another drawback is that we typically get a lot of "useless" non-terminals.[9] Analogously to the FSA case, a useless non-terminal is one that is not accessible and co-accessible. Although for CFGs we typically use a different terminology: "reachable" instead of "accessible" and "generating" instead of "co-accessible". We re-define these notions for the CFG case.

Definition 1.1.10 (Accessibility for CFGs)**.** *A symbol* $\mathrm{X} \in (\mathcal{N} \cup \Sigma)$ *is* ***reachable*** *(or accessible) if* $\mathrm{S} \overset{*}{\Rightarrow} \mathrm{X}$.

Definition 1.1.11 (Co-accessibility for CFGs)**.** *A non-terminal* A *is* ***generating*** *(or co-accessible) if* $\exists \boldsymbol{y} \in \Sigma^*$ *such that* $\mathrm{A} \overset{*}{\Rightarrow} \boldsymbol{y}$.

In words, reachable symbols are those that can be derived from the start symbol and generating non-terminals are those from which at least one string (including the empty string) can be derived. Note that we define reachable for both non-terminals and terminals, while generating is only defined for non-terminals. Note that terminals that are not reachable are not contained in the language generated by the grammar.

Definition 1.1.12 (Pruned CFG)**.** *A CFG is* ***pruned*** *(or trimmed) if it has no useless non-terminals, i.e. all non-terminals are both reachable and generating.*

Pruning is a grammar transformation that preserves weak equivalence, while eliminating all useless non-terminals. As for the FSA case, we can construct a simple algorithm for pruning a CFG, given that we have access to reachable and generating non-terminals, by iterating over all production rules and keeping only those where all participating non-terminals are not useless. We will return to finding reachable and generating non-terminals later in this chapter.

Relation to parsing Transform 1.1.1 does have a nice interpretation for parsing. Let $\mathcal{A}$ be a linear FSA that only accepts a given string $\boldsymbol{y} \in \Sigma^*$, i.e. one where

- $Q = \{q_0, q_1, ..., q_n\}$;
- $I = \{q_0\}$;
- $F = \{q_n\}$;
- $\delta = \{(q_0, \boldsymbol{y}_1, q_1), \ldots, (q_{n-1}, \boldsymbol{y}_n, q_n)\}$ with n being the length of $\boldsymbol{y}$.

[9] There do exist improvements of this algorithm. For instance, we can avoid useless rules by adding rules first through a bottom-up phase in which only rules where all right-hand side non-terminals have been shown to be generating are added, and a following top-down phase in which we check for reachable non-terminals. We refer to Nederhof and Satta (2008) for details.

The grammar $\mathfrak{G}_\cap$ resulting from an intersection of $\mathcal{A}$ with a CFG $\mathfrak{G}$, where $\boldsymbol{y} \in L(\mathfrak{G})$, can then be seen as a grammar that derives all parse trees of $\boldsymbol{y}$, a so-called **parse-forest grammar**. Moreover, with a weighted extension of the above construction algorithm we can use the parse-forest grammar to make inference; for instance finding the most probable parse tree or computing the normalizing constant for a probabilistic parse. This construction is hence closely related to parsing. We discuss parsing in depth in §1.4.

1.1.5 Semiring-weighted CFGs

As in the case of finite-state machines, we augment the classic, unweighted context-free grammars with a semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. We associate each rule $\mathrm{A} \to \boldsymbol{\alpha}$ with a weight $w = \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \in \mathbb{K}$.

Definition 1.1.13 (Weighted Context-free Grammar)**.** *A **weighted context-free grammar** over semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is a 5-tuple $(\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ where $\mathcal{N}$ is a non-empty set of non-terminal symbols, Σ is a set of terminal symbols (called alphabet) with $\mathcal{N} \cap \Sigma = \emptyset$, $\mathrm{S} \in \mathcal{N}$ is a designated start non-terminal symbol and $\mathcal{P}$ is the set of production rules, where each rule $p \in \mathcal{P}$ is a 3-tuple $(\mathrm{A}, \boldsymbol{\alpha}, w)$, with $\mathrm{A} \in \mathcal{N}$, $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$ and $w \in \mathbb{K}$, that we write as $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}$.*

For the sake of simplifying notation, we also overload $\mathcal{W}$ by writing the weight w associated with a rule $\mathrm{A} \to \boldsymbol{\alpha}$ as $w = \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha})$. As should be clear, the one extension compared to an ordinary CFG is that each production rule is associated with a weight in the semiring. Again analogously to the WFSA case, we say that a string $\boldsymbol{y}$ is in the language of a WCFG $\mathfrak{G}$ if there exists a derivation tree $\boldsymbol{t}$ in $\mathfrak{G}$ containing only non-zero weights with yield $\boldsymbol{y}(\boldsymbol{t}) = \boldsymbol{y}$. It follows that we can represent a CFG $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ with a WCFG with the Boolean semiring by setting $\mathcal{W}(p) = 1$ for all rules p in $\mathcal{P}$.

Additionally, we can transform any WCFG $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ to a Boolean WCFG $\mathfrak{G}' = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W}')$ by setting

- $\mathcal{W}'(p) = 1, \quad \forall p \in \mathcal{P}$ where $\mathcal{W}(p) \neq \mathbf{0}$
- $\mathcal{W}'(p) = 0, \quad \forall p \in \mathcal{P}$ where $\mathcal{W}(p) = \mathbf{0}$

We refer to this transformation as **booleanization**. A boolean WCFG is identical to an unweighted CFG when considering only those rules that have weight $\mathbf{1}$.

In the context of semiring-weighted WCFGs, we can define a weighting over derivation trees and strings. We start by giving a definition for the weight of a tree as the $\otimes$-multiplication over all rules belonging to the tree, i.e. as a *multiplicatively decomposable* function over the weights of its participating rules.

Definition 1.1.14. *The weight of a derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}$ defined by a WCFG $\mathfrak{G}$ is*[10]

$$\mathcal{W}(\boldsymbol{t}) = \bigotimes_{(\mathrm{A} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \tag{1.6}$$

As in the WFSA case, we can also define the **string sum** for a particular string under a WCFG. For a given string $\boldsymbol{y}$, the string sum is the $\oplus$-sum of all derivations in the WCFG with yield $\boldsymbol{y}$.

[10] For the case when the semiring is non-commutative we perform a pre-order traversal of the derivation tree to compute its weight.

Definition 1.1.15. *The* ***string sum*** $Z_{\mathfrak{G}}(\boldsymbol{y})$ *of a string* $\boldsymbol{y}$ *generated by a WCFG* $\mathfrak{G}$ *is defined by*

$$Z_{\mathfrak{G}}(\boldsymbol{y}) = \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})} \mathcal{W}(\boldsymbol{t}) \tag{1.7}$$

$$= \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})} \bigotimes_{(\mathrm{A} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \tag{1.8}$$

We will consider string sums as we discuss parsing later in this chapter. Before that, we start our discussion of WCFGs' counterpart to pathsums—**treesums**.[11]

1.1.6 Treesums in WCFGs

As in the pathsum for WFSA, a **treesum** is the $\oplus$-sum for the weights of all the trees in a WCFG. We first define the treesum for symbols (non-terminals and terminals).

Definition 1.1.16 (Treesum)**.** *The* ***treesum*** *of a non-terminal* A *in a grammar* $\mathfrak{G}$ *is defined by*

$$Z_{\mathfrak{G}}(\mathrm{A}) = \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A})} \mathcal{W}(\boldsymbol{t}) \tag{1.9}$$

$$= \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A})} \bigotimes_{(\mathrm{B} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{B} \to \boldsymbol{\alpha}) \tag{1.10}$$

The treesum for a terminal $a \in \Sigma \cup \{\varepsilon\}$ *is defined to be*

$$Z_{\mathfrak{G}}(a) \stackrel{\text{def}}{=} \boldsymbol{1}. \tag{1.11}$$

The treesum for a grammar is then simply the treesum of its start symbol.

Definition 1.1.17 (Treesum WCFG)**.** *The* ***treesum*** *of a WCFG* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ *is*

$$Z_{\mathfrak{G}} = Z_{\mathfrak{G}}(\mathrm{S}) \tag{1.12}$$

$$= \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{S})} \mathcal{W}(\boldsymbol{t}) \tag{1.13}$$

$$= \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{S})} \bigotimes_{(\mathrm{B} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{B} \to \boldsymbol{\alpha}) \tag{1.14}$$

When the grammar $\mathfrak{G}$ we refer to is clear from context, we drop the subscript and write e.g. $Z(\mathrm{S})$.

Although we can in some cases compute the treesum of a WCFG in closed-form, as we will see in the example below, we generally require some efficient algorithm to be able to do so.

Example 1.1.5 (Geometric Series as a Treesum)**.** *Consider the WCFG* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$*, given by* $\mathcal{N} = \{\mathrm{A}\}$*,* $\Sigma = \{x\}$*,* $\mathrm{S} = \mathrm{A}$*, a Rational semiring* $\mathcal{W}$ *and the rules:*

$$\mathrm{A} \xrightarrow{1/2} x\,\mathrm{A}$$

$$\mathrm{A} \xrightarrow{1} \varepsilon$$

[11] These quantities are also referred to as partition functions or free sums in the literature.

The language generated by $\mathfrak{G}$ *is* $L(\mathfrak{G}) = \{x^n \mid n \geq 0\}$. *Further note that this grammar is unambiguous – each string* $\boldsymbol{y} = x^m$, *for some* $m \geq 0$, *is associated with the derivation tree given by* $\underbrace{(\mathrm{A} \xrightarrow{1/3} x\,\mathrm{A}), \ldots, (\mathrm{A} \xrightarrow{1/3} x\,\mathrm{A})}_{m \text{ times}}, (\mathrm{A} \xrightarrow{1} \varepsilon)$. *Due to the multiplicative decomposition over the weights of the rules, the weight associated with each derivation tree* $\boldsymbol{t}$ *will hence be*

$$\mathcal{W}(\boldsymbol{t}) = \left(\frac{1}{3}\right)^m \times 1 = \left(\frac{1}{3}\right)^m$$

Accordingly, we can compute the treesum of $\mathfrak{G}$ *using the closed-form expression for geometric series:*

$$\begin{aligned} Z_{\mathfrak{G}} &= \sum_{m=0}^{\infty} \left(\frac{1}{3}\right)^m \\ &= \frac{1}{1 - 1/3} \\ &= \frac{3}{2} \end{aligned}$$

In the discussion on computing treesums that will follow, we will make use of the notion of **children**. Intuitively, X is a child of A if it is on the right-hand side of any production rule with left-hand side A. Formally:

Definition 1.1.18 (Child)**.** $\mathrm{X} \in \Sigma \cup \mathcal{N}$ *is a* ***child*** *of* $\mathrm{A} \in \mathcal{N}$ *iff* $\mathrm{A} \Rightarrow \mathrm{X}$. $\mathrm{X} \in \Sigma \cup \mathcal{N}$ *is a* ***descendant*** *of* $\mathrm{A} \in \mathcal{N}$ *iff* $\mathrm{A} \overset{*}{\Rightarrow} \mathrm{X}$.

We denote the set of children of A as $\mathsf{children}(\mathrm{A})$ and the set of descendants of A as $\mathsf{desc}(\mathrm{A})$. We also talk about the size of a grammar, which is useful e.g. when discussing algorithm time and space complexity.

Definition 1.1.19 (Size of Production Rule)**.** *The* ***size of a production rule*** $p \in \mathcal{P}$, *denoted* $|p|$, *is the number of symbols it includes, i.e., the length of its right-hand side plus one.*

Definition 1.1.20 (Size of Grammar)**.** *The* ***size of a grammar*** $\mathfrak{G}$, *denoted* $|\mathfrak{G}|$, *is the sum over the size of its production rules:*

$$|\mathfrak{G}| = \sum_{p \in \mathcal{P}} |p|$$

Expressing $Z_{\mathfrak{G}}$ as a Set of Non-Linear Equations

To derive an implementable algorithm for computing the treesum in a WCFG, it is easiest to recall how we derived an efficient algorithm for computing the pathsum in a WFSA. A lot of the work we put into deriving algorithms exploited a connection between computing the pathsum in a WFSA and solving a linear system. Indeed, it is exactly this reason that well-known algorithms for linear system solving, e.g., Gauss–Jordan, became of importance. We will exploit the very same connection for the WCFG case, as it will lead to a natural generalization of an all-purpose treesum algorithm in WCFGs. Only now, we will obtain a system of *non-linear* equations. In this brief section, we make this connection explicit for WCFGs.

Making use of the distributive property of semirings, we express the treesum of a non-terminal recursively as a function of its children:

$$
\begin{aligned}
Z(\mathrm{A}) &= \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A})} \bigotimes_{(\mathrm{B} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{B} \to \boldsymbol{\alpha}) && \text{(definition)} \\
&= \bigoplus_{\mathrm{A} \to \boldsymbol{\alpha}'} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}') \otimes \bigotimes_{\mathrm{C} \in \boldsymbol{\alpha}'} \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{C})} \bigotimes_{(\mathrm{C} \to \boldsymbol{\alpha}) \in \boldsymbol{t}} \mathcal{W}(\mathrm{C} \to \boldsymbol{\alpha}) && \text{(distributive property)} \\
&= \bigoplus_{\mathrm{A} \to \boldsymbol{\alpha}'} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}') \otimes \underbrace{\bigotimes_{\mathrm{C} \in \boldsymbol{\alpha}'} Z(\mathrm{C})}_{\stackrel{\text{def}}{=} Z(\boldsymbol{\alpha}')} && (\mathbf{1} \text{ being identity element of } (\mathbb{K}, \otimes)) \\
&= \bigoplus_{\mathrm{A} \to \boldsymbol{\alpha}'} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}') \otimes Z(\boldsymbol{\alpha}')
\end{aligned}
$$

It is in the first step that we make use of distributivity—we "separate" the rules from A from the rest of the rules in the derivation by moving them outside of the $\oplus$-sum. Since these rules may contain multiple non-terminals on the right-hand side, we need to $\otimes$-multiply over all such instances. We can additionally include terminals in this multiplication since we have defined the treesum of terminals to be $\mathbf{1}$.

This equation naturally leads to a system of non-linear equations. Note that the equation above is a *non-linear* polynomial function, of arbitrary order,[12] since we multiply over all non-terminals on the right-hand side of any given production rule. With some fixed indexing of the non-terminals $\mathrm{A}_1, ..., \mathrm{A}_\mathrm{n}$, we express the system as follows:

$$Z(\mathrm{A}_1) = \bigoplus_{\mathrm{A}_1 \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_1 \to \boldsymbol{\alpha}) \otimes Z(\boldsymbol{\alpha}) \tag{1.15}$$

$$Z(\mathrm{A}_2) = \bigoplus_{\mathrm{A}_2 \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_2 \to \boldsymbol{\alpha}) \otimes Z(\boldsymbol{\alpha}) \tag{1.16}$$

$$\vdots$$

$$Z(\mathrm{A}_\mathrm{i}) = \bigoplus_{\mathrm{A}_\mathrm{i} \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_\mathrm{i} \to \boldsymbol{\alpha}) \otimes Z(\boldsymbol{\alpha}) \tag{1.17}$$

$$\vdots$$

$$Z(\mathrm{A}_\mathrm{n}) = \bigoplus_{\mathrm{A}_\mathrm{n} \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_\mathrm{n} \to \boldsymbol{\alpha}) \otimes Z(\boldsymbol{\alpha}) \tag{1.18}$$

1.1.7 A Dynamic Program for Acyclic WCFGs

Again, following the exposition for pathsums in WFSAs, we start by giving an algorithm for the special case of acyclic WCFGs before moving on to the general, cyclic case. The notion of cyclicity in WCFGs is analogous to the notion of cyclicity in a WFSA. We give a formal definition below.

Definition 1.1.21. *A context-free grammar* $(\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ *is* ***acyclic*** *iff* $\forall \mathrm{A} \in \mathcal{N}$, $\mathrm{A} \notin \mathsf{desc}(\mathrm{A})$.

Much like acyclic WFSAs, acyclic WCFGs admit a very simple dynamic programming inference algorithm that runs in time linear in the size of the grammar. A key insight is that acyclic WCFGs

[12] Using the Chomsky normal form grammar transformation, as we will see in the next chapter, we can re-write the equation as one of second order.

derive a finite set of derivation trees, since there is no infinite recursion in the derivation relations. This allows us to construct an algorithm similar to the backward algorithm we discussed for WFSAs (**??**), making use of the distributive property to recursively $\oplus$-sum the weights of all production rules "backwards", where backwards in this case means in a bottom-up fashion.

By "backwards" ordering, we mean reverse topological ordering of the grammar. The topological ordering of a grammar is the topological ordering of the graph where the vertices are the non-terminals of the grammar and there is an edge from A to B if B is a child of A.

Definition 1.1.22 (Topological Ordering for Grammars)**.** *Let $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ be an acyclic CFG. Let $G = (V, E)$ be a directed graph formed as follows: $V = \mathcal{N}$ and*

$$E = \{(\mathrm{A}, \mathrm{B}) \mid (\mathrm{A}, \mathrm{B}) \in \mathcal{N}^2 \ \textbf{\textit{and}} \ \mathrm{B} \in \mathit{children}(\mathrm{A})\} \tag{1.19}$$

A topological sort of the non-terminals $\mathcal{N}$ of $\mathfrak{G}$, written as $\mathsf{topo}(\mathfrak{G})$ or $\mathsf{topo}(\mathcal{N})$, is the topological sort of the vertices V.

Note that a topological sort is only possible if the grammar is acyclic, as only then will the graph in the above definition be a directed acyclic one. The topological sort can be constructed in time linear of the size of the grammar. After that, for each non-terminal A, we simply $\oplus$-sum over all production rules where A is on the left-hand side to compute the treesum of A in one go. The reverse topological sort ensures that all treesums for non-terminals on the right-hand side of the production rules have already been computed. In addition, the treesums for the terminals are conveniently set to $\mathbf{1}$ in order to preserve the treesum values under $\otimes$-multiplication. We iterate once over all production rules in the grammar and multiply over the already computed treesums each time, giving an algorithm that runs in $\mathcal{O}(|\mathfrak{G}|)$. Pseudocode of the algorithm is given below.

Algorithm 1 Dynamic program for the treesum in acyclic WCFGs.

```
1. def AcyclicTreesum(𝔊, 𝒲):
2.     for A ∈ 𝒩 :                                  ▷Initialize non-terminal values to zero
3.         β(A) ← 0
4.     for a ∈ Σ :                                  ▷Set terminal values to one
5.         β(a) ← 1
6.     for A ∈ rtopo(𝔊) :                           ▷Visit non-terminals in reverse topological order
7.         β(A) ← ⨁_{(A→α)∈𝒫} 𝒲(A → α) ⊗ ⨂_{B∈α} β(B)
8.     return β(S)                                  ▷Note that S by definition comes last in the ordering
```

Note that the algorithm simply computes all non-linear equations defined above following the reverse topological ordering, in one single iteration. We can prove correctness of the algorithm using induction, similarly to the backward algorithm for WFSAs.

Theorem 1.1.3. *For an acyclic weighted context-free grammar $\mathfrak{G}$, AcyclicTreesum correctly computes the treesum.*

Proof. We consider an acyclic WCFG $\mathfrak{G}$ where we suppose that there exist k non-terminals with no non-terminal children. Since WCFG is acyclic, we must have $k \geq 1$. Let $\mathrm{A}_1, ..., \mathrm{A}_\mathrm{n}$ be a reverse topological ordering of $\mathfrak{G}$, in which $\mathrm{A}_1, ..., \mathrm{A}_\mathrm{k}$ will thus be without non-terminal children.

Base Case $(\mathrm{A}_1, \ldots, \mathrm{A}_k)$ In Line 7 of Alg. 1, the $\otimes$-product will reduce to $\mathbf{1}$ due to the lack of non-terminal children of $\mathrm{A}_1, \ldots, \mathrm{A}_k$. The $\boldsymbol{\beta}$ values computed will hence be the $\oplus$-sum over all production rules for which $\mathrm{A}_1, \ldots, \mathrm{A}_k$ are on the left-hand side, which is according to Def. 1.1.16. As such, we have that $\boldsymbol{\beta}(\mathrm{A}_i) = Z(\mathrm{A}_i), \forall i = 1, ..., k$.

Inductive Step $(\mathrm{A}_{k+1}, \ldots, \mathrm{A}_n)$ We want to show that for some non-terminal A_j, if $\boldsymbol{\beta}(\mathrm{A}_i)$ $\forall \mathrm{A}_i$ s.t. $\mathrm{A}_i \in \mathsf{children}(\mathrm{A}_j)$ are computed correctly, then so is $\boldsymbol{\beta}(\mathrm{A}_j)$. We plug in the correct treesums for A_i and follow the same derivation as we did in §1.1.6:

$$\begin{aligned}
\boldsymbol{\beta}(\mathrm{A}_j) &= \bigoplus_{\mathrm{A}_j \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_j \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{A}_i \in \boldsymbol{\alpha}} \boldsymbol{\beta}(\mathrm{A}_i) \\
&= \bigoplus_{\mathrm{A}_j \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A}_j \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{A}_i \in \boldsymbol{\alpha}} Z(\mathrm{A}_i) \\
&= Z(\mathrm{A}_j)
\end{aligned}$$

□

Relation to WFSAs

As should be clear, the above algorithm is a strict generalization of the finite-state case. Do note the structural similarity between the two – we iterate once over nodes in a reverse topological order and compute the sum.

To make this connection more formal, we must rely on the fact that all WFSAs may be encoded as right- or left-linear WCFGs. We start by defining a linear WCFG.

Definition 1.1.23 (Linear WCFG)**.** *A WCFG $\mathfrak{G}$ is **linear** if each of its production rules has at most one non-terminal on the right-hand side.*

Right- and left linear WCFGs are two special cases of linear grammars.

Definition 1.1.24 (Right-linear WCFG)**.** *A WCFG $\mathfrak{G}$ is **right-linear** if each of its production rules is of the form $\mathrm{A} \to \boldsymbol{y}\mathrm{B}$, with $\mathrm{B} \in \mathcal{N} \cup \{\varepsilon\}$ and $\boldsymbol{y} \in \Sigma^*$.*

Definition 1.1.25 (Left-linear WCFG)**.** *A WCFG $\mathfrak{G}$ is **left-linear** if each of its production rules is of the form $\mathrm{A} \to \mathrm{B}\boldsymbol{y}$, with $\mathrm{B} \in \mathcal{N} \cup \{\varepsilon\}$ and $\boldsymbol{y} \in \Sigma^*$.*

We can define a regular language as one that is generated by a right-linear or a left-linear grammar[13]. Not all linear grammars, however, generate a regular language. We refer back to Example 1.1.2 for an instance of a linear grammar that is not regular. We can transform the grammar in Example 1.1.2 into a weakly equivalent one consisting of only left-linear and right-linear rules, by introducing a new non-terminal B and replacing the rule $\mathrm{A} \to a\mathrm{A}b$ with the right-linear rule $\mathrm{A} \to a\mathrm{B}$ and the left-linear rule $\mathrm{B} \to \mathrm{A}b$. We can not, however, transform it into a grammar that consists *only* of either right-linear or left-linear rules, which implies that the language that it generates is not regular.

[13] Right- and left-linear grammars are weakly equivalent. See e.g. Révész (2015) for a conversion algorithm.

1.1.8 Fixed-Point Iteration

In the case of WCFGs, the acyclic case is unrealistically limiting. Indeed, even the simple grammars encountered in Example 1.1.1 and Example 1.1.2 are cyclic. Furthermore, acyclic grammars *always* have a finite language, and are thus regular. In this section we discuss an algorithm for computing the treesum for the general case of cyclic grammars, namely **fixed-point iteration**.

Having discussed the backwards algorithm, the extension to fixed-point is straightforward. In the general cyclic case, we cannot simply iterate over the system of equations backwards, since each treesum may rely on treesums that are yet to be computed. Instead, we do it repeatedly based on previously obtained approximations, until the treesums (hopefully) converge. We give the algorithm below.

Algorithm 2 The FixedPointGrammar algorithm for treesums.

def FixedPointGrammar($\mathfrak{G}$, $\mathcal{W}$):
  for $\mathrm{A} \in \mathcal{N}$: ▷*Initialize non-terminal values to zero*
    $\boldsymbol{\beta}^{(0)}(\mathrm{A}) \leftarrow \mathbf{0}$
  for $a \in \Sigma$: ▷*Set terminal values to one*
    $\boldsymbol{\beta}^{(0)}(a) \leftarrow \mathbf{1}$
  $n \leftarrow 1$
  repeat
    for $\mathrm{A} \in \mathcal{N}$: ▷*Initialize non-terminal values to zero*
      $\boldsymbol{\beta}^{(n)}(\mathrm{A}) \leftarrow \mathbf{0}$
    for $(\mathrm{A} \to \boldsymbol{\alpha}) \in \mathcal{P}$: ▷*Backwards sum over all production rules*
      $\boldsymbol{\beta}^{(n)}(\mathrm{A}) \leftarrow \boldsymbol{\beta}^{(n)}(\mathrm{A}) \oplus \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{B} \in \boldsymbol{\alpha}} \boldsymbol{\beta}^{(n-1)}(\mathrm{B})$
    for $a \in \Sigma$: ▷*Set terminal values to one*
      $\boldsymbol{\beta}^{(n)}(a) \leftarrow \mathbf{1}$
    $n \leftarrow n + 1$
  until $\boldsymbol{\beta}^{(n)} \approx \boldsymbol{\beta}^{(n-1)}$
  return $\boldsymbol{\beta}^{(n)}(\mathrm{S})$

Proving correctness for the fixed-point iteration is a bit more tricky than for the acyclic case. We start by showing that each iteration of Alg. 2 correctly computes an approximation of the treesum that only accounts for trees with height[14] up until an upper bound. For a non-terminal $\mathrm{A} \in \mathcal{N}$, we define this quantity as:

$$Z_k(\mathrm{A}) = \bigoplus_{\substack{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A}): \\ \mathsf{height}(\boldsymbol{t}) \leq k}} \mathcal{W}(\boldsymbol{t}), \quad k \in \{1, 2, \dots\} \tag{1.20}$$

We can now formalize and prove the aforementioned proposition.

Proposition 1.1.5. *Each step of FixedPointGrammar correctly computes $Z_k(\mathrm{A})$ for all non-terminals in $\mathcal{N}$, i.e., $\boldsymbol{\beta}^{(k)}(\mathrm{A}) = Z_k(\mathrm{A}), \forall \mathrm{A} \in \mathcal{N}$.*

Proof. As is standard practice in this course, we show this by induction.

[14] The height of a tree is defined as the length of the longest path from the root node to any leaf node in the tree. For instance, a derivation (sub)tree consisting of only one production rule has height 1.

Base Case ($\boldsymbol{\beta}^{(1)}(\mathrm{A}) = Z_1(\mathrm{A})$) We start by showing that $\boldsymbol{\beta}^{(1)}(\mathrm{A})$ correctly computes $Z_1(\mathrm{A})$ after iterating over all the production rules in Line 10 of Alg. 2, for all non-terminals A. We have that

$$\boldsymbol{\beta}^{(1)}(\mathrm{A}) = \sum_{(\mathrm{A} \to \boldsymbol{\alpha}) \in \mathcal{P}} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{B} \in \boldsymbol{\alpha}} \boldsymbol{\beta}^{0}(\mathrm{B}). \tag{1.21}$$

Since we have that $\boldsymbol{\beta}^{(0)}(\mathrm{A}) = \mathbf{0}$ for all $\mathrm{A} \in \mathcal{N}$, only the terms in the above equation such that there is no non-terminal on the right-hand side $\boldsymbol{\alpha}$ will be $\neq \mathbf{0}$. If there is no non-terminal on the right-hand side, we cannot apply any further production rules and hence we end up with a tree of height 1. All other possible derivation trees in $\mathcal{D}_{\mathfrak{G}}(\mathrm{A})$ will not be accounted for. Consequentially, we get that

$$\boldsymbol{\beta}^{(1)}(\mathrm{A}) = \bigoplus_{\substack{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{B}): \\ \mathsf{height}(\boldsymbol{t}) \leq 1}} \mathcal{W}(\boldsymbol{t}), \tag{1.22}$$

which by definition is equal to $Z_1(\mathrm{A})$. This finishes the base case.

Inductive Step ($\boldsymbol{\beta}^{(k)}(\mathrm{A}) = Z_k(\mathrm{A}) \Rightarrow \boldsymbol{\beta}^{(k+1)}(\mathrm{A}) = Z_{k+1}(\mathrm{A})$) Next, we want to show that, assuming that $\boldsymbol{\beta}^{(k)}(\mathrm{A})$ is correct, so is $\boldsymbol{\beta}^{(k+1)}(\mathrm{A})$. Note that given a set of trees of height k, we get a tree of height $k+1$ by adding a new root node with an edge to each of the root nodes of the subtrees with height k. This principle is used in the inductive step.

$$\begin{aligned}
\boldsymbol{\beta}^{(k+1)}(\mathrm{A}) &= \bigoplus_{\mathrm{A} \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{B} \in \boldsymbol{\alpha}} \boldsymbol{\beta}^{(k)}(\mathrm{B}) && \text{(line 11)} \\
&= \bigoplus_{\mathrm{A} \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{B} \in \boldsymbol{\alpha}} Z_k(\mathrm{B}) && \text{(inductive step)} \\
&= \bigoplus_{\substack{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A}): \\ \mathsf{height}(\boldsymbol{t}) \leq k+1}} \mathcal{W}(\boldsymbol{t}) && \text{(distributive property)} \\
&= Z_{k+1}(\mathrm{A}) && \text{(definition)}
\end{aligned}$$

This concludes the proof. □

Prop. 1.1.5 does not alone, however, show that Alg. 2 correctly computes the treesum. We need to additionally convince ourselves that in the limit, the $\boldsymbol{\beta}$ values will be equal to the treesums, i.e. that

$$\lim_{k \to \infty} \boldsymbol{\beta}^{(k)}(\mathrm{A}) = Z(\mathrm{A}), \quad \forall \mathrm{A} \in \mathcal{N}. \tag{1.23}$$

This will not in general be the case, as it depends on "numerical convergence" of the series (in quotation marks as there may not exist such a notion for a particular semiring). Hence, we do not have any general guarantees on the runtime for fixed-point.[15] If we assume convergence, however, for semirings for which we have a clear notion of convergence, e.g. the Real semiring, it is relatively straightforward to prove correctness of fixed-point.

Theorem 1.1.4. *Let $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a WCFG, with a Real semiring $\mathcal{W}$. Further assume that the treesum $Z(\mathfrak{G})$ is convergent, $Z(\mathfrak{G}) < \infty$. We then have that FixedPointGrammar converges to $Z(\mathfrak{G})$ in the limit.*

[15] We can get a speed-up of the algorithm by preceding it with a partitioning of the non-terminals into a partial order, such that we save unnecessary computations. See Nederhof and Satta (2009) for details.

Proof. We write the treesums as

$$Z(\mathrm{A}) = \sum_{l=1}^{\infty} \underbrace{\sum_{\substack{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A}): \\ \mathsf{height}(\boldsymbol{t})=l}} \mathcal{W}(\boldsymbol{t})}_{\stackrel{\text{def}}{=} a_l}, \quad \forall \mathrm{A} \in \mathcal{N} \tag{1.24}$$

where we have that $Z(\mathrm{A}) < \infty$ for all $\mathrm{A} \in \mathcal{N}$. With the treesums being convergent, we know that they must fulfill Cauchy's convergence criterion. Hence, for every $\varepsilon > 0$, $\exists N$ such that

$$|a_{m+1} + \cdots + a_{m+p}| < \varepsilon, \quad \forall m > N, \forall p > 1. \tag{1.25}$$

This implies that

$$|Z_{m+p}(\mathrm{A}) - Z_m(\mathrm{A})| < \varepsilon \tag{1.26}$$

since by definition we have

$$Z_k(\mathrm{A}) = \sum_{l=1}^{k} a_l. \tag{1.27}$$

Hence,

$$\lim_{k \to \infty} Z_k(\mathrm{X}) = Z(\mathrm{A}), \quad \forall \mathrm{A} \in \mathcal{N}. \tag{1.28}$$

By Prop. 1.1.5 we have that *Alg.* 2 computes the $\boldsymbol{\beta}^{(k)}$ values such that $\boldsymbol{\beta}^{(k)}(\mathrm{A}) = Z_k(\mathrm{A})$, which concludes the proof. □

One application of the treesum, and thus of fixed-point iteration for grammars, is to find all generating non-terminals in a grammar.

Example 1.1.6 (Finding Generating Non-terminals)**.** *Note that for the Boolean semiring, the treesum for each non-terminal will be a binary variable corresponding to whether that non-terminal derives a string in the language of the grammar or not. That is the exact definition of generating. Hence, we can apply Alg. 2 to find all* ***generating*** *non-terminals in a weighted context-free grammar* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$*. We simply booleanize the grammar by taking all rules* $X \to \boldsymbol{\alpha}$ *in* $\mathcal{P}$ *and replacing their weight with* 1 *if* $\mathcal{W}(X \to \boldsymbol{\alpha}) \neq \boldsymbol{0}$*, and* 0 *otherwise. We leave as an exercise to construct an algorithm that finds all* ***reachable*** *symbols in a grammar.*

1.2 Newton's Method

Fixed-point iteration as presented is a relatively simple algorithm to understand, but it is not the most efficient. In particular, recall how it sums over all trees with a given height in each step. Since the number of trees with a given height is finite, we can only sum over a finite number of trees in each derivation. If a grammar is characterized by having many complex and deep derivation trees, it may thus take many iterations to converge. In this section, we give an alternative method that allows for enumerating over an *infinite* number of trees in each step. This method is known as **Newton's method**.

You may already be familiar with Newton's method in the context of the real semiring, in which case it can be said to approximate the roots of a function. Consider a function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ and its Jacobian matrix $\mathbf{J} \in \mathbb{R}^{n \times n}$. Let $\boldsymbol{\nu}_0 \in \mathbb{R}^n$ be a vector of initial guesses of the roots of F. Newton's method then follows by repeated application of the following formula:[16]

$$\boldsymbol{\nu}_{d+1} = \boldsymbol{\nu}_d - \mathbf{J}^{-1}(\boldsymbol{\nu}_d)F(\boldsymbol{\nu}_d) \tag{1.29}$$

It can then be shown that under certain conditions, $\boldsymbol{\nu}_{d+1}$ is a better approximation of the root than $\boldsymbol{\nu}_d$ for all positive integers d.

In this section we present an algorithm that works on not only the real semiring, but on any commutative semiring[17] (first introduced by Esparza et al. (2010)). We will then relate back to the equation above to show how this algorithm is a generalization of Newton's method over the reals.

Newton's method for WCFGs

Our discussion requires some new definitions. We say that a **direct subtree** of a tree $\boldsymbol{t}$ is a subtree rooted at a child of the root node of $\boldsymbol{t}$. We can then introduce the notion of **tree dimension**, also known as Strahler number.

Definition 1.2.1 (Dimension)**.** *Let $\boldsymbol{t}$ be a tree. The **dimension** of $\boldsymbol{t}$, $\mathsf{dim}(\boldsymbol{t})$, is defined inductively as follows:*

- $\mathsf{dim}(\boldsymbol{t}) = 0$ *if $\boldsymbol{t}$ is a single node*
- *Let $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k$ be the direct subtrees of $\boldsymbol{t}$, and $D = \max_i\{\mathsf{dim}(\boldsymbol{t}_i) \mid i = 1, \ldots, k\}$. Then:*

$$\mathsf{dim}(\boldsymbol{t}) \stackrel{\text{def}}{=} \begin{cases} D + 1, & \text{if } \exists i \neq j : D = \mathsf{dim}(\boldsymbol{t}_i) = \mathsf{dim}(\boldsymbol{t}_j) \\ D, & \text{otherwise} \end{cases} \tag{1.30}$$

See Fig. 1.3 for an example of a tree with dimension 3. Note that, with the exception of the base case, the set of possible trees with a given dimension is (countably) infinite. Further, for a given tree $\boldsymbol{t}$, $\mathsf{dim}(\boldsymbol{t})$ is the maximum over the heights of the perfect binary subtrees of $\boldsymbol{t}$. We leave it as an exercise to prove this. It follows as a corollary that for any tree $\boldsymbol{t}$, $\mathsf{dim}(\boldsymbol{t}) \leq \mathsf{height}(\boldsymbol{t})$.

Now that we have introduced the concept of tree dimension, we can be a bit more precise about the statement we made about Newton's in the beginning of this section: Newton's method enumerates trees by dimension. This means that for each iteration d, Newton's adds all trees of dimension d to the approximation of the treesum. The set of trees with a given dimension is infinite. Newton's method thus iterates over an infinite number of trees in each iteration. This is made

[16] Assuming that $\mathbf{J}(\boldsymbol{\nu})$ is an invertible matrix for all Newton approximations $\boldsymbol{\nu}$.

[17] In principle, we can extend to the non-commutative case as well provided that we have an efficient algorithm that can solve a system of linear equations in a non-commutative semiring.

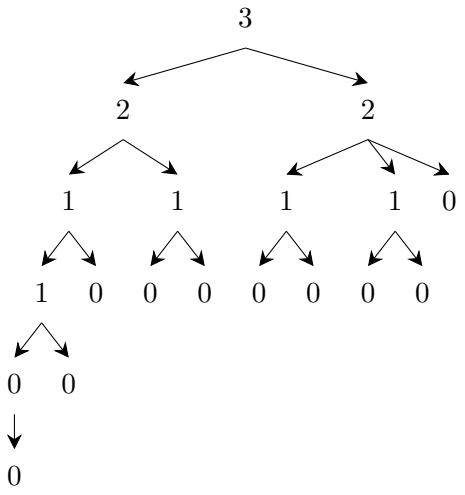


Figure 1.3: A tree for which each node is annotated with the dimension of the subtree rooted at that node. Note in particular that there may be an infinite number of trees with a given dimension. For instance, all trees created by only applying unary rules have dimension 0.

efficient by the following property of context-free grammars as relates to tree dimension: For every set of trees with dimension d, there exists a linear map from the weights of trees with dimension $< d$ to the weights of trees with dimension equal to d.

We map $\mathfrak{G}$ into a grammar $\mathfrak{G}^{<D}$ such that all derivation trees in $\mathfrak{G}^{<D}$ have dimension less than D and there is an injective map from the derivations in $\mathfrak{G}^{<D}$ to the derivations in $\mathfrak{G}$ which preserves dimension, weight and yield. This injective map becomes bijective if we only consider the trees with dimension less than D in $\mathfrak{G}$. In $\mathfrak{G}^{<D}$ we introduce two new types of nonterminals. Nonterminals on the form $\mathrm{A}^{=d}$, with $d = 0, \ldots, D$, have the interpretation that the subtrees rooted at $\mathrm{A}^{=d}$ have dimension equal to d. Nonterminals on the form $\mathrm{A}^{<d}$, with $d = 1, \ldots, D$, have the interpretation that subtrees rooted at $\mathrm{A}^{<d}$ have some dimension that is less than d. The **dimension transform** is defined as follows.

Transform 1.2.1 (Dimension transform)**.** *The **dimension transform** $\mathcal{T}_D(\mathfrak{G}, D)$ returns a WCFG $\mathfrak{G}^{<D}$ defined by the following rules:*

$$\mathrm{A}^{=0} \xrightarrow{w} \boldsymbol{\alpha} \qquad \forall \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha} \in \mathcal{P}, \boldsymbol{\alpha} \in \Sigma^* \tag{1.31}$$

$$\mathrm{A}^{=0} \xrightarrow{w} \boldsymbol{\alpha}\, \mathrm{B}^{=0} \boldsymbol{\beta} \qquad \forall \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\, \mathrm{B} \boldsymbol{\beta} \in \mathcal{P}, \boldsymbol{\alpha}, \boldsymbol{\beta} \in \Sigma^*, \mathrm{B} \in \mathcal{N} \tag{1.32}$$

$$\mathrm{A}^{<d} \xrightarrow{1} \mathrm{A}^{=e} \qquad 0 \leq e < d, d = 1, \ldots, D,\ \forall \mathrm{A} \in \mathcal{N} \tag{1.33}$$

$$\begin{aligned} \mathrm{A}^{=d} \xrightarrow{w} \boldsymbol{\alpha}_0\, \mathrm{B}_1\, \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\, \mathrm{B}_r\, \boldsymbol{\alpha}_r \qquad & \forall \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}_0\, \mathrm{A}_1\, \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\, \mathrm{A}_r\, \boldsymbol{\alpha}_r \in \mathcal{P}, r > 1 \\ & d = 1, \ldots, D, \forall \mathcal{I} \subseteq \{1, \ldots, r\}, |\, \mathcal{I}\, | \geq 2 : \\ & \mathrm{B}_i \stackrel{\text{def}}{=} \mathrm{A}_i^{=d-1}, \forall i \in \mathcal{I} \\ & \mathrm{B}_i \stackrel{\text{def}}{=} \mathrm{A}_i^{<d-1}, \forall i \in \{1, \ldots, r\} \setminus \mathcal{I} \end{aligned} \tag{1.34}$$

$$\begin{aligned} \mathrm{A}^{=d} \xrightarrow{w} \boldsymbol{\alpha}_0\, \mathrm{B}_1\, \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\, \mathrm{B}_r\, \boldsymbol{\alpha}_r \qquad & \forall \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}_0\, \mathrm{A}_1\, \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\, \mathrm{A}_r\, \boldsymbol{\alpha}_r \in \mathcal{P}, r \geq 1 \\ & d = 1, \ldots, D, \forall j \in \{1, \ldots, r\} : \\ & \mathrm{B}_j \stackrel{\text{def}}{=} \mathrm{A}_j^{=d}, \quad \mathrm{B}_i \stackrel{\text{def}}{=} \mathrm{A}_i^{<d}, \forall i \in \{1, \ldots, r\} \setminus \{j\} \end{aligned} \tag{1.35}$$

As you will notice, the dimension transform considers all cases in which a subtree rooted at some nonterminal $\mathrm{A}^{=d}$ can have dimension d. The base cases of $d = 0$ are covered by Eq. (1.31) and Eq. (1.32). Inductively, a tree can have dimension equal to d in one of two ways: (i) the recursive case of having exactly one direct subtree with dimension equal to d (handled by Eq. (1.32) and Eq. (1.35)) and (ii) the incremental case of having at least two direct subtrees with dimension equal to $d-1$ (handled by Eq. (1.34)). Lower dimension trees are recovered using the rules in Eq. (1.33). We provide a proof below.

Lemma 1.2.1. *Every tree $\boldsymbol{t}$ rooted at some nonterminal $\mathrm{A}^{=\mathrm{d}}$ has dimension exactly d and every tree $\boldsymbol{t}$ rooted at some nonterminal $\mathrm{A}^{<\mathrm{d}}$ has dimension less than d.*

Proof. We prove the above statements by induction on d.

Base Case In the case of $\mathrm{A}^{=\mathrm{d}}$ trees the base case is $d = 0$ and in the case of $\mathrm{A}^{<\mathrm{d}}$ trees the base case is $d = 1$.

- $\boldsymbol{t}$ rooted at $\mathrm{A}^{=0}$. There are two types of rules where $\mathrm{A}^{=0}$ can be the left hand side: $\mathrm{A}^{=0} \to \boldsymbol{\alpha}$ and $\mathrm{A}^{=0} \to \boldsymbol{\alpha}\ \mathrm{B}^{=0}\boldsymbol{\beta}$. Thus $\boldsymbol{t}$ has dimension 0.
- $\boldsymbol{t}$ rooted at $\mathrm{A}^{<1}$. There is one type of rule where $\mathrm{A}^{<1}$ can be the left hand side: $\mathrm{A}^{<1} \to \mathrm{A}^{=0}$. Since the tree rooted at $\mathrm{A}^{=0}$ has dimension 0, $\boldsymbol{t}$ must as well.

Inductive Step

- $\boldsymbol{t}$ rooted at $\mathrm{A}^{=\mathrm{d}}, d > 0$. There are two types of rules where $\mathrm{A}^{=\mathrm{d}}$ can be the left hand side: Eq. (1.34) and Eq. (1.35). For Eq. (1.34), there exists at least two direct subtrees for which the root nonterminal is superscripted by "$= d-1$", and all other nonterminals are superscripted by "$= d'$" with $d' < d$. By induction, the trees for which the root nonterminal is superscripted by "$d-1$" have dimension $d-1$. It follows by the definition of dimension that $\boldsymbol{t}$ has dimension d. For Eq. (1.35), there exists by construction a direct subtree of $\boldsymbol{t}$, $\boldsymbol{t}'$, for which the root nonterminal is superscripted by "$= d$". Thus, the dimension of $\boldsymbol{t}$ must be equal to the dimension of $\boldsymbol{t}'$. By induction on the length of this chain we get that the dimension of $\boldsymbol{t}$ is d: The base case is Eq. (1.34) and the inductive step preserves the dimension.
- $\boldsymbol{t}$ rooted at $\mathrm{A}^{<\mathrm{d}}, d > 1$. There is one type of rule where $\mathrm{A}^{<\mathrm{d}}$ can be the left hand side: $\mathrm{A}^{<d} \to \mathrm{A}^{=e}, 0 \leq e < d$. By induction, the subtrees rooted at $\mathrm{A}^{=\mathrm{e}}$ will have dimension e, and therefore it follows that $\boldsymbol{t}$ has dimension less than d.

□

Now, we can show that the treesums in $\mathfrak{G}^{<D}$ correspond to the treesums in $\mathfrak{G}$.

Lemma 1.2.2. *There exists a yield- and weight preserving bijection between the trees in $\mathfrak{G}^{<D}$ rooted at nonterminal $\mathrm{A}^{=\mathrm{d}}$ and those trees in $\mathfrak{G}$ rooted at nonterminal A that have dimension exactly d. Similarly, there exists a yield- and weight preserving bijection between the trees in $\mathfrak{G}^{<D}$ rooted at nonterminal $\mathrm{A}^{<\mathrm{d}}$ and those trees in $\mathfrak{G}$ rooted at nonterminal A that have dimension less than d.*

Proof. We construct a mapping $\boldsymbol{\phi}$ from the derivation trees in $\mathfrak{G}^{<D}$ to derivation trees in $\mathfrak{G}$. Consider a tree $\boldsymbol{t}$ in $\mathfrak{G}^{<D}$ with some root rule p. $\boldsymbol{\phi}$ is defined recursively through the following two cases:

- p takes the form $\mathrm{A}^{<d} \xrightarrow{\mathbf{1}} \mathrm{A}^{=e}$. $\boldsymbol{\phi}$ then maps $\boldsymbol{t}$ to the tree rooted at $\mathrm{A}^{=\mathrm{e}}$, and removes the superscript from $\mathrm{A}^{=\mathrm{e}}$.

- p takes any other form, in general $\mathrm{A}^{=d} \xrightarrow{w} \boldsymbol{\alpha}_0\ \mathrm{B}_1\ \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\ \mathrm{B}_r\ \boldsymbol{\alpha}_r, r \geq 1$. $\boldsymbol{\phi}$ then maps $\boldsymbol{t}$ to the tree with root rule $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}_0\ \mathrm{B}'_1\ \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\ \mathrm{B}'_r\ \boldsymbol{\alpha}_r, r \geq 1$, where B' results from removing the superscript from B, and the subtrees are mapped recursively with $\boldsymbol{\phi}$.

Note that nonterminals of the form $\mathrm{A}^{<\mathrm{d}}$ can only rewrite as $\mathrm{A}^{=\mathrm{e}}$ for some $e < d$, with weight $\mathbf{1}$. Adding such a rule at the root of a tree will therefore not alter its yield or weight. Further note that the rules in $\mathfrak{G}^{<D}$ with a left-hand side of the form $\mathrm{A}^{=\mathrm{d}}$ are created by taking rules in $\mathfrak{G}$ and introducing superscripts (by definition in Transform 1.2.1). Thus, when removing the superscripts from the rules, as $\boldsymbol{\phi}$ does, both weight and yield are preserved.

It remains to show that $\boldsymbol{\phi}$ bijectively maps trees rooted at $\mathrm{A}^{=\mathrm{d}}, d \geq 0$ to trees with dimension d and that it, in separation, bijectively maps trees rooted at $\mathrm{A}^{<\mathrm{d}}, d > 0$ to trees with dimension less than d. We thus show that for every tree $\boldsymbol{t}'$ with $\dim(\boldsymbol{t}') = k$ in the grammar $\mathfrak{G}$, there is both a unique tree rooted at some $\mathrm{A}^{=\mathrm{d}}$ and a unique tree rooted at some $\mathrm{A}^{<\mathrm{d}}$ for every $d > k$ in $\mathfrak{G}^{<D}$. We proceed by structural induction on $\boldsymbol{t}'$.

Base Case Let $\boldsymbol{t}'$ be rooted at A with $\mathrm{A} \to \boldsymbol{\alpha}, \boldsymbol{\alpha} \in \Sigma^*$. By construction there exists a tree in $\mathfrak{G}^{<D}$ rooted at $\mathrm{A}^{=0} \to \boldsymbol{\alpha}$. In addition, there exists some rule $\mathrm{A}^{<\mathrm{d}} \to \mathrm{A}^{=0}$ for all $0 < d \leq D$. Combining these with $\mathrm{A}^{=0} \to \boldsymbol{\alpha}$ yield trees that are unique to $\boldsymbol{t}'$.

Inductive Step #1 Let $\boldsymbol{t}'$ be rooted at A with $\mathrm{A} \to \boldsymbol{\alpha}\ \mathrm{B}\ \boldsymbol{\beta}$, with dimension k. There exists a tree in $\mathfrak{G}^{<D}$ that is rooted at $\mathrm{A}^{=\mathrm{k}}$ with $\mathrm{A}^{=\mathrm{k}} \to \boldsymbol{\alpha}\ \mathrm{B}^{=\mathrm{k}}\ \boldsymbol{\beta}$, which is unique by induction. Analogously to the base case, there exists a unique tree rooted at $\mathrm{A}^{<\mathrm{k}}$ with $\mathrm{A}^{<\mathrm{d}} \to \mathrm{A}^{=\mathrm{k}}$, for all $k < d \leq D$, which is unique.

Inductive Step #2 Let $\boldsymbol{t}'$ be rooted at A with $\mathrm{A} \to \boldsymbol{\alpha}_0\ \mathrm{B}_1\ \boldsymbol{\alpha}_1 \ldots \boldsymbol{\alpha}_{r-1}\ \mathrm{B}_r\ \boldsymbol{\alpha}_r$, with dimension k. We perform induction on the length of the chain of rules for which the subtree rooted at the left-hand side nonterminal preserves dimension. If $\boldsymbol{t}'$ has dimension larger than its subtrees (base case), then there exist at least two direct subtrees with dimension $k-1$. Let I be the set of indices for the nonterminals at which those trees are rooted. The corresponding tree to $\boldsymbol{t}'$ in $\mathfrak{G}^{<D}$ is then the tree rooted at $\mathrm{A}^{=\mathrm{k}}$ with rule as in Eq. (1.34), where the right-hand side nonterminals superscripted by "$= k-1$" are those in I. By induction there are unique trees in $\mathfrak{G}^{<D}$ for each of the direct subtrees of $\boldsymbol{t}'$. Now consider the case where there is a unique direct subtree of $\boldsymbol{t}'$ that has dimension k (inductive step). Consequently, all other direct subtrees have some dimension $< k$. It follows analogously that there exist unique trees in $\mathfrak{G}^{<D}$ via rule Eq. (1.35) at the root. The case for trees rooted at $\mathrm{A}^{<\mathrm{d}}$ follows in a similar fashion. □

Given the treesum correspondence, the above transform thus suggests a treesum algorithm in which we simply enumerate over the treesum approximations in $\mathfrak{G}^{<D}$ in increasing dimension. However, note that the number of introduced rules scales exponentially in the length of the production rules through Eq. (1.34). Therefore, it might not be wise to implement an algorithm directly based on the above transform if we want a fast algorithm that deals with any general grammars.[18] Luckily, there does exist a rather simple grammar form that we can make use of – the **Chomsky Normal Form** (CNF). All context-free grammars can be mapped to a grammar in CNF which has the same language and string sums as the original grammar. We will revisit this topic in depth in the next section, but for now let us just define CNF.

[18] There are tricks we can use that involve storing further variables and adding intermediate nonterminals, if we do not wish to take the CNF route.

Definition 1.2.2. *A context-free grammar is in* ***Chomsky normal form*** *if every production rule is on one of the following forms:*

- $\mathrm{A} \to \mathrm{B}\ \mathrm{C}$*, with* $\mathrm{A} \in \mathcal{N}$ *and* $\mathrm{B}, \mathrm{C} \in \mathcal{N} \setminus \{\mathrm{S}\}$*;*
- $\mathrm{A} \to \mathrm{a}$*, with* $\mathrm{A} \in \mathcal{N}$ *and* $\mathrm{a} \in \Sigma$*;*
- $\mathrm{S} \to \varepsilon$*.*

Assuming a grammar in CNF, the dimension transform can be simplified to the following.

Transform 1.2.2 (Dimension transform for CNF grammars)**.** *The* ***dimension transform*** $\mathcal{T}_D(\mathfrak{G}, D)$ *for a WCFG* $\mathfrak{G}$ *in CNF, returns a WCFG* $\mathfrak{G}^{<D}$ *defined by the following rules:*

$$\mathrm{A}^{=0} \xrightarrow{w} \mathrm{a} \qquad \forall \mathrm{A} \xrightarrow{w} \mathrm{a} \in \mathcal{P}, \mathrm{a} \in \Sigma \cup \{\varepsilon\} \tag{1.36}$$

$$\mathrm{A}^{<d} \xrightarrow{\mathbf{1}} \mathrm{A}^{=e} \qquad 0 \leq e < d, d = 2, \ldots, D,\ \forall \mathrm{A} \in \mathcal{N} \tag{1.37}$$

$$\mathrm{A}^{=d} \xrightarrow{w} \mathrm{B}^{=d}\ \mathrm{C}^{<d} \mid \mathrm{B}^{<d}\ \mathrm{C}^{=d} \mid \mathrm{B}^{=d-1}\ \mathrm{C}^{=d-1} \qquad d = 1, \ldots, D-1,\ \forall \mathrm{A} \xrightarrow{w} \mathrm{B}\ \mathrm{C} \tag{1.38}$$

With the CNF dimension transform defined, we can easily derive an algorithm for treesum computation. We simply enumerate over the nonterminals of $\mathfrak{G}^{<D}$ in increasing d. The base case of 0-dimension trees constitutes to for each nonterminal summing over the weight of all its terminal rules, which for a grammar in CNF span all trees with dimension 0. In the inductive case for a given d, all nonterminals $\mathrm{A}^{=\mathrm{d}'}$ for which $d' < d$ will already be solved for. Thus, each step of Newton's method reduces to solving a system of *linear* equations instead of a polynomial one, which is a much easier problem (that we studied extensively when discussing WFSAs!).

To make this clear, consider the rule $\mathrm{A} \xrightarrow{w} \mathrm{B}\ \mathrm{C}$ in the original grammar, and for the sake of simplicity, assume there is only this one rule with left-hand side A in $\mathcal{P}$. This rule will be represented by the three rules $\mathrm{A}^{=d} \xrightarrow{w} \mathrm{B}^{=d}\ \mathrm{C}^{<d} \mid \mathrm{B}^{<d}\ \mathrm{C}^{=d} \mid \mathrm{B}^{=d-1}\ \mathrm{C}^{=d-1}$ in the transformed grammar. We can write the treesum of $\mathrm{A}^{=\mathrm{d}}$ as follows:

$$\begin{aligned} Z(\mathrm{A}^{=d}) = {} & \mathcal{W}(\mathrm{A}^{=d} \to \mathrm{B}^{=d}\ \mathrm{C}^{<d}) \otimes Z(\mathrm{B}^{=d}) \otimes Z(\mathrm{C}^{<d}) \\ & \oplus \mathcal{W}(\mathrm{A}^{=d} \to \mathrm{B}^{<d}\ \mathrm{C}^{=d}) \otimes Z(\mathrm{B}^{<d}) \otimes Z(\mathrm{C}^{=d}) \\ & \oplus \mathcal{W}(\mathrm{A}^{=d} \to \mathrm{B}^{=d-1}\ \mathrm{C}^{=d-1}) \otimes Z(\mathrm{B}^{=d-1}) \otimes Z(\mathrm{C}^{=d-1}) \end{aligned} \tag{1.39}$$

Enumerating over the treesums in increasing dimension starting with the base case, $Z(\mathrm{B}^{<d})$, $Z(\mathrm{C}^{<d})$, $Z(\mathrm{B}^{=d-1})$ and $Z(\mathrm{C}^{=d-1})$ have all already been solved. Thus Eq. (1.39) is simply a linear equation in $Z(\mathrm{B}^{=d})$ and $Z(\mathrm{C}^{=d})$. We arrive at the pseudocode in Alg. 3.

Short note on solving systems of linear equations In the case of commutative semirings, we can write Eq. (1.39) as a right-linear equation, in which the variables $Z(\mathrm{B}^{=d})$ and $Z(\mathrm{C}^{=d})$ are pushed to the right in the factors they participate in. We can then write our system on the following form

$$\mathbf{z} = \mathbf{A} \otimes \mathbf{z} \oplus \mathbf{b} \tag{1.40}$$

where the elements of $\mathbf{z} \in \mathbb{R}^n$ are the variables. The solution of such a system is given by $\mathbf{A}^* \otimes \mathbf{b}$, where $\mathbf{A}^*$ can be computed using Lehmann's algorithm.

Algorithm 3 Newton's algorithm to compute treesums for WCFGs in CNF.

def Newton($\mathfrak{G}$, $\mathfrak{G}^{<D}$, $\mathcal{W}$):
  ▷ *Base case*
  for $\mathrm{A}^{=0} \to \mathrm{a} \in \mathcal{P}^{<D}$: ▷*Sum over trees of dimension* 0
    $\boldsymbol{\beta}(\mathrm{A}^{=0}) \leftarrow \boldsymbol{\beta}(\mathrm{A}^{=0}) \oplus \mathcal{W}(\mathrm{A}^{=0} \to \mathrm{a})$
  for $\mathrm{A}^{<1} \in \mathcal{N}^{<D}$:
    $\boldsymbol{\beta}(\mathrm{A}^{<1}) \leftarrow \boldsymbol{\beta}(\mathrm{A}^{=0})$
  ▷ *Inductive step*
  for $d = 1, \ldots, D-1$:
    for $\mathrm{A} \to \mathrm{B}\ \mathrm{C} \in \mathcal{P}$: ▷*For all binary rules, sum over all ways to get trees of dimension d*
      ▷ *Note that all* $\boldsymbol{\beta}(\mathrm{X}^{=\mathrm{d}})$*'s are unknown*
      $\boldsymbol{\beta}(\mathrm{A}^{=\mathrm{d}}) \leftarrow \boldsymbol{\beta}(\mathrm{A}^{=\mathrm{d}})$
        $\oplus\, \mathcal{W}(\mathrm{A}^{=\mathrm{d}} \to \mathrm{B}^{=\mathrm{d}}\ \mathrm{C}^{<\mathrm{d}}) \otimes \boldsymbol{\beta}(\mathrm{B}^{=\mathrm{d}}) \otimes \boldsymbol{\beta}(\mathrm{C}^{<\mathrm{d}})$
        $\oplus\, \mathcal{W}(\mathrm{A}^{=\mathrm{d}} \to \mathrm{B}^{<\mathrm{d}}\ \mathrm{C}^{=\mathrm{d}}) \otimes \boldsymbol{\beta}(\mathrm{B}^{<\mathrm{d}}) \otimes \boldsymbol{\beta}(\mathrm{C}^{=\mathrm{d}})$
        $\oplus\, \mathcal{W}(\mathrm{A}^{=\mathrm{d}} \to \mathrm{B}^{=\mathrm{d}-1}\ \mathrm{C}^{=\mathrm{d}-1}) \otimes \boldsymbol{\beta}(\mathrm{B}^{=\mathrm{d}-1}) \otimes \boldsymbol{\beta}(\mathrm{C}^{=\mathrm{d}-1})$
    $\boldsymbol{\beta}(\mathrm{A}_1^{=\mathrm{d}}), \ldots, \boldsymbol{\beta}(\mathrm{A}_\mathrm{n}^{=\mathrm{d}}) \leftarrow \mathrm{LinearSolver}(\boldsymbol{\beta}(\mathrm{A}_1^{=\mathrm{d}}), \ldots, \boldsymbol{\beta}(\mathrm{A}_\mathrm{n}^{=\mathrm{d}}))$ ▷*Solve system of linear equations*
    for $\mathrm{A}^{<\mathrm{d}+1} \in \mathcal{N}^{<D}$: ▷*Update treesum values*
      $\boldsymbol{\beta}(\mathrm{A}^{<\mathrm{d}+1}) \leftarrow \boldsymbol{\beta}(\mathrm{A}^{<\mathrm{d}}) \oplus \boldsymbol{\beta}(\mathrm{A}^{=\mathrm{d}})$
  for $\mathrm{A} \in \mathcal{N}$:
    $\boldsymbol{\beta}(\mathrm{A}) \leftarrow \boldsymbol{\beta}(\mathrm{A}^{<\mathrm{D}})$
  return $\boldsymbol{\beta}(\mathrm{A}_1), \ldots, \boldsymbol{\beta}(\mathrm{A}_\mathrm{n})$

Sketching the connection to the $(+, \times)$-semiring case

Let us now establish a connection between the above algorithm and the Newton's method of Eq. (1.29) that we all know and love. Consider the Real semiring. First, note that our case is restricted to functions on the form $G(\mathbf{x}) \overset{\text{def}}{=} F(\mathbf{x}) - \mathbf{x}$, where $F_i(\mathbf{x})$ are polynomials. Eq. (1.29) then rewrites as[19]

$$\begin{aligned} \boldsymbol{\nu}_{k+1} &= \boldsymbol{\nu}_k - (\mathbf{J}(\boldsymbol{\nu}_k) - \mathbf{I})^{-1}(F(\boldsymbol{\nu}_k) - \boldsymbol{\nu}_k) \\ &= \boldsymbol{\nu}_k + \mathbf{J}^*(\boldsymbol{\nu}_k)(F(\boldsymbol{\nu}_k) - \boldsymbol{\nu}_k) \end{aligned} \tag{1.41}$$

Now let us argue how Eq. (1.41) can be seen as equivalent to our generalized Newton's method in the case of the Real semiring. For brevity we introduce the notation $\mathrm{A}' \overset{\text{def}}{=} Z(\mathrm{A})$ for all $\mathrm{A} \in \mathcal{N}$. We define the function $F : \mathbb{R}^{|\mathcal{N}|} \mapsto \mathbb{R}^{|\mathcal{N}|}$ as follows:

$$F_i \overset{\text{def}}{=} f(\mathrm{A}_\mathrm{i}') = \sum_{\mathrm{A_i} \to \mathrm{B}\ \mathrm{C} \in \mathcal{P}} \mathcal{W}(\mathrm{A_i} \to \mathrm{B}\ \mathrm{C}) \cdot \mathrm{B}' \cdot \mathrm{C}' \tag{1.42}$$

Define $F\Big|_{x=y}$ to be the substitution $F[x \leftarrow y]$. We can then write the treesum of some nonterminal

[19] Refer back to **??** for a discussion related to the last step.

$\mathrm{A}_\mathrm{i}^{=\mathrm{d}}$ in $\mathcal{N}^{<D}$ as

$$Z(\mathrm{A}_\mathrm{i}^{=\mathrm{d}}) = \sum_{\mathrm{X} \in \mathcal{N}} \frac{\delta f_i(\mathrm{A}'_\mathrm{i})}{\delta \mathrm{X}}\Big|_{\mathrm{Y}'=Z(\mathrm{Y}^{<\mathrm{d}})} \cdot Z(\mathrm{X}^{=\mathrm{d}}) \tag{1.43}$$

$$+ \sum_{\substack{(\mathrm{X},\mathrm{Y}) \text{ s.t.} \\ \mathrm{A} \to \mathrm{X}\ \mathrm{Y} \in \mathcal{P}}} \mathcal{W}(\mathrm{A} \to \mathrm{X}\ \mathrm{Y}) \cdot Z(\mathrm{X}^{=\mathrm{d}-1}) \cdot Z(\mathrm{Y}^{=\mathrm{d}-1}), \tag{1.44}$$

where Eq. (1.43) corresponds to the case of one subtree having dimension d and Eq. (1.44) corresponds to the case of incrementing the dimension. Note that if we set $\mathbf{J}'$ to be the Jacobian of F, then

$$\underbrace{\mathbf{J}'_{ij}\Big|_{\mathrm{Y}'=Z(\mathrm{Y}^{<\mathrm{d}})}}_{\stackrel{\text{def}}{=}\mathbf{J}_{ij}} = \frac{\delta f_i(\mathrm{A}'_\mathrm{i})}{\delta \mathrm{A}_\mathrm{j}}\Big|_{\mathrm{Y}'=Z(\mathrm{Y}^{<\mathrm{d}})} \tag{1.45}$$

Additionally, defining $\boldsymbol{\delta}^d \in \mathbb{R}^{|\mathcal{N}|}$, with

$$\boldsymbol{\delta}_i^d \stackrel{\text{def}}{=} \sum_{\substack{(\mathrm{X},\mathrm{Y}) \text{ s.t.} \\ \mathrm{A_i} \to \mathrm{X}\ \mathrm{Y} \in \mathcal{P}}} \mathcal{W}(\mathrm{A_i} \to \mathrm{X}\ \mathrm{Y}) \cdot Z(\mathrm{X}^{=\mathrm{d}-1}) \cdot Z(\mathrm{Y}^{=\mathrm{d}-1}), \tag{1.46}$$

we can write the Newton's method updates for $d \geq 1$ in a succinct vector form as follows

$$\overrightarrow{\boldsymbol{\beta}}^{<d+1} = \overrightarrow{\boldsymbol{\beta}}^{<d} + \overrightarrow{\boldsymbol{\beta}}^{=d} \tag{1.47}$$

$$\overrightarrow{\boldsymbol{\beta}}^{=d} = \mathbf{J}(\overrightarrow{\boldsymbol{\beta}}^{<d}) \cdot \overrightarrow{\boldsymbol{\beta}}^{=d} + \boldsymbol{\delta}^d \tag{1.48}$$

The solution to the above system of linear equations is $\mathbf{J}^*(\overrightarrow{\boldsymbol{\beta}}^{<d}) \cdot \boldsymbol{\delta}^d$, so the update rewrites as

$$\overrightarrow{\boldsymbol{\beta}}^{<d+1} = \overrightarrow{\boldsymbol{\beta}}^{<d} + \mathbf{J}^*(\overrightarrow{\boldsymbol{\beta}}^{<d}) \cdot \boldsymbol{\delta}^d \tag{1.49}$$

This looks a lot like Eq. (1.41). It only remains to argue that $\boldsymbol{\delta}^d$ is equal to $F(\boldsymbol{\nu}_d) - \boldsymbol{\nu}_d$. Since the grammar is in CNF, the only way to get a tree of dimension d by using trees of dimension less than d (which $\overrightarrow{\boldsymbol{\beta}}^{<d}$ restricts us to) is to combine two trees of dimension $d-1$. The values of $F(\overrightarrow{\boldsymbol{\beta}}^{<d})$ will include the weights of dimension d by such increments, while $\overrightarrow{\boldsymbol{\beta}}^{<d}$ does not. Thus, the only values that are accounted for in $F(\overrightarrow{\boldsymbol{\beta}}^{<d})$ but not in $\overrightarrow{\boldsymbol{\beta}}^{<d}$ will be the values in $\boldsymbol{\delta}^d$.

Convergence rate in the counting semiring

There is a rich body of literature around convergence rates of Newton's method which deal with different classes of functions, convergence criteria and variations of the algorithm. Most of these consider the standard Newton's formulation as in Eq. (1.29), and thus only apply to the real semiring. Here, we present one result on the convergence rate for our generalized method in the case of the counting semiring, which can be used to count the number of derivations with a particular string (which can be seen as the string's degree of ambiguity).

We need to distinguish expansive and nonexpansive grammars.

Definition 1.2.3 (Expansive WCFG)**.** *A WCFG $\mathfrak{G}$ is **expansive** iff there exists a derivation on the form $\mathrm{A} \stackrel{*}{\Rightarrow} \boldsymbol{\alpha}_0\ \mathrm{A}\ \boldsymbol{\alpha}_1\ \mathrm{A}\ \boldsymbol{\alpha}_2$ in $\mathfrak{G}$, where $\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in (\mathcal{N} \cup \Sigma)^*$.*

A WCFG is **nonexpansive** iff it is not expansive. We show the following convergence result.

Lemma 1.2.3. *Let $\mathfrak{G}$ be a WCFG over the semiring $\mathcal{W} = (\mathbb{N}, +, \times, 0, 1)$.*

- *If $\mathfrak{G}$ is nonexpansive, Newton's method converges to the correct solution in $n(\mathfrak{G})$ iterations, where $n(\mathfrak{G})$ is the maximum number of distinct nonterminals in a derivation tree in $\mathfrak{G}$. Note that $n(\mathfrak{G}) \leq |\mathcal{N}|$.*
- *If $\mathfrak{G}$ is expansive, Newtons' method does not converge.*

Proof. In the counting semiring, each derivation tree must be visited before the algorithm converges. Consider the case where $\mathfrak{G}$ is expansive. Note that by using derivations like $\mathrm{A} \overset{*}{\Rightarrow} \boldsymbol{\alpha}_0\ \mathrm{A}\ \boldsymbol{\alpha}_1\ \mathrm{A}\ \boldsymbol{\alpha}_2$, we can construct derivation trees of any arbitrary and unbounded dimension. Since Newton's method sums over trees of increasing dimension in each iteration, it will not converge.

Now, consider the case where $\mathfrak{G}$ is nonexpansive. Then for each derivation (sub)tree with root nonterminal A, there is at most one direct subtree that contains A. Thus, there exists a unique path $\boldsymbol{\pi}$ from the root node to a leaf node that visits all nodes that are labeled with the nonterminal A. Let $n(\boldsymbol{t})$ be the number of distinct nonterminals in derivation tree $\boldsymbol{t}$. We use induction on $n(\boldsymbol{t})$ to show that each $\boldsymbol{t}$ must have dimension less than $n(\boldsymbol{t})$.

Base case ($n(\boldsymbol{t}) = 1$) In this case, $\boldsymbol{t}$ can not contain any rule with two or more nonterminals on the right hand side. The dimension is then 0, and $0 < n(\boldsymbol{t})$.

Inductive step ($n(\boldsymbol{t}) > 1$) Let $\boldsymbol{\pi}$ be the path from root node labeled with A to a leaf node that visits all nodes labeled with A. Removing all nodes visited by $\boldsymbol{\pi}$ from $\boldsymbol{t}$ yields a forest with at most $n(\boldsymbol{t}) - 1$ distinct nonterminals. By the inductive hypothesis, each subtree in this forest has dimension less than $n(\boldsymbol{t}) - 1$. Thus, $\boldsymbol{t}$ has dimension less than $n(\boldsymbol{t})$.

It then follows by the fact that Newton's method sums over all derivation trees of dimension d in iteration d, that it converges after $n(\mathfrak{G})$ iterations. □

1.3 Grammar Transforms

1.3.1 Introduction to Grammar Transforms

This section introduces the notion of a grammar transform. At an intuitive level, a grammar transform is a mapping from a grammar $\mathfrak{G}$ to another grammar $\mathfrak{G}'$ such that $\mathfrak{G}$ and $\mathfrak{G}'$ are "equivalent." But, if the grammars $\mathfrak{G}$ and $\mathfrak{G}'$ are equivalent, why would we want to transform? The answer to that lies in notion of what we mean by equivalent, which we intentionally scare-quoted. To understand why grammar transforms might be useful, it's easiest to start with an example. One famous transform is the Chomksy normal form (CNF) transform, which was introduced in the previous section. Recall that in a CNF grammar, all productions are forced to take the form of $\mathrm{A} \to \mathrm{a}$, $\mathrm{A} \to \mathrm{B}\ \mathrm{C}$ (where $\mathrm{A}, \mathrm{B}, \mathrm{C} \in \mathcal{N}$ and $\mathrm{a} \in \Sigma$), or $S \to \varepsilon$. However, the CNF transform maintains a very important invariance—namely, for every string $\boldsymbol{y} \in \Sigma^*$, the string sums remain the same. So, we keep the same string sums and we gain an important computational advantage: We can now determine the string sum in $\mathcal{O}\left(|\boldsymbol{y}|^3\right)$ time using the CKY algorithm that we introduce in the next section. Thus, the trade-off that we see in the CNF transform is that we give up some structure in $\mathfrak{G}$ to gain easier parsing in $\mathfrak{G}'$. We call transforms that preserve the string sums **weak equivalence preserving**. Later in this chapter, we will also encounter more nuanced transforms that enforce a stronger notion of equivalence.

First, let us start by giving a formal definition of a grammar transform.

Definition 1.3.1 (Grammar Transform)**.** *A **grammar transform** $\mathcal{T}$ is a mapping taking as input a CFG $\mathfrak{G}$, possibly together with an additional set of arguments, and returns a CFG $\mathfrak{G}'$, i.e., $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$. The input and output grammars may be either weighted or unweighted.*

Recall that we redefined the notions of weak and strong equivalence specifically for *unweighted* context-free grammars in the previous chapter. We now give two formal definitions of what equivalence can mean in the weighted case.

Definition 1.3.2 (Weak Equivalence)**.** *Two weighted context-free grammars $\mathfrak{G}$ and $\mathfrak{G}'$, defined over semiring $\mathcal{W}$, are **weakly equivalent** if the two following conditions hold:*

- $L(\mathfrak{G}) = L(\mathfrak{G}')$
- $Z_{\mathfrak{G}}(\boldsymbol{y}) = Z_{\mathfrak{G}'}(\boldsymbol{y}), \quad \forall \boldsymbol{y} \in L(\mathfrak{G}) = L(\mathfrak{G}')$

Weak equivalence is a stronger notion in the weighted case in comparison to the unweighted case, as it additionally requires the string sums to be equal. We say that a grammar transform $\mathcal{T}$ is weak equivalence preserving if $\mathfrak{G}$ and $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ are weakly equivalent. Weak equivalence preserving transforms are sensitive to the choice of semiring. This means that a given transform may preserve weak equivalence for some semiring $\mathcal{W}_1$, but not for another one $\mathcal{W}_2$. We give an example below.

Example 1.3.1 (Weak Equivalence Preserving Transforms)**.** *Let $\mathcal{T}$ be a grammar transform that for each string $\boldsymbol{y}$ in the input grammar $\mathfrak{G}$, picks one derivation $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$ where $\mathcal{W}(\boldsymbol{t}) \neq \boldsymbol{0}$ uniformly at random, and only keeps the rules associated with those derivations. Now, consider a WCFG $\mathfrak{G}$ defined by $\mathcal{N} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$, $\Sigma = \{\mathrm{a}\}$, $\mathrm{S} = \mathrm{S}$ and the rules*

$$\mathrm{S} \xrightarrow{w_{sa}} \mathrm{A}$$
$$\mathrm{S} \xrightarrow{w_{sb}} \mathrm{B}$$
$$\mathrm{A} \xrightarrow{w_{aa}} \mathrm{a}$$
$$\mathrm{B} \xrightarrow{w_{ba}} \mathrm{a}$$

with a generic semiring $\mathcal{W}$. Note that the language generated by the grammar consists of a single string, $L(\mathfrak{G}) = \{\mathrm{a}\}$. The string sum of a *is*

$$Z_{\mathfrak{G}}(\mathrm{a}) = w_{sa} \otimes w_{aa} \oplus w_{sb} \otimes w_{ba}$$

Now, feed $\mathfrak{G}$ through the transform $\mathcal{T}$ to retrieve the new grammar $\mathfrak{G}'$. Assume wlog that the derivation $\boldsymbol{t} = (\mathrm{S} \xrightarrow{w_{sa}} \mathrm{A}, \mathrm{A} \xrightarrow{w_{aa}} \mathrm{a})$ was randomly picked. The new grammar will be defined by $\mathcal{N}' = \{\mathrm{S}, \mathrm{A}\}$, $\Sigma = \{\mathrm{a}\}$, $\mathrm{S} = \mathrm{S}$ and the rules

$$\mathrm{S} \xrightarrow{w_{sa}} \mathrm{A}$$
$$\mathrm{A} \xrightarrow{w_{aa}} \mathrm{a}$$

The language is preserved, $L(\mathfrak{G}') = \{\mathrm{a}\}$, and the string sum of a *is*

$$Z_{\mathfrak{G}'}(\mathrm{a}) = w_{sa} \otimes w_{aa}$$

*If we let $\mathcal{W}$ be for instance the Real semiring, it is easy to see that $Z_{\mathfrak{G}}(\mathrm{a}) \neq Z_{\mathfrak{G}'}(\mathrm{a})$ in general, rendering the transform **not** to be weak equivalence preserving.*

In the case of the Boolean semiring however, $\mathfrak{G}$ and $\mathfrak{G}'$ will be weakly equivalent. This can be seen by the following. Since $\boldsymbol{t}$ was picked at random by the transform, we must have that

$\mathcal{W}(\boldsymbol{t}) = w_{sa} \wedge w_{aa} \neq 0$, *which implies that* $\mathcal{W}(\boldsymbol{t}) = 1$ *and* $Z_{\mathfrak{G}'}(\mathrm{a}) = 1$. *For the second derivation of* a *in* $\mathfrak{G}$, $\boldsymbol{t}' = (\mathrm{S} \xrightarrow{w_{sb}} \mathrm{B}, \mathrm{B} \xrightarrow{w_{ba}} \mathrm{a})$, *there are two cases: (i)* $\mathcal{W}(\boldsymbol{t}') = 1$ *and (ii)* $\mathcal{W}(\boldsymbol{t}') = 0$. *In case (i) we will have*

$$Z_{\mathfrak{G}}(\mathrm{a}) = 1 \vee 1 = 1 = Z_{\mathfrak{G}'}(\mathrm{a}),$$

and in case (ii) we will have

$$Z_{\mathfrak{G}}(\mathrm{a}) = 1 \vee 0 = 1 = Z_{\mathfrak{G}'}(\mathrm{a}).$$

Hence, both cases will yield $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ *to be weakly equivalent of* $\mathfrak{G}$.

Note that the definition of weak equivalence makes no mention of derivation sets. In particular, two weakly equivalent grammars may have different number of derivations for the same string. We may want to relate a particular derivation of a string in one grammar to a derivation of the same string in another grammar, which two weakly equivalent grammars would not necessarily allow for. We therefore introduce another stronger property of grammar transforms, which we call **semantics preserving**.

Definition 1.3.3 (Semantics Preserving)**.** *Let* $\mathcal{T}$ *be a grammar transform,* $\mathfrak{G}$ *be a WCFG and* $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ *be the output grammar of transforming* $\mathfrak{G}$ *with* $\mathcal{T}$*, where both* $\mathfrak{G}$ *and* $\mathfrak{G}'$ *are defined over semiring* $\mathcal{W}$. *We say that* $\mathcal{T}$ *is* ***semantics preserving*** *if there exists a bijection* $f : \mathcal{D}_{\mathfrak{G}} \to \mathcal{D}_{\mathfrak{G}'}$ *such that for every* $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}$,

$$\mathcal{W}(\boldsymbol{t}) = \mathcal{W}(f(\boldsymbol{t})), \tag{1.50}$$

and

$$\boldsymbol{y}(\boldsymbol{t}) = \boldsymbol{y}(f(\boldsymbol{t})). \tag{1.51}$$

We want a bijection between the derivation sets so that we do not **undercount** or **overcount** derivations. By undercount we mean that a transform from $\mathfrak{G}$ to $\mathfrak{G}'$ would drop derivations from $\mathcal{D}_{\mathfrak{G}}$. By overcount we mean the opposite, namely that a transform from $\mathfrak{G}$ to $\mathfrak{G}'$ would introduce new derivations in $\mathcal{D}_{\mathfrak{G}'}$ (also called spurious ambiguity). In addition to the bijection, we require the derivations to be of equivalent weight and preserve yield. A transform being semantics preserving implies that it is weak equivalence preserving.

Note that the transform in Example 1.3.1 is not semantics preserving, and cannot be since the derivation sets of the grammars are of different size. As we will see, parts of the CNF transform are semantics preserving. We will also discuss two other semantics preserving transforms, namely the left-corner transform and the speculation transform, later in the chapter.

1.3.2 Chomsky Normal Form

As a reminder of what was mentioned in the previous section, a CNF grammar is a grammar where all rules take the form of one of the following:

- $\mathrm{A} \to \mathrm{B}\ \mathrm{C}$, with $\mathrm{A} \in \mathcal{N}$ and $\mathrm{B}, \mathrm{C} \in \mathcal{N} \setminus \{\mathrm{S}\}$;
- $\mathrm{A} \to \mathrm{a}$, with $\mathrm{A} \in \mathcal{N}$ and $\mathrm{a} \in \Sigma$;
- $\mathrm{S} \to \varepsilon$.

Do note that CNF excludes so-called **nullary rules**, i.e. rules on the form $\mathrm{A} \to \varepsilon$, with one exception: We can still have a nullary rule from the start symbol. This is in order to allow the grammar to generate the empty string. All other rules are either **binary rules** on the form $\mathrm{A} \to \mathrm{B}\ \mathrm{C}$, or **unary terminal productions** on the form $\mathrm{A} \to \mathrm{a}$.

As we will see, all context-free grammars can be expressed in CNF. Derivations stemming from a grammar in CNF will yield a full binary derivation tree (when excluding the edges corresponding to unit productions). This means that if we take a string $\boldsymbol{y}$ with $|\boldsymbol{y}| > 0$ generated from a grammar $\mathfrak{G}$ in CNF, we know that each derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$ will comprise exactly $2|\boldsymbol{y}| - 1$ non-terminal nodes. We also know that each derivation sequence will contain $2|\boldsymbol{y}| - 1$ production rules. This structural feature is what makes CNF particularly useful for parsing.

1.3.3 Weighted Conversion to Chomsky Normal Form

In this section we discuss the conversion of weighted context-free grammars to CNF. A standard recipe for converting CFGs is given in most texts on formal language theory. For instance, see Sipser (1996). This chapter goes beyond these more elementary treatments, however, in that we deal with *weighted* CFGs. The weighted conversion algorithm follows the same series of steps as the unweighted version. Therefore, we will consider each step separately, starting by introducing the unweighted case and then extending it to the weighted case. As we do so, we give a constructive proof showing that the CNF transform is weak equivalence preserving in both the weighted and the unweighted cases. We wrap them up in theorems below, before proceeding with the conversion algorithms.

Theorem 1.3.1 (CNF Unweighted Weak Equivalence)**.** *Let $\mathfrak{G}$ be a CFG and $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ be a transform of $\mathfrak{G}$ to Chomsky normal form. Then, $\mathfrak{G}$ and $\mathfrak{G}'$ are weakly equivalent.*

Theorem 1.3.2 (CNF Weighted Weak Equivalence)**.** *Let $\mathfrak{G}$ be a WCFG and $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ be a transform of $\mathfrak{G}$ to Chomsky normal form. Then, $\mathfrak{G}$ and $\mathfrak{G}'$ are weakly equivalent.*

Throughout this section, we will refer to $\mathfrak{G}$ as the original grammar and $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$ as a transformed grammar given by transform $\mathcal{T}$.

Folding and Unfolding

Before diving into the steps of the conversion, we introduce the **folding** and **unfolding** transforms, the former of which we make use of in the CNF conversion below. Although they can be seen as general concepts applied to sets of rules, we will view them as grammar transforms. We start by discussing the unweighted case, before extending to the weighted case.

Unweighted case. Recall the definition of **applying** a production rule $\mathrm{A} \to \boldsymbol{\beta}$ to $(\boldsymbol{\alpha}\,\mathrm{A}\,\boldsymbol{\gamma}) \in (\mathcal{N} \cup \Sigma)^*$, where we take A and replace it by the right-hand side $\boldsymbol{\beta}$ to yield $\boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma}$. We say that a rule $\mathrm{A} \to \boldsymbol{\beta}$ is **applicable** to A in another rule p, if p contains A on the right-hand side. We make use of this notion as we define unfolding.

Definition 1.3.4 (Unfolding)**.** *Let $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ be a CFG, $p = (\mathrm{B} \to \boldsymbol{\alpha}\,\mathrm{A}\,\boldsymbol{\gamma})$ be a rule in $\mathcal{P}$, $\mathrm{A} \in \mathcal{N}$, and $\mathcal{P}_{\mathrm{A}} = \{\mathrm{A} \to \boldsymbol{\beta}_1, \cdots, \mathrm{A} \to \boldsymbol{\beta}_n\} \subseteq \mathcal{P}$ be the set of rules that are applicable to A in p. Then, the result of **unfolding** p at A in $\mathfrak{G}$ is a grammar $\mathfrak{G}'$, with*

- $\mathcal{N}' = \mathcal{N}$
- $\Sigma' = \Sigma$
- $\mathrm{S}' = \mathrm{S}$
- $\mathcal{P}' = \mathcal{P} \cup \{p'_1, \ldots, p'_n\} \setminus \{p\}$, *where* $p'_i = (\mathrm{B} \to \boldsymbol{\alpha}\,\boldsymbol{\beta}_i\,\boldsymbol{\gamma})$, *for all* $i = 1, \ldots, n$.

More informally, unfolding a rule at a non-terminal gives a grammar with a new set of rules as a result, in which we have applied all applicable rules to A. In a sense it can be seen as merging two levels of a tree. It is easy to see that this grammar transform is weakly equivalent in the unweighted case. Note that the result of *applying* production rules are strings in $(\mathcal{N} \cup \Sigma)^*$, while the process of *unfolding* results in new rules.

As you may have already guessed, folding is the opposite.

Definition 1.3.5 (Folding)**.** *We say that a rule* $\mathrm{A} \to \boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma}$*, with* $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*$*, is* ***foldable*** *by a rule* p *if* p *takes the form* $p = (\mathrm{B} \to \boldsymbol{\beta})$*. Let* $\mathfrak{G}$ *be a CFG,* $p = (\mathrm{A} \to \boldsymbol{\alpha}_1\,\boldsymbol{\beta}_1\,\boldsymbol{\alpha}_2 \cdots \boldsymbol{\alpha}_n\,\boldsymbol{\beta}_n\,\boldsymbol{\alpha}_{n+1})$ *be a rule in* $\mathcal{P}$ *and* $\mathcal{P}_p = \{\mathrm{B}_1 \to \boldsymbol{\beta}_1, \ldots, \mathrm{B}_n \to \boldsymbol{\beta}_n\}$ *be a multiset of foldable rules with left-hand side* $\mathrm{B}_1, \ldots, \mathrm{B}_n \notin \mathcal{N}$ *respectively. The result of* ***folding*** p *by* $\mathcal{P}_p$ *is a grammar* $\mathfrak{G}'$*, with*

- $\mathcal{N}' = \mathcal{N} \cup \{\mathrm{B}_1, \ldots, \mathrm{B}_n\}$
- $\Sigma' = \Sigma$
- $\mathrm{S}' = \mathrm{S}$
- $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}_p \cup \{\mathrm{A} \to \boldsymbol{\alpha}_1\,\mathrm{B}_1\,\boldsymbol{\alpha}_2 \cdots \boldsymbol{\alpha}_n\,\mathrm{B}_n\,\boldsymbol{\alpha}_{n+1}\} \setminus \{p\}$

Note that the members of the set $\mathcal{P}_p$ are not members of $\mathcal{P}$, since $\mathrm{B}_1, \ldots, \mathrm{B}_n \notin \mathcal{N}$. Had a subset of $\{\mathrm{B}_1, \ldots, \mathrm{B}_n\}$ already been in the grammar, we would have added derivations that were previously not present, since the B_i's would have been the head of other production rules. Hence, we would have possibly expanded the language of the grammar. Thus, we note that we expand the grammar not only with one folded rule, but with all rules in $\mathcal{P}_p$.

If unfolding can be seen as merging two levels of a tree, folding can be seen as expanding one level of a tree into two. In both transforms, we replace a rule in the original grammar with a new set of rules.

Weighted case The weighted unfold and fold are actually instances of transforms that are semantics preserving, i.e., stronger than weak equivalence preserving. We prove this property for folding below and give the proof for unfolding as an exercise.

We can ensure that the weighted unfolding process is semantics preserving by setting the weight of the resulting rules to be the product over the two rules stemming from the unfolding. It is easiest to illustrate this process by giving it in pseudocode – see Alg. 4.

Algorithm 4 Unfolding in the weighted case.

```
1.  def WeightedUnfold(𝒲, p = (B → α A γ)):
2.      𝔊' ← 𝔊                                          ▷Initialize grammar
3.      𝒫_A ← {}
4.      for (C → β) ∈ 𝒫 :                               ▷Find all rules with A as head
5.          if C = A :
6.              𝒫_A ← 𝒫_A ∪ {C → β}
7.      for (A → β_i) ∈ 𝒫_A :                           ▷Unfold p at A by all rules in 𝒫_A
8.          𝒫' ← 𝒫' ∪ {B → α β_i γ}
9.          𝒲'(B → α β_i γ) ← 𝒲(B → α A γ) ⊗ 𝒲(A → β_i)
10.     𝒫' ← 𝒫' \ {B → α A γ}                           ▷Remove p from output grammar
11.     return 𝔊'
```

For the folding to be semantics preserving, we can trivially set the weight of the rules in the foldable multiset to be $\mathbf{1}$. We let the rule resulting from the folding have the same weight as the original rule. We give the pseudocode in Alg. 5.

Algorithm 5 Folding in the weighted case.

1. **def** WeightedFold($\mathcal{W}, p = (\mathrm{A} \to \boldsymbol{\alpha}_1 \boldsymbol{\beta}_1 \boldsymbol{\alpha}_2 \cdots \boldsymbol{\alpha}_n \boldsymbol{\beta}_n \boldsymbol{\alpha}_{n+1}), \mathcal{P}_p = \{(\mathrm{B}_1 \to \boldsymbol{\beta}_1), \ldots, (\mathrm{B}_n \to \boldsymbol{\beta}_n)\}$):
2. $\quad \mathfrak{G}' \leftarrow \mathfrak{G}$ ▷*Initialize grammar*
3. $\quad \boldsymbol{\gamma} \leftarrow \varepsilon$ ▷*Initialize substring of result of the folding*
4. $\quad$ **for** $(\mathrm{B}_i \to \boldsymbol{\beta}_i) \in \mathcal{P}$: ▷*Follow left-to-right order in* $\boldsymbol{\beta}$
5. $\quad\quad \boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \circ \boldsymbol{\alpha}_i \circ \mathrm{B}_i$ ▷*Update result of folding*
6. $\quad\quad \mathcal{N}' \leftarrow \mathcal{N}' \cup \{\mathrm{B}_i\}$ ▷*Add new nonterminal*
7. $\quad\quad \mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathrm{B}_i \to \boldsymbol{\beta}_i\}$ ▷*Add foldable rules*
8. $\quad\quad \mathcal{W}'(\mathrm{B}_i \to \boldsymbol{\beta}_i) \leftarrow \mathbf{1}$
9. $\quad \mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathrm{A} \to \boldsymbol{\gamma}\, \boldsymbol{\alpha}_{n+1}\}$ ▷*Add result of folding*
10. $\quad \mathcal{W}'(\mathrm{A} \to \boldsymbol{\gamma}\, \boldsymbol{\alpha}_{n+1}) \leftarrow \mathcal{W}(\mathrm{A} \to \boldsymbol{\gamma}\, \boldsymbol{\alpha}_{n+1})$
11. $\quad \mathcal{P}' \leftarrow \mathcal{P}' \setminus \{\mathrm{A} \to \boldsymbol{\gamma}\, \boldsymbol{\alpha}_{n+1}\}$ ▷*Remove* $\boldsymbol{p}$ *from grammar*
12. $\quad$ **return** $\mathfrak{G}'$

A note on grammar proofs: Next, we give a proof of correctness that the weighted fold is semantics preserving. Before diving into that however, let us give a note on a general proof strategy that we will apply for grammar transforms.

In the general case, we are mapping from an infinite set of derivation trees in the input to another infinite set of derivation trees in the output, and must therefore deploy some suitable proof technique to handle infinite sets. Up until now we have often relied on mathematical induction. Mathematical induction requires us to define an explicit integer induction variable. A natural choice for such a variable that is often made for (derivation) trees would be the height of the tree. However, many, if not most, of the grammar transforms we will discuss do not preserve the height of the tree. So instead, we will make use of a generalization known as **structural induction**.

Structural induction is used to prove a proposition over all elements in some recursively defined structure. As in mathematical induction, base cases will handle the minimal structure, which in the case of grammars will be *trees that have no subtrees*. The inductive step is defined by some recursive rule that dictates how a structure decomposes in terms of its substructures. In the case of trees, a root node decomposes over the subtrees of its children.

For structural induction proofs to be valid we must have a well-founded partial order defined over the structures. We define the partial order over derivation trees based on subtrees.

Definition 1.3.6 (Partial order over derivation trees)**.** $\boldsymbol{t} \prec \boldsymbol{t}'$ *if* $\boldsymbol{t}$ *is a subtree of* $\boldsymbol{t}'$*.*

A binary relation is well-founded if there is no infinite descending chain, i.e., there is no infinite sequence $\boldsymbol{t}_0, \boldsymbol{t}_1, \boldsymbol{t}_2, \ldots$ such that $\boldsymbol{t}_{i+1} \prec \boldsymbol{t}_i$ for all $i \geq 0$. This clearly holds for trees: $\boldsymbol{t} \prec \boldsymbol{t}'$ implies that $\mathsf{height}(\boldsymbol{t}) < \mathsf{height}(\boldsymbol{t}')$, so the sequence must be finite since the height of a tree cannot be less than 0.

With that, let us state and prove the weighted fold theorem.

Theorem 1.3.3. *The WeightedFold transform is semantics preserving.*

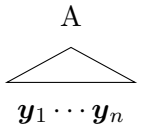
Proof. We must show that there is a bijection between derivations in $\mathfrak{G}$ and $\mathfrak{G}'$, and that the weight and yield for these trees is preserved. A function is bijective if it is both (i) injective and (ii)

surjective. We show injection by showing that for each derivation tree $\boldsymbol{t}$ in $\mathfrak{G}$, there exists a unique derivation tree $\boldsymbol{t}'$ in $\mathfrak{G}'$ such that the weight and yield are preserved. This is done by structural induction over subtrees in $\mathfrak{G}$. Surjection is shown with structural induction over the subtrees in $\mathfrak{G}'$ – we show that for each derivation tree $\boldsymbol{t}'$ in $\mathfrak{G}'$, there exists a derivation tree $\boldsymbol{t}$ in $\mathfrak{G}$ that maps to $\boldsymbol{t}'$ which has the same weight and yield.

We introduce some new notation: $\boldsymbol{t}(\mathrm{A})$ for some symbol A and derivation tree $\boldsymbol{t}$ will denote the subtree of $\boldsymbol{t}$ rooted at A. We will consider the weight and yield of such trees, which we denote by $\mathcal{W}(\boldsymbol{t}(\mathrm{A}))$ and $\boldsymbol{y}(\boldsymbol{t}(\mathrm{A}))$ respectively.

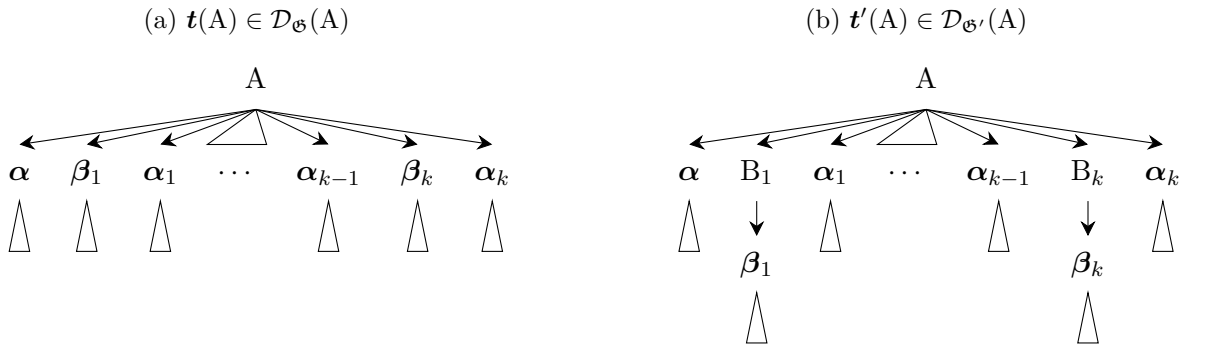
(i) Injection: We start with the case $\mathfrak{G} \Rightarrow \mathfrak{G}'$.

Base case ($p = \mathrm{A} \xrightarrow{w} \boldsymbol{y}$) The base case is a terminal rule $\mathrm{A} \xrightarrow{w} \boldsymbol{y}$, corresponding to a tree of no subtrees. This rule trivially carries over to $\mathfrak{G}'$ as it is not affected by the folding transform, so we have $\mathcal{W}'(\boldsymbol{t}'(\mathrm{A})) = \mathcal{W}(\boldsymbol{t}(\mathrm{A}))$ and $\boldsymbol{y}'(\boldsymbol{t}'(\mathrm{A})) = \boldsymbol{y}(\boldsymbol{t}(\mathrm{A}))$. We note that this is an injective map since all rules in $\mathcal{P}$ are unique.



Inductive step ($p = \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\boldsymbol{\beta}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\boldsymbol{\beta}_k\,\boldsymbol{\alpha}_k$) To avoid case distinction, we consider some arbitrary rule $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\boldsymbol{\beta}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\boldsymbol{\beta}_k\,\boldsymbol{\alpha}_k$ in $\mathcal{P}$, where $\boldsymbol{\alpha}$ is some arbitrary sequence of symbols and k is either 0 or takes some non-zero integer value. The latter case corresponds to the folded rule and the former to any other arbitrary rule in $\mathcal{P}$. Hence note that all rules in $\mathcal{P}$ will take this form.

As our inductive hypothesis we assume that for each subtree $\boldsymbol{t}(\mathrm{C})$ of $\boldsymbol{t}(\mathrm{A})$ in $\mathfrak{G}$, i.e. $\boldsymbol{t}(\mathrm{C}) \prec \boldsymbol{t}(\mathrm{A})$, there is an injective map to a subree $\boldsymbol{t}'(\mathrm{C})$ in $\mathfrak{G}'$ for which weight and yield are preserved. At the inductive step, we must show that the same holds for the subtree rooted at A. We have $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\boldsymbol{\beta}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\boldsymbol{\beta}_k\,\boldsymbol{\alpha}_k$ and foldable rules $\mathrm{B}_1 \to \boldsymbol{\beta}_1, \ldots, \mathrm{B}_k \to \boldsymbol{\beta}_k$. By construction we get $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\mathrm{B}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\mathrm{B}_k\,\boldsymbol{\alpha}_k$ and $\mathrm{B}_1 \xrightarrow{\mathbf{1}} \boldsymbol{\beta}_1, \ldots, \mathrm{B}_k \xrightarrow{\mathbf{1}} \boldsymbol{\beta}_k$ in $\mathcal{P}'$. See the following figure.



It is easy to see that $\mathcal{W}'(\boldsymbol{t}'(\mathrm{A})) = \mathcal{W}(\boldsymbol{t}(\mathrm{A}))$ follows from the inductive hypothesis, since the additional weights of $\mathbf{1}$ do not modify the weight of the subtree rooted at A. We also have that $\boldsymbol{y}'(\boldsymbol{t}'(\mathrm{A})) = \boldsymbol{y}(\boldsymbol{t}(\mathrm{A}))$ since the left-to-right order of the subtrees of $\boldsymbol{t}(\mathrm{A})$ is not permuted in $\boldsymbol{t}'(\mathrm{A})$.

That concludes the first direction. Next, we consider **(ii) Surjection:** $\mathfrak{G}' \Rightarrow \mathfrak{G}$. Here we have one subtle difference: In the inductive step we cannot assume a surjection for all subtrees in $\mathfrak{G}'$, since the subtrees rooted at non-terminals in $\mathfrak{G}'$ that were added by folding will not have a corresponding

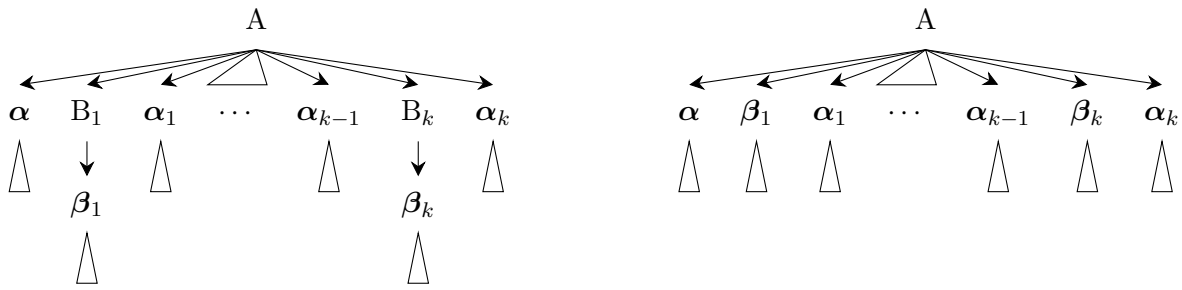
subtree in $\mathfrak{G}$. We denote the set of new non-terminals outputted by the fold as $\mathcal{N}_\mathrm{B}$, and note that $\mathcal{N}' = \mathcal{N} \cup \mathcal{N}_\mathrm{B}$ and $\mathcal{N} \cap \mathcal{N}_\mathrm{B} = \emptyset$ by construction. We exclude subtrees rooted at $\mathrm{B} \in \mathcal{N}_\mathrm{B}$ from the inductive hypothesis and handle these cases separately. Note that it is not necessary for surjection to hold for all subtrees to prove Thm. 1.3.3, since surjection must only hold for derivation trees, i.e. trees rooted at S. It is hence sufficient to show the surjection for all subtrees rooted at A for all $\mathrm{A} \notin \mathcal{N}_\mathrm{B}$, which will cover all A such that $\mathrm{A} \overset{*}{\Rightarrow} \mathrm{B}$.

Base case ($p' = \mathrm{A} \xrightarrow{w} \boldsymbol{y}$) As in the base case for the other direction, the base case is a terminal rule corresponding to a tree of no subtrees, this time in $\mathfrak{G}'$. There are however two subcases: (i) p is not a result of the folding transform, in which case the rule is mapped directly from a rule in $\mathcal{P}$ with the same weight and yield as in $\mathfrak{G}$, or (ii) $p = \mathrm{B} \xrightarrow{\mathbf{1}} \boldsymbol{y}$ for $\mathrm{B} \in \mathcal{N}_\mathrm{B}$. For (ii) there is no surjection.

Inductive step ($p' = \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\mathrm{B}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\mathrm{B}_k\,\boldsymbol{\alpha}_k$) We again consider an arbitrary rule $p' = \mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\mathrm{B}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\mathrm{B}_k\,\boldsymbol{\alpha}_k$ in $\mathcal{P}'$, where $\boldsymbol{\alpha}$ is some arbitrary sequence of symbols and k is either 0 or takes some non-zero integer value, with the latter case corresponding to the folded rule and the former to any other arbitrary rule in $\mathcal{P}'$. All rules in $\mathcal{P}'$ will take this form. Additionally we have $\mathrm{B}_i \xrightarrow{\mathbf{1}} \boldsymbol{\beta}_i$ for all $i \in \{1, 2, \ldots, k\}$.

As our inductive hypothesis we assume that for each subtree $\boldsymbol{t}'(\mathrm{C})$ of $\boldsymbol{t}'(\mathrm{A})$ in $\mathfrak{G}'$ where $\mathrm{C} \notin \mathcal{N}_\mathrm{B}$, there is a surjective map from a subree $\boldsymbol{t}(\mathrm{C})$ in $\mathfrak{G}$ for which weight and yield are preserved. We must show that the same holds for the subtree rooted at A. For $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\mathrm{B}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\mathrm{B}_k\,\boldsymbol{\alpha}_k$ and $\mathrm{B}_1 \xrightarrow{\mathbf{1}} \boldsymbol{\beta}_1, \ldots, \mathrm{B}_k \xrightarrow{\mathbf{1}} \boldsymbol{\beta}_k$ in $\mathcal{P}'$, we must have $\mathrm{A} \xrightarrow{w} \boldsymbol{\alpha}\,\boldsymbol{\beta}_1\,\boldsymbol{\alpha}_1 \cdots \boldsymbol{\alpha}_{k-1}\,\boldsymbol{\beta}_k\,\boldsymbol{\alpha}_k$ in $\mathcal{P}$ by construction.

(a) $\boldsymbol{t}'(\mathrm{A}) \in \mathcal{D}_{\mathfrak{G}'}(\mathrm{A})$ (b) $\boldsymbol{t}(\mathrm{A}) \in \mathcal{D}_{\mathfrak{G}}(\mathrm{A})$



With the inductive hypothesis assumed for subtrees in $\boldsymbol{\alpha}, \boldsymbol{\beta}_1, \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_{k-1}, \boldsymbol{\beta}_k, \boldsymbol{\alpha}_k$, we get that $\mathcal{W}(\boldsymbol{t}(\mathrm{A})) = \mathcal{W}(\boldsymbol{t}(\mathrm{A})) \otimes \mathbf{1}^{\otimes k} = \mathcal{W}'(\boldsymbol{t}'(\mathrm{A}))$ and $\boldsymbol{y}(\boldsymbol{t}(\mathrm{A})) = \boldsymbol{y}(\boldsymbol{t}(\mathrm{A}))$.

Thus, we have shown that the transform is injective and surjective with the desired properties, which implies that it must be semantics preserving.

□

Separate Terminals

Unweighted case The first step of the conversion is to separate all rules that mix non-terminals and terminals, or that mix terminals with other terminals. See the following formal definition.

Definition 1.3.7 (Mixed Rule)**.** *We say that a rule p is mixed if it is on the form*

$$\mathrm{A} \to \boldsymbol{\alpha}\ \mathrm{B}\ \boldsymbol{\beta}\ \mathrm{a}\ \boldsymbol{\gamma}$$

where $\mathrm{B} \in \mathcal{N}$*,* $\mathrm{a} \in \Sigma$ *and* $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathcal{N} \cup \Sigma^*$*.*

We first look at the unweighted case, for a CFG $\mathfrak{G}$. Consider such a **mixed rule** $\mathrm{A} \to \boldsymbol{\alpha}$. We can simply replace all terminals a' in $\boldsymbol{\alpha}$ with new non-terminals A', and for each such instance, add a new unary production rule $\mathrm{A}' \to \mathrm{a}'$. This is the same as folding: We fold $\mathrm{A} \to \boldsymbol{\alpha}$ by a multiset of new rules $\{\mathrm{A}'_1 \to \mathrm{a}'_1, \ldots, \mathrm{A}'_\mathrm{n} \to \mathrm{a}'_\mathrm{n}\}$ for every non-terminal symbol a'_i that participates on the right-hand side $\boldsymbol{\alpha}$ of $\mathrm{A} \to \boldsymbol{\alpha}$.

Iterating this procedure over all productions will result in a grammar with no mixed rules.

Weighted case Now consider a WCFG $\mathfrak{G}$. The weighted version of this step is a very trivial extension – we apply the weighted fold algorithm given in Alg. 5 to every mixed rule in $\mathfrak{G}$. We give the algorithm in pseudocode in Alg. 6. As we showed that the weighted folding is semantics preserving, it is easy to see that so must be the case also for the separating terminals step.

Algorithm 6 Separating terminals in the weighted case.

```
1.  def SeparateTerminals(𝔊 = (𝒩, Σ, S, 𝒫, 𝒲)):
2.      𝔊' ← 𝔊
3.      𝒫' ← {}
4.      for (A → α) ∈ 𝒫 :                          ▷Order does not matter
5.          if (A → α) is mixed :                  ▷We need to separate the terminals
6.              𝒫_fold ← {}                        ▷Initialize foldable rules
7.              for X_i ∈ α :                      ▷Iterate from left-to-right in α
8.                  if X_i ∈ Σ :                   ▷New foldable rule
9.                      𝒩' ← 𝒩' ∪ {X'_i}
10.                     𝒫_fold ← 𝒫_fold ∪ {X'_i → X_i}
11.             𝔊' ← WeightedFold(𝔊', A → α, 𝒫_fold)   ▷Perform weighted fold
12.             ▷ Note that weights for the new rules are returned by weighted fold
13.         else                                   ▷Add rule as is
14.             𝒫' ← 𝒫' ∪ {A → α}
15.             𝒲'(A → α) ← 𝒲(A → α)
16.     return 𝔊'
```

Removing Nullary Rules

Unweighted case Next, we will remove all nullary rules from the grammar. For the unweighted case this should be simple, you might already think, we just delete the nullary rules and we are done. They just generate the empty string and do not make a difference anyway, right? Not quite. We also need to account for the alternatives. Consider a non-terminal A with two unary terminal productions, $\mathrm{A} \to \mathrm{a}$ and $\mathrm{A} \to \varepsilon$. If we simply delete the rule $\mathrm{A} \to \varepsilon$, the grammar will produce an a

for all derivations that previously contained the nullary rule. This might lead to strings getting "lost in translation", in that their derivation trees might have included the empty string that is no more, leading to their absence in the language generated by the transformed grammar $\mathfrak{G}'$—i.e., we would undercount the strings. Hence, we need to add additional rules that exclude the non-terminal that may have derived an ε in the original grammar. This leads us to the notion of **nullable** non-terminals.

Definition 1.3.8 (Nullable non-terminal)**.** *A non-terminal* $\mathrm{A} \in \mathcal{N}$ *is* ***nullable*** *if it derives the empty string, i.e. if* $\mathrm{A} \overset{*}{\Rightarrow} \varepsilon$.

We start the transform algorithm by finding all nullable non-terminals. In order to do so, we transform the grammar $\mathfrak{G}$ into a new grammar $\mathfrak{G}_\varepsilon$ that only generates ε, by removing all rules from $\mathfrak{G}$ that generate a terminal. More specifically, we remove all rules on the form

$$\mathrm{A} \to \boldsymbol{\alpha}\,\mathrm{a}\,\boldsymbol{\beta}, \quad \mathrm{a} \in \Sigma,\, \boldsymbol{\alpha}, \boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^*$$

Note that in $\mathfrak{G}_\varepsilon$, all non-terminals that are co-accessible will correspond to nullable non-terminals in $\mathfrak{G}$. Hence, to find the set of all nullable non-terminals in $\mathfrak{G}$ we find all co-accessible non-terminals in $\mathfrak{G}_\varepsilon$. This is a problem we are familiar with by now – it can be done e.g. by applying fixed-point iteration on the corresponding WCFG that has a boolean semiring, as discussed in the previous chapter.

Next, we follow the intuition presented above and introduce new rules corresponding to cases where nullable rules would have derived ε. More precisely, for each $\mathrm{A} \to \mathrm{A}_1 \ldots \mathrm{A}_\mathrm{n}$, we take every possible *subsequence* $\boldsymbol{\alpha}'$ of $\mathrm{A}_1 \ldots \mathrm{A}_n$ that contain *all* $\mathrm{A}_i \in \{\mathrm{A}_1, \ldots, \mathrm{A}_n\}$ that are *not* nullable and are *not* ε themselves, and add the rule $\mathrm{A} \to \boldsymbol{\alpha}'$.[20] As an example, consider the rule $\mathrm{A} \to \mathrm{B}\,\mathrm{C}\,\varepsilon\,\mathrm{D}$, with B and D being nullable. The set of such valid subsequences would then be $\{\mathrm{C}, \mathrm{B}\,\mathrm{C}, \mathrm{C}\,\mathrm{D}, \mathrm{B}\,\mathrm{C}\,\mathrm{D}\}$, and we would hence add $\mathrm{A} \to \mathrm{C}$, $\mathrm{A} \to \mathrm{B}\,\mathrm{C}$, $\mathrm{A} \to \mathrm{C}\,\mathrm{D}$ and $\mathrm{A} \to \mathrm{B}\,\mathrm{C}\,\mathrm{D}$ to the transformed CFG $\mathfrak{G}'$. Note that for a rule that contains no nullable non-terminals, the only subsequence we will consider is the full sequence itself. In all, this step ensures that $\mathfrak{G}$ and $\mathfrak{G}'$ are weakly equivalent.

Finally, we remove all nullary rules except for $\mathrm{S} \to \varepsilon$. We do not add $\mathrm{S} \to \varepsilon$ to $\mathfrak{G}'$ unless $\varepsilon \in L(\mathfrak{G})$. That is to make sure that ε is still present in the transformed grammar if it was present in the original grammar.

Weighted case In the weighted case we additionally need to consider the weights lost due to the removal of nullary rules. As in the unweighted case, we transform the WCFG $\mathfrak{G}$ into a WCFG $\mathfrak{G}_\varepsilon$ that only generates ε, possessing the same semiring $\mathcal{W}$ as $\mathfrak{G}$. Again, we use $\mathfrak{G}_\varepsilon$ to find all nullable non-terminals but now we additionally require their treesums. More precisely, we require the treesums computed over the ε-generating derivations *only*. Consequently, we just apply our favorite treesum algorithm on $\mathfrak{G}_\varepsilon$. The nullable non-terminals will be the non-terminals with a nonzero treesum as returned by the algorithm.

In the next phase, we follow the same procedure for adding rules that make up for cases where nullable non-terminals would have generated ε. In so doing, we need to additionally substitute in the weight that these nullable non-terminals would have been associated with, i.e., their treesum. More specifically, for a rule $\mathrm{A} \to \boldsymbol{\alpha}'$ with a valid subsequence $\boldsymbol{\alpha}'$ of the right-hand side $\boldsymbol{\alpha} = \mathrm{A}_1 \ldots \mathrm{A}_n$, of some rule $\mathrm{A} \to \mathrm{A}_1 \ldots \mathrm{A}_n$ in the original grammar as explained above; we associate the weight of $\mathrm{A} \to \mathrm{A}_1 \ldots \mathrm{A}_n$, $\otimes$-multiplied with the treesums computed over $\mathfrak{G}_\varepsilon$ of the nullable non-terminals

[20] You might have to read that twice.

that are *not* in $\boldsymbol{\alpha}'$. In other words, we set

$$\mathcal{W}'(\mathrm{A} \to \boldsymbol{\alpha}') = \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) \otimes \bigotimes_{\mathrm{A}_i \in \boldsymbol{\alpha} \setminus \boldsymbol{\alpha}'} Z_{\mathfrak{G}_\varepsilon}(\mathrm{A}_i) \tag{1.52}$$

Intuitively, this ensures that the weights of the ε-generating paths that we removed are still accounted for.

Removing Unary Rules

Unweighted case In the next phase of the conversion, we want to get rid of what we call **unary rules**, which are rules on the form $\mathrm{A} \to \mathrm{B}$, $\mathrm{A}, \mathrm{B} \in \mathcal{N}$. As was the case in the nullary removal phase, we cannot just naively remove all such rules from the grammar. Doing so would erase derivations from the grammar and hence, in general, sacrifice weak equivalence. Instead, we could try to expand all unary rules $\mathrm{A} \to \mathrm{B}$ and replace them with rules going directly from A to the right-hand side of the rules where B participates as the left-hand side. For instance, consider the rules $\mathrm{A} \to \mathrm{B}$ and $\mathrm{B} \to \boldsymbol{\alpha}$. We would then remove $\mathrm{A} \to \mathrm{B}$ and $\mathrm{B} \to \boldsymbol{\alpha}$, and replace them with a rule going directly from A to $\boldsymbol{\alpha}$, $\mathrm{A} \to \boldsymbol{\alpha}$. If there is another set of rules $\mathrm{B} \to \mathrm{C}$ and $\mathrm{C} \to \boldsymbol{\alpha}'$, we would create an additional unary rule $\mathrm{A} \to \mathrm{C}$, before continuing in the same way and replace $\mathrm{A} \to \mathrm{C}$ and $\mathrm{C} \to \boldsymbol{\alpha}'$ with $\mathrm{A} \to \boldsymbol{\alpha}'$.

This strategy breaks, however, in the presence of unary cycles. Consider the same example, with the addition of the rule $\mathrm{C} \to \mathrm{A}$. Our procedure would then add the rule $\mathrm{A} \to \mathrm{A}$, and we are stuck with a unary cyclic rule. Instead, we must consider all pairs (A, B) for which there is a **unary path** from A to B.

Definition 1.3.9. *We say that there is a* ***unary path*** *from* $\mathrm{A} \in \mathcal{N}$ *to* $\mathrm{B} \in \mathcal{N}$ *if there is a sequence of production rules that can be applied, starting from* A *and ending with* B*, consisting exclusively of unary rules.*

Once we have all such pairs (A, B), we can replace the unary sequence of rules from A to B and all rules with a participating B on the left-hand side, $\mathrm{B} \to \boldsymbol{\alpha}$, with rules going directly from A to $\boldsymbol{\alpha}$. Note that this procedure also handles the unary cycle case: If we have (A, A), all unary paths from A to itself will be removed, and we are left with all non-unary rules from A.

We can find all such unary pairs by a simple inductive procedure:

- **Base case:** The pair (A, A) holds for all $\mathrm{A} \in \mathcal{N}$.
- **Inductive step:** If we have found a pair (A, B), and there is a unary rule from B to some non-terminal C, i.e. $\mathrm{B} \to \mathrm{C}$, we add (B, C) to the set of unary pairs.

We leave as an exercise to show that this procedure indeed leads to all, and exactly all, unit pairs being found. With that, we are finished with the unweighted case.

Weighted case For the weighted case, in order for the conversion to be weak equivalence preserving, we need to consider the weight lost from the unary paths that we remove. More specifically, by removal of the unary rules for a unary pair (A, B) , we lose the weight equivalent to the $\oplus$-sum over the weight of all possible sequences of unary production rules we can apply from A to B. Note that this is a notion distinct from anything previously defined in §1.1. In contrast to the treesum, which considers the weight over all production rules in a *derivation* (which as we defined it ends with a string in Σ^*), this notion only includes rules on the "path" from A to B, and exclusively the

unary ones at that. We have not defined the notion of a path in the context of WCFGs, and the algorithms we have discussed for WCFGs do not allow for computation of sums over such paths. The mention of "path" does however remind us of another formalism we have extensively discussed earlier in the course—WFSA.

Indeed, from the WCFG $\mathfrak{G}$ we wish to remove unary rules from, we can create a WFSA in which the pairwise *pathsums* constitute the quantities we would like to compute. We call the resulting WFSA a **unary WFSA**.

Definition 1.3.10 (Unary WFSA)**.** *Let $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a WCFG, where we denote $\mathcal{P}_{unary} \subseteq \mathcal{P}$ as the set of unary rules in $\mathfrak{G}$. Its corresponding **unary WFSA** is $\mathcal{A} = (\Sigma', Q', I', F', \delta', \lambda', \rho')$, with*

- $\Sigma' = \{\varepsilon\}$;
- $Q' = \mathcal{N}$;
- $I' = \mathcal{N}$;
- $F' = \mathcal{N}$;
- $\delta' = \{(\mathrm{A}, \varepsilon, w, \mathrm{B}) \mid \mathrm{A} \xrightarrow{w} \mathrm{B} \in \mathcal{P}_{unary}\}$;
- $\lambda'(q) = \mathbf{1}, \quad \forall q \in Q$;
- $\rho'(q) = \mathbf{1}, \quad \forall q \in Q$;

In words, we consider all non-terminals in $\mathcal{N}$ as states in the WFSA, and add arcs for each pair (A, B) for which there is a unary rule $\mathrm{A} \to \mathrm{B}$. For every such rule we add an arc from A to B in the WFSA, and weight it with the same weight the rule has in $\mathfrak{G}$, i.e., $\mathcal{W}(\mathrm{A} \to \mathrm{B})$. In addition, we consider every state to be both an initial state and a final state, with initial weight and final weight both being set to $\mathbf{1}$. This is in order to be able to consider the notion of a pathsum between any two states. As a reminder, the pathsum between two states $q_1, q_2 \in Q$ is

$$Z(q_1, q_2) = \bigoplus_{\boldsymbol{\pi} \in \Pi(q_1, q_2)} \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \tag{1.53}$$

With the initial and final weights being set to $\mathbf{1}$ for all states, all pathsums in a unary WFSA will be reduced to

$$Z(q_1, q_2) = \begin{cases} \bigoplus_{\boldsymbol{\pi} \in \Pi(q_1, q_2)} w\left(\boldsymbol{\pi}\right), & |\Pi(q_1, q_2)| > 0 \\ \mathbf{1}, & |\Pi(q_1, q_2)| = 0 \end{cases} \tag{1.54}$$

Note that for any non-terminal C that does not participate in a unary rule, we have $\Pi(\mathrm{C}, \mathrm{A}) = \Pi(\mathrm{A}, \mathrm{C}) = \{\}$ for all $\mathrm{A} \in \mathcal{N}$.

Furthermore, since we are not interested in the notion of accepting strings, we label all arcs with ε.

We give the conversion algorithm in Alg. 7. It runs in $\mathcal{O}(|\mathfrak{G}|)$ time. We also give an example of a unary WFSA below.

Example 1.3.2 (A WCFG and its corresponding unary WFSA)**.** *Consider again, as in Example 1.1.5, a WCFG where the treesum represents a geometric series. This grammar, will be slightly*

Algorithm 7 Make a unary WFSA from a WCFG.

def MakeUnaryWFSA($\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$):
  ▷ *Initialize the WFSA*
  $\mathcal{A} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)$
  $Q \leftarrow \mathcal{N}$
  $I \leftarrow \mathcal{N}$
  $\lambda \leftarrow \mathbf{1}$
  $F \leftarrow \mathcal{N}$
  $\rho \leftarrow \mathbf{1}$
  for $(\mathrm{A} \to \mathrm{B}) \in \mathcal{P}$: ▷*Iterate over all unary rules*
    $Q \leftarrow Q \cup \{\mathrm{A}, \mathrm{B}\}$
    $\delta \leftarrow \delta \cup \{(\mathrm{A}, \varepsilon, \mathcal{W}(\mathrm{A} \to \mathrm{B}), \mathrm{B})\}$
  return $\mathcal{A}$

different however – we will make exclusive use of unary rules. We let $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$*, where* $\mathcal{N} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$*,* $\Sigma = \{\varepsilon\}$*,* $\mathrm{S} = \mathrm{S}$*,* $\mathcal{W}$ *is the Rational semiring and* $\mathcal{P}$ *includes:*

$$\mathrm{S} \xrightarrow{1} \mathrm{A}$$
$$\mathrm{A} \xrightarrow{1/3} \mathrm{A}$$
$$\mathrm{A} \xrightarrow{1/2} \mathrm{B}$$
$$\mathrm{B} \xrightarrow{1} \varepsilon$$

Note that the language generated by $\mathfrak{G}$ *is the empty string alone, and that the treesum is*

$$\begin{aligned} Z_{\mathfrak{G}} &= 1 \times \frac{1}{2} \times \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i \times 1 \\ &= \frac{3}{4} \end{aligned}$$

Due to the rules $S \to \mathrm{A}$ *and* $\mathrm{B} \to \varepsilon$ *having weight* 1*, the "sum over all unary paths" from* A *to* B *will be the same as the treesum. Now, we transform* $\mathfrak{G}$ *into a unary WFSA with Alg. 7, and arrive at* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *with* $\Sigma = \{\varepsilon\}$*,* $Q = I = F = \{\mathrm{S}, \mathrm{A}, \mathrm{B}\}$*,* $\lambda(\cdot) = \rho(\cdot) = 1$ *and the transitions given by Fig. 1.7.*

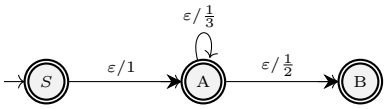


Figure 1.7: Unary WFSA corresponding to geometric series WCFG.

It is easy to see that the pathsum between A *and* B *will be the same as the treesum in* $\mathfrak{G}$*:*

$$\begin{aligned} Z(q_1, q_2) &= \frac{1}{2} \times \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i \\ &= \frac{3}{4} \end{aligned}$$

As per usual, we can compute the pathsums in the unary WFSA with any algorithm that solves the algebraic path problem as discussed in **??**. In the general case we can resort to Lehmann's which will exactly compute the pathsums for any closed semiring, given that its closure exists. Note also that the unary pairs will correspond to all non-zero pathsums in the unary WFSA.

With all unary pairs found and the pathsums computed, we follow the same procedure as in the unweighted case. For each unary pair (A, B), and non-unary production rule $\mathrm{B} \xrightarrow{w} \boldsymbol{\alpha}$, we add the rule

$$\mathrm{A} \to \boldsymbol{\alpha}$$

with

$$\mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha}) = w \otimes Z(q_A, q_B)$$

to the output grammar $\mathfrak{G}'$. Do note that all non-unary rules will be added as they are by default, since we have that $Z(q_A, q_A) = \mathbf{1}$, unless there is a unary path from A to itself, in which case the pathsum will be included in the weight in the transformed grammar.

Binarizing

Unweighted case What remains is to **binarize** all those remaining rules that are not on the unary terminal form. By binarizing we mean transforming all rules into binary rules on the form $\mathrm{A} \to \mathrm{B}\,\mathrm{C}$, i.e. where the right-hand side is of length 2. Note that rules with only one participating non-terminal on the right-hand side, i.e. unary rules, will have been removed in a previous step. Note also that the case in which a rule has multiple participating terminal symbols, without any participating non-terminals, are taken care of by SeparateTerminals, by the way we defined mixed rules. Hence, after the previous steps, all remaining rules that do not comply with CNF will take the form

$$\mathrm{A} \to \mathrm{A}_1 \cdots \mathrm{A}_n, \tag{1.55}$$

where $n > 2$.

To handle these rules, we break them up into $n - 1$ new rules that are all binary. This is done by introducing $n - 2$ new non-terminals $\mathrm{B}_1, \ldots, \mathrm{B}_{n-2}$, each as the head of a new binary production rule. This process can be done by folding in a step-wise manner, i.e., by folding $\mathrm{A} \to \mathrm{A}_1 \cdots \mathrm{A}_n$ with one rule at a time until the result of the folding process is binarized. More specifically, we start by folding $\mathrm{A} \to \mathrm{A}_1 \cdots \mathrm{A}_n$ by $\mathrm{B}_1 \to \mathrm{A}_1\,\mathrm{A}_2$, in order to obtain $\mathrm{A} \to \mathrm{B}_1\,\mathrm{A}_3 \cdots \mathrm{A}_n$. Next, we take our newly obtained rule $\mathrm{A} \to \mathrm{B}_1\,\mathrm{A}_3 \cdots \mathrm{A}_n$ and fold it by $\mathrm{B}_2 \to \mathrm{B}_1\,\mathrm{A}_3$, and obtain $\mathrm{A} \to \mathrm{B}_2\,\mathrm{A}_4 \cdots \mathrm{A}_n$. We terminate the process when the resulting rule is binary. This process is applied to all rules on the form of Eq. (1.55).

Weighed case Again, the weighted case is a trivial extension since we have already discussed weighted folding. All new binary rules by which we fold will be given weight $\mathbf{1}$, while the final result of the folding process will preserve the weight of the original rule. We give the pseudocode below.

Trimming

Unweighted case As a last step, we may trim the CFG. Although not strictly necessary for a CFG to be in CNF, eliminating useless non-terminals will clearly not affect the language generated by the grammar, and they can hence safely be removed.

As we have discussed how we can find all accessible and co-accessible symbols in §1.1, we will assume access to such. All that remains then is to eliminate all rules for which we have participating useless symbols. We give the algorithm below.

Algorithm 8 Binarizing in the weighted case.

def Binarize($\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$):
  ▷ *Initialize output grammar*
  $\mathfrak{G}' \leftarrow \mathfrak{G}$
  $\mathcal{P}' \leftarrow \{\}$
  stack $\leftarrow \mathcal{P}$ ▷*Add all rules to stack*
  while |stack| > 0 :
    $(\mathrm{A} \to \boldsymbol{\alpha}) \leftarrow$ stack.pop()
    if $(\mathrm{A} \to \boldsymbol{\alpha})$ **complies with** CNF : ▷*Add rule as is*
      $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathrm{A} \to \boldsymbol{\alpha}\}$
      $\mathcal{W}'(\mathrm{A} \to \boldsymbol{\alpha}) \leftarrow \mathcal{W}(\mathrm{A} \to \boldsymbol{\alpha})$
    else ▷*Need to binarize*
      ▷ *We have* $\boldsymbol{\alpha} = \mathrm{A}_1 \cdots \mathrm{A}_n$
      $\mathrm{B} \leftarrow$ generateNonterminal()
      push WeightedFold($\mathcal{W}, \mathrm{A} \to \mathrm{A}_1 \cdots \mathrm{A}_n, \{\mathrm{B} \to \mathrm{A}_1\,\mathrm{A}_2\}$) to stack
      ▷ *Note that we push both the result of the folding and* $\mathrm{B} \to \mathrm{A}_1\,\mathrm{A}_2$ *to the stack*
  return $\mathfrak{G}'$

Algorithm 9 Trim a CFG.

def TrimCFG($\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$):
  $\mathfrak{G}' \leftarrow \mathfrak{G}$
  $\mathcal{P}' \leftarrow \{\}$ ▷*Initialize grammar without rules*
  $\mathcal{A} \leftarrow$ accessible($\mathfrak{G}$)
  $\mathcal{C} \leftarrow$ co-accessible($\mathfrak{G}$)
  $\mathcal{U} \leftarrow \mathcal{N} \setminus \mathcal{A} \setminus \mathcal{C}$ ▷*Set of useless non-terminals*
  for $(\mathrm{A} \to \boldsymbol{\alpha}) \in \mathcal{P}$:
    if no participant of $\boldsymbol{\alpha}$ **in** $\mathcal{U}$: ▷*Add rule if it does not contain any useless non-terminals*
      $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathrm{A} \to \boldsymbol{\alpha}\}$
  return $\mathfrak{G}'$

There is an additional caveat however – we need to run trim twice in order to eliminate all useless non-terminals. The following example, taken from Hopcroft et al. (2006), will illustrate why. Two times is always enough to guarantee a grammar without useless non-terminals. We leave it as an exercise to show that so is the case.

Example 1.3.3 (Trim a CFG)**.** *Consider the CFG* $\mathfrak{G}$ *with* $\mathcal{N} = \{S, \mathrm{A}, \mathrm{B}\}$, $\Sigma = \{a, b\}$, $\mathrm{S} = S$ *and the rules*

- $S \to \mathrm{A}\,\mathrm{B}$*;*
- $S \to a$*;*
- $\mathrm{A} \to b$*;*

We see that the set of accessible symbols is $\{\mathrm{S}, \mathrm{A}, \mathrm{B}, a\}$*, the set of co-accessible non-terminals is* $\{\mathrm{S}, \mathrm{A}\}$ *and the language generated by* $\mathfrak{G}$ *is* $L(\mathfrak{G}) = \{a\}$*. Note that* B *is a useless non-terminal. Trimming* $\mathfrak{G}$ *as outlined above will then remove all rules in which* B *participates, which yields a reduced set of rules*

- $S \to a$;
- $\mathrm{A} \to b$;

In this new grammar, however, we note that A *is no longer accessible, which means that we have not yet arrived at a grammar that is trimmed. We again eliminate all rules in which we have a participating useless non-terminal, and arrive at*

- $S \to a$;

This new grammar, $\mathfrak{G}'$, has no useless non-terminals and is trimmed. It preserves weak equivalence as $L(\mathfrak{G}') = L(\mathfrak{G}) = \{a\}$.

Weighted case The weighted case requires no further extension. As a reminder, the string sum of a string $\boldsymbol{y}$ is the sum over the weights of all derivations starting from the start symbol S:

$$Z_{\mathfrak{G}}(\boldsymbol{y}) = \bigoplus_{\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})} \mathcal{W}(\boldsymbol{t})$$

It is easy to see that useless symbols will not affect the string sum. Symbols that are not accessible will not participate in any rules for which the weight is included in the string sum, since they can not, by definition, be derived from S. Non-terminals that are not co-accessible will not either, since they do not derive any terminal. A string $\boldsymbol{y} \in L(\mathfrak{G})$ can hence not include any rule with a participating non-terminal that is not co-accessible.

1.3.4 Left-Corner Transform

In the previous section, we focused on the derivation of the CNF transform. The CNF transform is best thought of as a meta transform as it comprises a series of individual transforms. The next two transforms we will study, the **left-corner transform** and **speculation**, are more self-contained. Interestingly, they are also instances of transforms that are semantics preserving. This section introduces the left-corner transform. Note that although the discussion below will consider the left-corner transform, a right-corner transform can be defined analogously.

In order to be able to talk about the utility and intuition of the left-corner transform, we must introduce the notion of a **left-corner**. We start by defining the **direct left-corner** relation.

Definition 1.3.11 (Direct Left-Corner)**.** *A symbol* $\mathrm{X} \in (\mathcal{N} \cup \Sigma)$ *is a* ***direct left-corner*** *of a non-terminal* $\mathrm{A} \in \mathcal{N}$ *if there exists a* $p \in \mathcal{P}$ *such that* p *takes the form*

$$\mathrm{A} \to \mathrm{X}\,\boldsymbol{\alpha}, \quad \boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$$

The above definition just means that there exists a rule where the symbol X takes the left-most position on the right-hand side in a rule from A. We then say that a symbol X is the **left-corner** of a non-terminal A if there is a sequence of rules that starts from A and derives some X, where each rule is applied on the direct left-corner. More formally we have the following definition.

Definition 1.3.12 (Left-Corner)**.** *The* ***left-corner*** *relation is the reflexive and transitive closure of the direct left-corner relation.*

The standard left-corner transform eliminates what is known as **left-recursion** from the grammar.[21] See the definition below.

Definition 1.3.13 (Left-Recursion)**.** *A non-terminal* A *is* ***left-recursive*** *if it is a left-corner of itself. Furthermore, we say that* A *is* ***directly left-recursive*** *if it is a direct left-corner of itself, and* ***indirectly left-recursive*** *if it is left-recursive but not directly left-recursive.*

We will also consider **left-recursive rules** in the discussion below. A left-recursive rule is one that starts a sequence of rules that lead to left-recursion.

Definition 1.3.14 (Left-Recursive Rule)**.** *A rule* $p = (\mathrm{A} \to \mathrm{B}\,\boldsymbol{\alpha})$*, with* $\mathrm{A}, \mathrm{B} \in \mathcal{N}$ *and* $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$*, is* ***left-recursive*** *if* A *is a left-corner of* B*.*

Example 1.3.4 (Top-down parsing with left-recursion)**.** *Consider a CFG* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ *that generates* $L = \{a(ba)^n \mid n \geq 0\}$*, defined as follows:*

- $\mathcal{N} = \{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{C}\}$
- $\Sigma = \{a, b\}$
- $\mathrm{S} = \mathrm{S}$
- $\mathcal{P} = \{\mathrm{S} \to \mathrm{C}\ \mathrm{A}, \mathrm{C} \to \mathrm{S}\ \mathrm{B}, \mathrm{C} \to \varepsilon, \mathrm{A} \to a, \mathrm{B} \to b\}$

We see that C *is a direct left-corner of* S*, and that* S *is a direct left-corner of* C*. Hence,* $\mathrm{S} \overset{*}{\Rightarrow} \mathrm{C}$ *and* $\mathrm{C} \overset{*}{\Rightarrow} \mathrm{S}$ *with direct left-corner relations only, which implies that* S *is a left-corner of* S *and* A *is a left-corner of* A*. It follows, by definition, that* S *and* C *are left-recursive. Both* $\mathrm{S} \to \mathrm{C}\ \mathrm{A}$ *and* $\mathrm{C} \to \mathrm{S}\ \mathrm{B}$ *are left-recursive rules.*

Now let us see how this poses a problem for top-down parsing. Consider a recursive descent parser, which is a simple top-down left-to-right parsing algorithm. It proceeds by building a left-most derivation of the input string by applying production rules, one rule at a time, starting at the root. Say we have a string aba *and we want to know whether this string is in* L*. This is a parsing recognition problem (we will generalize this problem to arbitrary semirings and discuss more nuanced parsing algorithms later in this chapter). The recursive descent parser first applies the (only possible) root rule* $\mathrm{S} \to \mathrm{C}\ \mathrm{A}$*. In accordance with left-most derivations, it then applies a production rule with head* C*. There are two such rules and suppose it picks the correct one for this step, namely* $\mathrm{C} \to \mathrm{S}\ \mathrm{B}$*. It then applies* $\mathrm{S} \to \mathrm{C}\ \mathrm{A}$ *again. We are now at the derivation* $\mathrm{S} \Rightarrow \mathrm{C}\ \mathrm{A} \Rightarrow \mathrm{S}\ \mathrm{B}\ \mathrm{A} \Rightarrow \mathrm{C}\ \mathrm{A}\ \mathrm{B}\ \mathrm{A}$*. This is the correct derivation for* aba *given that the next production rule applied is* $\mathrm{C} \to \varepsilon$*. But if the parser instead prioritizes the production rule* $\mathrm{C} \to \mathrm{S}\ \mathrm{B}$*, it will end up in an infinite recursive loop.*

Now consider another grammar that generates the same language, where the production rules are the following:

- $\mathcal{P} = \{\mathrm{S} \to \mathrm{A}\ \mathrm{C}, \mathrm{C} \to \mathrm{B}\ \mathrm{S}, \mathrm{C} \to \varepsilon, \mathrm{A} \to a, \mathrm{B} \to b\}$

We have now replaced left-recursion by right-recursion. In this grammar, the parser will be able to derive the terminal symbols on the left-corners without the issue of infinite recursion. Having derived the string aba*, it can then use the trailing* C *to derive an* ε*.*

[21] Note that the definition of left-recursion encapsulates cases of unary cycles. Unary cycle removal is not semantics preserving and we must therefore assume that the input grammar has no unary cycles for the left-corner transform to remove all left-recursion. These are an exception that cannot be removed from the grammar for the transform to be semantics preserving.

The original motivation behind the left-corner transform was to allow for simple top-down parsing, since left-recursion may lead to infinite recursion as illustrated in the example above. In addition it has also seen applications as cognitively plausible parsers, since left-corner parsers have been argued to more closely resemble how humans process sentences (in comparison to top-down or bottom-up parsers). See e.g. Resnik (1992) and Roark and Johnson (1999) if you are interested in learning more about the left-corner transform in the context of psycholinguistics.

The left-corner transform (Rosenkrantz and Lewis, 1970), in a sentence, lifts up the left-corner to the top of the tree, bringing information from the bottom of the tree up to the top. Here, we present a generalized transform that can give a smaller output grammar than the original left-corner transform as introduced by Rosenkrantz and Lewis (1970). Through additional input parameters to the transform it gives finer-grained control over the trees in the output grammar than the original transform, thus possibly resulting in smaller grammars (depending on the choice of those parameters).

There is a simple set of rules that map a grammar to its left-corner transform. The transform creates two new sets of non-terminals: $Y\backslash X$ ("slashed non-terminals", where $\mathrm{X} \in (\mathcal{N} \cup \Sigma)$ and $\mathrm{Y} \in \mathcal{N}$) and $\sim X$ ("marked non-terminals", where $\mathrm{X} \in \mathcal{N}$). Slashed non-terminals derive the difference between Y and X in the original grammar. In other words, this non-terminal would be the root of the subtree in the original grammar derived by Y, but excluding the subtree rooted at the left-corner X. A marked terminal $\sim X$ indicates that X is not slashed anymore within the subtree rooted at ~X. Further, these non-terminals mark where a subtree has been hoisted. These two types of non-terminals will hopefully become more clear as we talk about examples below.

In addition to the WCFG $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$, we take as input a set of non-terminals $\mathcal{N}_\backslash \subseteq \mathcal{N}$ and a set of rules $\mathcal{P}_\backslash \subseteq \mathcal{P}$ where $\mathcal{P}_\backslash$ may not include any nullary rules. We have no further constraints on $\mathcal{N}_\backslash$ and $\mathcal{P}_\backslash$ in order for the transform to be semantics preserving. There are, however, constraints if we wish for $\mathcal{T}(\mathfrak{G})$ to eliminate left-recursion.

The rules are the following:

Transform 1.3.1 (Left-Corner Transform)**.** *The* ***generalized left-corner transform*** $\mathcal{T}_{LC}(\mathfrak{G}, \mathcal{P}_\backslash, \overline{\mathcal{N}}, \underline{\mathcal{N}})$*, for* $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P}, \mathcal{W})$ *returns a WCFG* $\mathfrak{G}'$ *defined by the following rules:*

$$\mathrm{X}\backslash\mathrm{X} \xrightarrow{\mathbf{1}} \varepsilon, \qquad \mathrm{X} \in \mathcal{N} \tag{1.56}$$

$$\sim\mathrm{X} \xrightarrow{w} \boldsymbol{\alpha}, \qquad \mathrm{X} \xrightarrow{w} \boldsymbol{\alpha} \notin \mathcal{P}_\backslash \text{ or } \mathrm{X} \notin \underline{\mathcal{N}} \tag{1.57}$$

$$\mathrm{Y}\backslash\mathrm{X}_1 \xrightarrow{w} \boldsymbol{\alpha}\, \mathrm{Y}\backslash\mathrm{X}, \qquad \mathrm{X} \xrightarrow{w} \mathrm{X}_1\, \boldsymbol{\alpha} \in \mathcal{P}_\backslash, \ \mathrm{X} \in \underline{\mathcal{N}}, \mathrm{Y} \in \underline{\mathcal{N}} \tag{1.58}$$

$$\sim\mathrm{X} \xrightarrow{w} \sim\mathrm{X}_1\, \boldsymbol{\alpha}, \qquad \mathrm{X} \xrightarrow{w} \mathrm{X}_1\, \boldsymbol{\alpha} \in \mathcal{P}_\backslash, \ \mathrm{X} \in \underline{\mathcal{N}}, \mathrm{X}_1 \in \mathcal{N} \setminus \overline{\mathcal{N}}, \tag{1.59}$$

$$\mathrm{X} \xrightarrow{\mathbf{1}} \sim\mathrm{X}, \qquad \mathrm{X} \in \mathcal{N} \setminus \overline{\mathcal{N}} \tag{1.60}$$

$$\mathrm{X} \xrightarrow{\mathbf{1}} \sim\mathrm{Z}\ \mathrm{X}\backslash\mathrm{Z}, \qquad \mathrm{X} \in \mathcal{N}, \mathrm{Z} \in \overline{\mathcal{N}} \tag{1.61}$$

$$\mathrm{X} \xrightarrow{\mathbf{1}} a\ \mathrm{X}\backslash a, \qquad \mathrm{X} \in \mathcal{N}, \mathrm{a} \in \Sigma \tag{1.62}$$

Left-corner recognition is performed by Eq. (1.58) – this rule is through which we bring information of the left-corner up the tree. We slash all non-terminals $\mathrm{Y} \in \underline{\mathcal{N}}$ on X_1, which can be either a terminal or a non-terminal symbol. The rules Eq. (1.60), Eq. (1.61) and Eq. (1.62) are called **recovery rules**. It is through these rules that we recover the yield of derivations of the

original grammar. Further note that all non nullary terminal rules stem from Eq. (1.57), due to the constraint that $\mathcal{P}_\backslash$ can not include any rules that produce terminal symbols.

We recover a transform similar to the original left-corner transform by setting $\mathcal{N}_\backslash = \mathcal{N}$ and $\mathcal{P}_\backslash$ to be all rules in $\mathcal{P}$ that are not nullary rules. In the original left-corner we have that if $\mathrm{Y} \overset{*}{\Rightarrow} \mathrm{X}\,\boldsymbol{\alpha}$ in $\mathfrak{G}$, then $\mathrm{Y}\backslash\mathrm{X} \overset{*}{\Rightarrow} \boldsymbol{\alpha}$ in the transformed grammar $\mathfrak{G}'$. Informally this means that the new non-terminal $\mathrm{Y}\backslash\mathrm{X}$ in $\mathfrak{G}'$ derives the *difference* between Y and X, namely $\boldsymbol{\alpha}$, in the original grammar.

The original transform, however, introduces a large number of useless non-terminals (Johnson and Roark, 2000; Moore, 2000), which blows up the run time for parsing algorithms. The selective left-corner transform introduced in Johnson and Roark (2000) remedies this issue by only slashing on a subset of rules. This can be seen as another special case of the transform presented here, recovered in approximation by setting $\mathcal{N}_\backslash = \mathcal{N}$ and $\mathcal{P}_\backslash$ to be all left-recursive rules in $\mathcal{P}$. Doing so maintains the properties of eliminating left-recursion and that $\mathrm{Y}\backslash\mathrm{X}$ derives the difference between Y and X in the original grammar.

A novel introduction of this transform in contrast to that of Johnson and Roark (2000) however, is the addition of the negation rules. The negation rules allow us to "hoist" up subtrees of derivation trees in the original grammar. More specifically, a subtree in the original grammar can be lifted to be a child of a marked non-terminal, which in turn will be a child of the start symbol, in the output grammar. Previous left-corner transforms only hoisted minimal subtrees, i.e. trees of height 1 with a terminal leaf node.

Understanding the left-corner transform can be a bit tricky, so it is best learned through studying various examples.

Example 1.3.5 (Eliminating left-recursion)**.** *Consider the CFG from Example 1.3.4. We apply the left-corner transform with* $\mathcal{N}_\backslash = \{\mathrm{S}, \mathrm{X}\}$ *and* $\mathcal{P}_\backslash = \{(\mathrm{S} \to \mathrm{X}\ a), (\mathrm{X} \to \mathrm{S}\ b)\}$ *and get as output the CFG* $\mathfrak{G}'$*, with:*

- $\mathcal{N}' = \{\mathrm{S}, \mathrm{S}\backslash\mathcal{N}, {\sim}X, {\sim}\mathrm{S}, \mathrm{X}, \mathrm{S}\backslash a, \mathrm{S}\backslash\mathrm{S}, X\backslash a, S\backslash b, X\backslash b, X\backslash X, X\backslash S\}$*;*
- $\Sigma' = \{a, b\}$*;*
- $\mathrm{S}' = \mathrm{S}$*;*

and a large set of rules $\mathcal{P}'$*, most of which are useless. If we trim* $\mathfrak{G}'$*, we get the following rules:*

- $S \to {\sim}X\ S\backslash X$
- ${\sim}X \to \varepsilon$
- $S\backslash X \to a\ S\backslash S$
- $S\backslash S \to \varepsilon$
- $S\backslash S \to b\ S\backslash X$

Now consider the string $\boldsymbol{y} = aba$*. The derivation for this string in* $\mathfrak{G}$ *contains left-recursion, as we can see on the left-hand side in Fig. 1.8. On the right-hand side we see the corresponding derivation tree in* $\mathfrak{G}'$*, in which left-recursion is removed and replaced by right-recursion. In fact, this is a general trait of the construction – left-recursion in the original grammar will be replaced by right-recursion in the transformed grammar.*

Do also note the bijection between the derivations in $\mathfrak{G}$ *and* $\mathfrak{G}'$*. Both CFGs are unambiguous – the left-hand side derivation tree in Fig. 1.8 is the unique derivation for aba in* $\mathfrak{G}$ *while the right-hand side derivation tree in Fig. 1.8 is the unique derivation for aba in* $\mathfrak{G}'$*.*

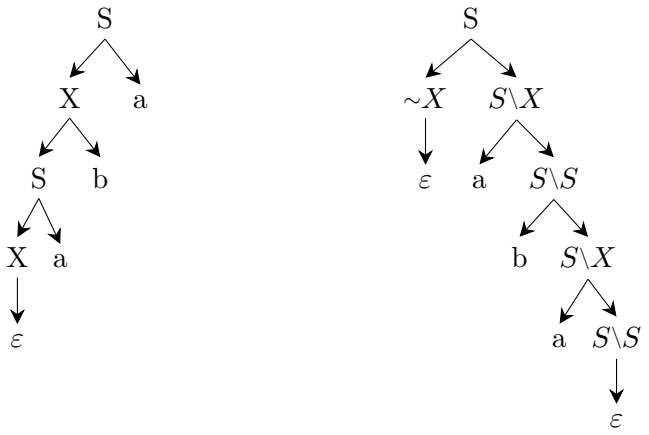


Figure 1.8: **Left**: Derivation tree for *aba* in $\mathfrak{G}$. **Right**: Derivation tree for *aba* in $\mathfrak{G}'$.

Note that we must include both $\mathrm{S} \to \mathrm{X}\ a$ *and* $\mathrm{X} \to \mathrm{S}\ b$ *in* $\mathcal{P}_\setminus$ *in order to eliminate left-recursion, as both of these rules are left-recursive. However, we can make use of the additional degree of freedom in* $\mathcal{N}_\setminus$*. Take for instance* $\mathcal{N}_\setminus = \{\mathrm{S}\}$*. The resulting tree for aba will be identical to the tree on the right hand side of Fig. 1.8. As an exercise, we leave you to think about the case* $\mathcal{N}_\setminus = \{\mathrm{X}\}$*. Will that eliminate left-recursion?*

Example 1.3.6 (Left-corner transform and semantics preserving)**.** *This example is adapted from Johnson (1998). Consider a WCFG* $\mathfrak{G}$ *that generates a single string,* $L(\mathfrak{G}) = \{$*the dog ran fast*$\}$*, with the rational semiring* $\mathcal{W}$*. It is defined by the following rules:*

- $\mathrm{S} \xrightarrow{1} NP\ VP$
- $NP \xrightarrow{1/2} Det\ N$
- $Det \xrightarrow{1/2}$ *the*
- $N \xrightarrow{1/2}$ *dog*
- $VP \xrightarrow{1} V\ ADV$
- $V \xrightarrow{1}$ *ran*
- $ADV \xrightarrow{1/2}$ *fast*

The derivation tree corresponding to $\boldsymbol{y} =$*"the dog ran fast" is illustrated in Fig. 1.9. It is easy to see that the weight of the derivation tree* $\boldsymbol{t}$ *is*

$$\mathcal{W}(\boldsymbol{t}) = 1 \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \times 1 \times 1 \times \frac{1}{2} = \frac{1}{16} \tag{1.63}$$

Next, we set $\mathcal{N}_\setminus = \{\mathrm{S}, \mathrm{NP}, \mathrm{VP}, \mathrm{Det}, \mathrm{V}\}$ *and* $\mathcal{P}_\setminus = \{(\mathrm{S} \to \mathrm{NP\ VP}), (\mathrm{NP} \to \mathrm{Det\ N}), (\mathrm{VP} \to \mathrm{V\ ADV})\}$ *and apply the left-corner transform to get* $\mathfrak{G}' = \mathcal{T}(\mathfrak{G})$*. We get the following non-useless rules:*

- $\mathrm{S} \xrightarrow{1} \sim\mathrm{Det\ S}\backslash\mathrm{Det}$
- $\sim\mathrm{Det} \xrightarrow{1/2}$ *the*

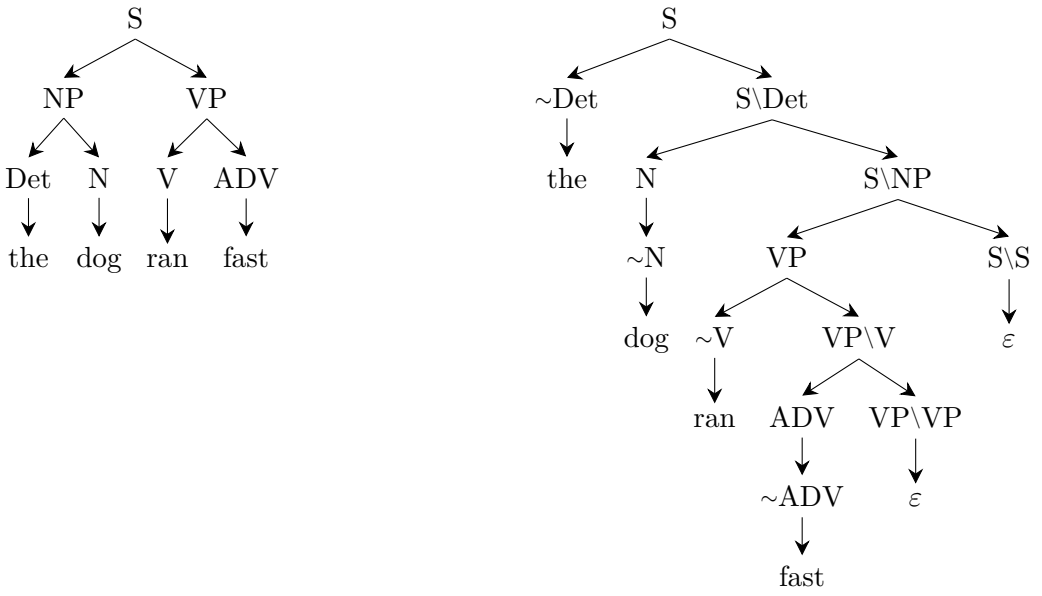


Figure 1.9: **Left**: Derivation tree for *aba* in $\mathfrak{G}$. **Right**: Derivation tree for *aba* in $\mathfrak{G}'$.

- S\Det $\xrightarrow{1/2}$ *N* S\NP
- N $\xrightarrow{1}$ ~N
- ~N $\xrightarrow{1/2}$ *the*
- S\NP $\xrightarrow{1}$ VP S\S
- VP $\xrightarrow{1}$ ~V VP\V
- ~V $\xrightarrow{1}$ *ran*
- VP\V $\xrightarrow{1}$ ADV VP\VP
- ADV $\xrightarrow{1}$ ~ADV
- ~ADV $\xrightarrow{1/2}$ *fast*
- VP\VP $\xrightarrow{1}$ ε
- S\S $\xrightarrow{1}$ ε

We, again, get the weight of the derivation by multiplying over all rules, and observe that

$$\mathcal{W}'(\boldsymbol{t}') = \mathcal{W}(\boldsymbol{t}) = \frac{1}{16} \tag{1.64}$$

The corresponding derivation tree for $\boldsymbol{y}$ *is illustrated on the right-hand side of Fig. 1.9.*

Another interesting feature of the left-corner transform is that dominance relationships in $\mathfrak{G}$ *are inverted in* $\mathfrak{G}'$*. For instance, the left-most chain* (S NP Det) *in* $\mathfrak{G}$ *corresponds to the right-most corner chain* (S\Det S\NP S\NP) *in* $\mathfrak{G}'$*.*

1.3.5 Speculation

A closely related transform to left-corner is speculation. Originally introduced for logic programs (Eisner and Blatz, 2007), we here present an analogous transform for WCFGs.

With the speculation, we can separate the dependence on non-terminals in the grammar by creating speculative rules. More formally, by separating a dependence we mean that if $\mathrm{X} \overset{*}{\Rightarrow} \mathrm{Y}$ in $\mathfrak{G}$, then a new non-terminal X\Y will not derive Y in $\mathfrak{G}'$. We present an example of this below.

In addition, speculation has a close connection to the folding transform. In particular, folding can be seen as a special case of speculation.

As for the left-corner transform, speculation admits a simple set of rules to map a grammar to its speculative transform. The newly created rules once again include slashed and marked non-terminals, and the transform again takes as input a set of non-terminals $\mathcal{N}_{\backslash}$ and a set of production rules $\mathcal{P}_{\backslash}$. The set of non-terminals $\mathcal{N}_{\backslash}$ corresponds to the non-terminals that will be slashed on and the set of rules $\mathcal{P}_{\backslash}$ will be the rules in which we slash. In addition, we require a map $\mu : \mathcal{P}_{\backslash} \mapsto \mathbb{N}$. For a given rule p, $\mu(p)$ determines the position on the right-hand side of the rule where we slash. As an example, if $p = \mathrm{X} \to \mathrm{Y}\,\mathrm{Z}$ is in $\mathcal{P}_{\backslash}$, with $\mathrm{A} \in \mathcal{N}_{\backslash}$ and $\mu(p) = 1$, we get the new rule $\mathrm{X}\backslash\mathrm{A} \to \mathrm{Y}\backslash\mathrm{A}\,\mathrm{Z}$.

The complete set of rules that define the transform are presented below.

Transform 1.3.2.

$$\mathrm{Y}\backslash\mathrm{Y} \xrightarrow{1} \varepsilon \qquad \forall \mathrm{Y} \in \mathcal{N}_{\backslash} \tag{1.65}$$

$$\sim\mathrm{X} \xrightarrow{w} \boldsymbol{\alpha} \qquad \forall (\mathrm{X} \xrightarrow{w} \boldsymbol{\alpha}) \notin \mathcal{P}_{\backslash} \tag{1.66}$$

$$\mathrm{X}\backslash\mathrm{Y} \xrightarrow{w} \mathrm{X}_1 \cdots \mathrm{X}_{\mu(p)-1}\ \mathrm{X}_{\mu(p)}\backslash\mathrm{Y}\ \mathrm{X}_{\mu(p)+1} \cdots \mathrm{X}_n \qquad \forall p = (\mathrm{X} \xrightarrow{w} \mathrm{X}_1 \cdots \mathrm{X}_n) \in \mathcal{P}_{\backslash}, \forall \mathrm{Y} \in \mathcal{N}_{\backslash} \tag{1.67}$$

$$\sim\mathrm{X} \xrightarrow{w} \mathrm{X}_1 \cdots \mathrm{X}_{\mu(p)-1}\ \sim\mathrm{X}_{\mu(p)}\ \mathrm{X}_{\mu(p)+1} \cdots \mathrm{X}_n \qquad \forall (\mathrm{X} \xrightarrow{w} \mathrm{X}_1 \cdots \mathrm{X}_n) \in \mathcal{P}_{\backslash}, \mathrm{X}_{\mu(p)} \notin \mathcal{N}_{\backslash} \tag{1.68}$$

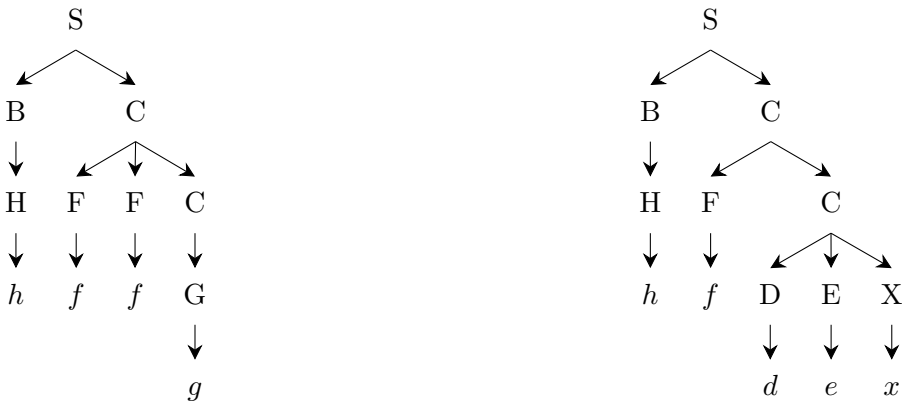
$$\mathrm{X} \xrightarrow{1} \sim X \qquad \forall \mathrm{X} \notin \mathcal{N}_{\backslash} \tag{1.69}$$

$$\mathrm{X} \xrightarrow{1} \mathrm{X}\backslash\mathrm{Y}\ \sim\mathrm{Y} \qquad \forall \mathrm{X} \in \mathcal{N}, \mathrm{Y} \in \mathcal{N}_{\backslash} \tag{1.70}$$

Next, we provide an example to better understand the speculation transform.

Example 1.3.7 (Speculation)**.** *The following example is adapted from Eisner and Blatz (2007). We let $\mathfrak{G}$ be a CFG defined by the following rules:*

- $\mathrm{S} \to \mathrm{B}\,\mathrm{C}$
- $\mathrm{B} \to \mathrm{H}$
- $\mathrm{C} \to \mathrm{F}\,\mathrm{C} \mid \mathrm{D}\,\mathrm{E}\,\mathrm{X} \mid \mathrm{G}$
- $D \to d$
- $E \to e$
- $F \to f$
- $G \to g$
- $H \to h$
- $X \to x$



with $\mathrm{S} = \mathrm{S}$, $\mathcal{N} = \{\mathrm{S}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{X}\}$ *and* $\Sigma = \{d, e, f, g, h, x\}$. *Note that the language of* $\mathfrak{G}$ *is*

$$L(\mathfrak{G}) = \{hf^n g \mid n \geq 0\} \cup \{hf^n dex \mid n \geq 0\}$$

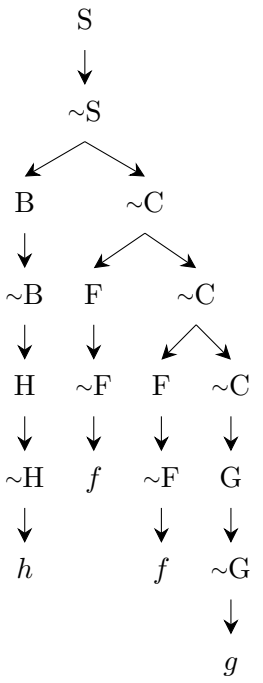
See two examples of derivation trees in the grammar in the figure below.

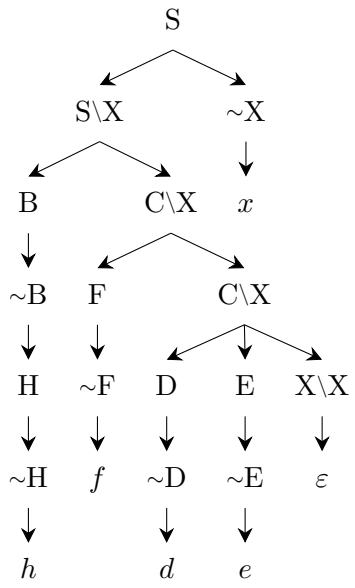
Next, we apply speculation in order to separate the dependency on X. *Note that the non-terminals that derive* X *are* S *and* C. *We hence set* $\mathcal{N}_{\backslash} = \{\mathrm{X}\}$ *and select the speculative rules* $\mathcal{P}_{\backslash} = \{\mathrm{S} \to \mathrm{B\,C}, \mathrm{C} \to \mathrm{F\,C}, \mathrm{C} \to \mathrm{D\,E\,X}\}$ *with* $\mu(\mathrm{S} \to \mathrm{B\,C}) = 2$, $\mu(\mathrm{C} \to \mathrm{F\,C}) = 2$ *and* $\mu(\mathrm{C} \to \mathrm{D\,E\,X}) = 3$ *in order to hit the dependency on* X. *As a result we get a grammar* $\mathfrak{G}'$ *with the following non-useless rules:*

- $\mathrm{S} \to \mathrm{S}\backslash\mathrm{X}\sim\mathrm{X}$
- $\mathrm{Y} \to \sim\mathrm{Y}, \quad \forall \mathrm{Y} \in \mathcal{N} \setminus \mathcal{N}_{\backslash}$
- $\sim\mathrm{Y} \to y, \quad \forall \mathrm{Y} \in \{\mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{X}\}$
- $\sim\mathrm{S} \to \mathrm{B}\sim\mathrm{C}$
- $\sim\mathrm{C} \to \mathrm{F}\sim\mathrm{C}$
- $\mathrm{S}\backslash\mathrm{X} \to \mathrm{B\,C}\backslash\mathrm{X}$
- $\mathrm{C}\backslash\mathrm{X} \to \mathrm{F\,C}\backslash\mathrm{X} \mid \mathrm{D\,E\,X}\backslash\mathrm{X}$
- $\mathrm{X}\backslash\mathrm{X} \to \varepsilon$

By staring at the above for a bit you should be able to convince yourself that the transform has preserved the language in the original grammar. Next, we show the corresponding derivation trees from above in $\mathfrak{G}'$ in the figure below.

Note that the dependence on X is separated already at the first rule at the root – $\mathrm{S} \to \sim\mathrm{S}$ will derive a string without x and $\mathrm{S} \to \mathrm{S}\backslash\mathrm{X}\sim\mathrm{X}$ will derive a string with x. Further note that the yield of all subtrees rooted at the slashed non-terminals on X have the same yield as the corresponding subtrees in the original grammar, but with a missing x. For instance, the lower subtree rooted at $\mathrm{C}\backslash\mathrm{X}$ yields de in $\mathfrak{G}$, while the corresponding subtree rooted at C in $\mathfrak{G}$ yields dex.





1.4 The Parsing Problem

1.4.1 Introducing Parsing

This section discusses the parsing problem. Traditionally, parsing concerns itself with assigning a structure for a given input string, which in the context of CFGs translates into finding the set of derivation trees that may have generated the string. This is a *recognition problem*. However, with the semiring weighted CFGs we have discussed in the previous two chapters, we can perform a simple generalization over the traditional parsing algorithms in order to go beyond recognition. Some common applications of weighted parsing in NLP include computing the normalizing constant (real semiring) or finding the most likely derivation tree for an input string (max-plus semiring).

As we have alluded to, the difference compared to computing the treesum of a WCFG is that instead of considering *all* derivation trees admissible under the grammar, we are interested in a given string $\boldsymbol{y}$. As such, the parsing problem is really nothing more than finding the string sum $Z_{\mathfrak{G}}(\boldsymbol{y})$. By only considering a given string we can restrict the search space, rendering the parsing problem to be a more computationally tractable problem than computing the treesum. It is also a more well-studied and arguably more practically useful problem, and there exist numerous efficient parsing algorithms. Perhaps the most famous one is CKY (Kasami, 1965; Younger, 1967; Cocke, 1969), but we will also discuss others.

First, let us start by giving an example.

Example 1.4.1 (The Parsing Problem)**.** *Consider the natural language grammar we saw in the*

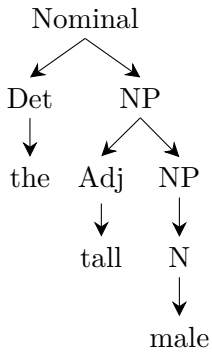


Figure 1.12: Derivation tree for "the tall male".

beginning of §1.1, which we augment with weights in the Tropical semiring as follows:

$$\begin{aligned}
&\text{Nominal} \xrightarrow{2} \text{Det NP}\\
&\text{NP} \xrightarrow{3} \text{N}\\
&\text{NP} \xrightarrow{2} \text{Adj NP}\\
&\text{Det} \xrightarrow{2} \textit{a} \mid \textit{the}\\
&\text{N} \xrightarrow{3} \textit{giraffe}\\
&\text{N} \xrightarrow{2} \textit{female} \mid \textit{male}\\
&\text{Adj} \xrightarrow{3} \textit{big} \mid \textit{tall}\\
&\text{Adj} \xrightarrow{1} \textit{female} \mid \textit{male}
\end{aligned}$$

Say we want to parse the sentence $\boldsymbol{y}$ ="the tall male". In this case, although there is local ambiguity at "male", there is only one correct derivation tree for this string. It can be seen in Fig. 1.12. We compute the string sum over this one tree as

$$\begin{aligned}
Z_{\mathfrak{G}}(\boldsymbol{y}) &= \min_{\boldsymbol{t}\in\mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})} \sum_{p\in\boldsymbol{t}} \mathcal{W}(p)\\
&= 2+2+2+3+3+1\\
&= 13
\end{aligned}$$

We make two observations about the correct derivation tree. Clearly, it must have a yield equal to the string of interest. In addition, any valid derivation tree must be rooted at the start symbol. With these two constraints in mind, our goal is to compute the string sum over all such correct derivation trees for the string.

As made clear in the example above, we have two constraints to consider when parsing a string. Firstly, the derivation tree must have n leaf nodes (not counting empty strings) that correspond to the n terminal symbols in a string $\boldsymbol{y}$ of length n. Secondly, the derivation tree must be a valid such tree under the grammar. In particular, the root of the derivation tree must be the start symbol S. These two constraints lead to two separate parsing paradigms: **bottom-up parsing** and **top-down parsing**.

Bottom-up parsers start from the input string, and build up subtrees using applicable rules in the grammar. A successful derivation tree is found when the parser reaches the start symbol.

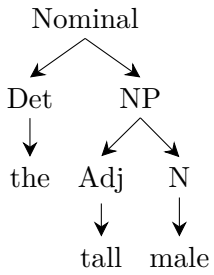


Figure 1.13: Derivation tree for "the tall male" in CNF.

Top-down parsers, on the other hand, take the other direction. They start from the start symbol S and work their way down by applying rules in the grammar.

1.4.2 The CKY Algorithm

The first parsing algorithm we discuss is a bottom-up parser—CKY. CKY is a dynamic program that leverages the structure entailed by a grammar being in CNF. For a grammar in CNF, we know that all terminals in the string must have been produced by a unary rule that produces only that terminal. As such we can search for all minimal subtrees, i.e. subtrees rooted at non-terminals that are parents of terminal nodes, in worst-case $\mathcal{O}(|\boldsymbol{y}| \cdot |\mathcal{P}|)$ time. By the second rule type, rules of the form A $\rightarrow$ B C, we can split all other subtrees into exactly two smaller subtrees. This allows for efficient dynamic programming – we can compute the string sum for all such subtrees and work our way up to the start symbol, storing the results as we go.

Let us give some more details. First, note that any derivation tree of a string $\boldsymbol{y}$ with length n can be represented by the upper triangular portion of an $(n+1) \times (n+1)$ dimensional matrix. We will call this a chart. The element at position $[i, j]$ will correspond to a non-terminal, the subtree of which spans the input between position i and j. Hence, letting the indexing start at 0, all terminals in the string will be represented by spans $[i, i+1]$, with $i \in \{0, 1, \ldots, n-1\}$. The start symbol will span the entire input, and will therefore be represented at position $[0, n]$ of the chart, i.e. the upmost, rightmost element. Note that the triangular chart is not fully populated – we only have an element at positions where there is a non-terminal that spans that corresponding part of the input.

As an example, consider the derivation tree in Example 1.4.1 transformed into CNF, as presented in Fig. 1.13. We index the string by ($_0$ the $_1$ tall $_2$ male $_3$). The non-terminal Det will span the input at $[0, 1]$, and therefore be represented at position $[0, 1]$ in the chart. Likewise, the non-terminal NP will span the input at $[1, 3]$ and hence be represented at position $[1, 3]$ in the chart. There is no subtree that spans the input at index $[0, 2]$, and the corresponding entry will hence be empty.

We do not want to include just one derivation tree however, but all trees that may produce the string, a so-called **parse forest** of $\boldsymbol{y}$. Hence, we add a third dimension to the chart, one that corresponds to the non-terminal that roots a subtree that spans the input at the corresponding indices. Its value will be the weight of all subtrees rooted at this non-terminal that span the input at these indices. For a non-terminal A and span $[i, j]$, we will call the set of such subtrees a parse forest rooted at A spanning $[i, j]$. For example, in Fig. 1.13, the weight of the subtree rooted at NP will be found at position $[\text{NP}, 1, 3]$ in the chart.

The idea of CKY is to populate this chart bottom-up. We do so by iterating over the length of the spans. We start with the spans of length 1, i.e., spans over one terminal symbol. Making use of the fact that all subtrees that span one terminal symbol must have one rule only, specifically on the form A $\rightarrow x$, we fill in the weight of such rules at positions $[\text{A}, i, i+1]$ in the chart – at indices

Algorithm 10 The weighted CKY algorithm.

def WeightedCKY($\mathfrak{G}$, $\boldsymbol{y}$):
  $\boldsymbol{\beta} \leftarrow \mathbf{0}$ ▷*Initialize the chart*
  $n \leftarrow |\boldsymbol{y}|$
  for $i = 0, \ldots, n-1$:
    for $\mathrm{A} \rightarrow x \in \mathcal{P}$: ▷*Iterate over unary rules*
      if $\boldsymbol{y}_i = x$:
        ▷ *If we find rule that produces the terminal at position i, add its weight*
        $\boldsymbol{\beta}([\mathrm{A}, i, i+1]) \leftarrow \boldsymbol{\beta}([\mathrm{A}, i, i+1]) \oplus \mathcal{W}(\mathrm{A} \rightarrow x)$
  for span $= 2, \ldots, n$: ▷*Iterate over span lengths*
    for $i = 0, \ldots, n - \text{span}$: ▷*The start of the span*
      $k \leftarrow i + \text{span}$ ▷*The end of the span*
      for $j = i+1, \ldots, k-1$: ▷*Iterate over all possible spans of subtrees*
        for $\mathrm{A} \rightarrow \mathrm{B\,C} \in \mathcal{P}$:
          $\boldsymbol{\beta}([\mathrm{A}, i, k]) \leftarrow \boldsymbol{\beta}([\mathrm{A}, i, k]) \oplus \boldsymbol{\beta}([\mathrm{B}, i, j]) \otimes \boldsymbol{\beta}([\mathrm{C}, j, k]) \otimes \mathcal{W}(\mathrm{A} \rightarrow \mathrm{B\,C})$
          ▷ *We multiply the weight of the subtrees with the weight of the rule and add to the chart*
  return $\boldsymbol{\beta}([\mathrm{S}, 0, n])$

where x matches the terminal symbol in the span $[i, i+1]$ of the input.

Next, we consider spans of length > 1. Note that, due to the rules being on the form $\mathrm{A} \rightarrow \mathrm{B\,C}$, all parse forests that span the input at $[i, j]$ can be split up in *exactly two* parse forests that together span $[i, j]$. In other words, there must exist some k with $i < k < j$, such that there exist two parse forests of the non-terminal that span $[i, j]$, one spanning $[i, k]$ and the other spanning $[k, j]$. We can use this insight to construct an efficient dynamic program, in which we traverse up the chart by combining parse forests. The weight of the chart at each position $[\mathrm{A}, i, j]$ will be computed by multiplying the weights of the subtrees of A with the rule headed by A, and summing over all such rules headed by A.

We present the pseudocode in Alg. 10. As can be seen by reading off the for loops, the runtime of CKY is $\mathcal{O}(|\boldsymbol{y}|^3 \cdot |\mathcal{P}|)$. The space complexity required is the size of the chart, which is $\mathcal{O}(|\boldsymbol{y}|^2 \cdot |\mathcal{N}|)$.

1.4.3 Bar-Hillel Parsing

Let us now present a generalization of CKY that can parse multiple strings in one go, which we call Bar-Hillel parsing (Bar-Hillel et al., 1961). The Bar-Hillel parsing algorithm makes use of the fact that a CFG intersects with an FSA, as we discussed in §1.1.

Recall that we can encode a string as a linear FSA that accepts only that string. In the weighted case, we set all weights of the WFSA to $\mathbf{1}$. Recall also, from **??**, that we can concatenate FSAs. In the Bar-Hillel parsing algorithm, we take a concatenation of an arbitrary number of strings, $\boldsymbol{y} = \boldsymbol{y}_1 \circ \cdots \circ \boldsymbol{y}_N$, and return the sum over the string sums for these strings, i.e.

$$\bigoplus_{i=1}^{N} Z_{\mathfrak{G}}(\boldsymbol{y}_i) \tag{1.71}$$

Note that in the standard CKY algorithm, such a task would require us to compute the treesum separately for each string, since the concatenated string $\boldsymbol{y}$ does not necessarily have to be in the language of the grammar.

Algorithm 11 The Bar-Hillel parsing algorithm.

def BarHillelParsing($\mathfrak{G}$, $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$):
  $\boldsymbol{y} \leftarrow \boldsymbol{y}_1 \circ \cdots \circ \boldsymbol{y}_N$ ▷*Concatenate strings*
  $\mathcal{A} \leftarrow \textsc{LinearWFSA}(\boldsymbol{y})$ ▷*Encode concatenated string as linear WFSA*
  $\boldsymbol{\beta} \leftarrow \mathbf{0}$ ▷*Initialize the chart*
  $n \leftarrow |Q|$
  $O \leftarrow \mathsf{topo}(\mathcal{A})$ ▷*Get topological ordering over states*
  for $q \in O$: ▷*Retrieve states and indices in topological order*
    for $(x, q', w) \in \mathcal{E}_{\mathcal{A}}(q)$: ▷*Take all outgoing arcs from q*
      for $\mathrm{A} \to x \in \mathcal{P}$: ▷*Iterate over unary rules*
        ▷ *If we find rule that produces the terminal at position i, add its weight*
        $\boldsymbol{\beta}([\mathrm{A}, O(q), O(q')]) \leftarrow \boldsymbol{\beta}([\mathrm{A}, i, i+1]) \oplus \mathcal{W}(\mathrm{A} \to x) \otimes w$
  ▷ *Standard CKY loop*
  for span $= 2, \ldots, n$: ▷*Iterate over span lengths*
    for $i = 0, \ldots, n - \text{span}$: ▷*The start of the span*
      $k \leftarrow i + \text{span}$ ▷*The end of the span*
      for $j = i+1, \ldots, k-1$: ▷*Iterate over all possible spans of subtrees*
        for $\mathrm{A} \to \mathrm{B}\,\mathrm{C} \in \mathcal{P}$:
          $\boldsymbol{\beta}([\mathrm{A}, i, k]) \leftarrow \boldsymbol{\beta}([\mathrm{A}, i, k]) \oplus \boldsymbol{\beta}([\mathrm{B}, i, j]) \otimes \boldsymbol{\beta}([\mathrm{C}, j, k]) \otimes \mathcal{W}(\mathrm{A} \to \mathrm{B}\,\mathrm{C})$
          ▷*We multiply the weight of the subtrees with the weight of the rule and add to the chart*
  $\boldsymbol{\gamma} \leftarrow \mathbf{0}$ ▷*Initialize output*
  for $(q, w) \in I$:
    for $(q', w') \in F$:
      $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \oplus w \otimes \boldsymbol{\beta}([\mathrm{S}, O(q), O(q')]) \otimes w'$
  return $\boldsymbol{\gamma}$

We present the pseudocode in Alg. 11. Note that the a topological ordering over the states in a WFSA that encodes $\boldsymbol{y}$, will correspond to the indices as we have used them for CKY.

1.4.4 Agenda-based Parsing

Next, we present an instance of what is known as **agenda-based parsers** (Knuth, 1977). Note that in CKY, the order of the computations is fixed: We traverse left-to-right inside and outside of spans, with an increasing order of the span. Agenda-based parsers, instead, dynamically determine the order of the computations depending on the properties of the grammar. They do so by maintaining an **agenda**, which is an explicit ordering of items that are yet to be parsed.

The algorithm we present assumes that the grammar is in CNF, with the addition of unary rules.[22] The ordering of the agenda is defined by the ordering of the weights over the items in the agenda, and all items take the form $[\mathrm{A}, i, j]$ for some non-terminal A and span of the input $[i, j]$. Like for CKY, the weights of the items in the agenda will at each step correspond to the sum over the computed weights of the subtrees rooted at A that span $[i, j]$ of the input. The agenda acts as a priority queue, and whether the algorithm proceeds in an increasing or decreasing order of the weights depends on the semiring.

[22] Note that we can easily extend the CKY algorithm to handle unary production rules in a similar approach as presented here.

Algorithm 12 The agenda-based parsing algorithm.

```
def AgendaParsing(𝔊, y):
  β ← 0                                                    ▷Initialize the chart
  agenda ← 0                                               ▷Initialize the agenda
  n ← |y|
  for i = 0, ..., n − 1 :
    for A → x ∈ P :                                        ▷Iterate over unary rules
      if y_i = x :
        ▷ If we find rule that produces the terminal at position i, add item to agenda
        push ([A, i, i + 1], W(A → x)) to agenda
  while |agenda| > 0 :
    pop (item, w) from agenda
    β([item]) ← β([item]) ⊕ w
    ▷ Early stopping
    if item = [S, 0, n] : return β([S, 0, n])
    ▷ Unary attachment
    [A, i, k] ← item
    for unary chains B ⇒* A in 𝔊 :
      if β([B, i, k]) == 0 :
        push ([B, i, k], β([A, i, k]) ⊗ W(B ⇒* A)) to agenda
    ▷ Right attachment
    [B, i, j] ← item
    for k = j + 1, ..., n :
      for A → B C ∈ P :
        if β([C, j, k]) ≠ 0 :
          push ([A, i, k], β[B, i, j] ⊗ β([C, j, k]) ⊗ W(A → B C) to agenda
    ▷ Left attachment
    [C, j, k] ← item
    for i = 0, ..., j − 1 :
      for A → B C ∈ P :
        if β([B, i, j]) ≠ 0 :
          push ([A, i, k], β([B, i, j]) ⊗ β([C, j, k]) ⊗ W(A → B C) to agenda
  return β([S, 0, n])
```

Hence, the algorithm follows the bottom-up scheme, and similarly to CKY starts by processing all terminal rules in accordance to the input string. For all rules $\mathrm{A} \xrightarrow{w} x$ for which there is a matching terminal symbol at some span $[i, i+1]$ of the input string, we add the **item** $[\mathrm{A}, i, i+1]$ to the agenda with weight w. After that, it iterates the ordered elements of the agenda and advances up the tree. The main idea is that given that the item we consider is the first on in the ordering of the agenda, we must have finished parsing all subtrees of that item. Take for instance the tropical semiring $\mathcal{W} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. We proceed in an increasing order of the weights for this semiring, since for a subtree $\boldsymbol{t}$ for which we have $\boldsymbol{t} \prec \boldsymbol{t}'$ in the partial order over subtrees, it must hold that $\mathcal{W}(\boldsymbol{t}) \leq \mathcal{W}(\boldsymbol{t}')$.

The weights are maintained in a chart $\boldsymbol{\beta}$, which is updated at the start of each iteration. The tree is then traversed upward from three angles: (i) unary attachment, (ii) right attachment and

(iii) left attachment.

In the first step, unary attachment, we consider unary chains. Weights over unary chains are pre-computed in the same fashion as we saw in the previous chapter on unary rule removal, by computing the pathsum over unary WFSAs. For item $[\mathrm{A}, i, j]$, we take each B for which there is a unary chain from B to A and add $[\mathrm{B}, i, j]$ with weight

$$\boldsymbol{\beta}([\mathrm{A}, i, j]) \otimes \mathcal{W}(\mathrm{B} \stackrel{*}{\Rightarrow} \mathrm{A})$$

to the agenda, for all such $[\mathrm{B}, i, j]$ that have not already been parsed.

In the right and left attachment steps, we consider binary rules. In right attachment, for a given item $[\mathrm{B}, i, j]$ we take all rules such that $\mathrm{A} \rightarrow \mathrm{B}\,\mathrm{C}$ where there is some item $[\mathrm{C}, j, k]$ that has already been parsed. For these instances, we add the new item $[\mathrm{A}, i, k]$ with weight

$$\boldsymbol{\beta}([\mathrm{B}, i, j]) \otimes \boldsymbol{\beta}[\mathrm{C}, j, k] \otimes \mathcal{W}(\mathrm{A} \rightarrow \mathrm{B}\,\mathrm{C})$$

to the agenda. Intuitively, we take all subtrees to the right of the given non-terminal-span pair that can appear subsequently in a derivation, and attach them. The left attachment step works analogously. Do also note the similarity of the update step to CKY.

We present the pseudocode in Alg. 12.

1.4.5 Earley's Algorithm

Earley's algorithm (Earley, 1970) is another dynamic program. In contrast to CKY and the agenda-based algorithm however, it follows the top-down scheme. Moreover, it does not require a specific normal form of the grammar.

We describe Earley's aglorithm using a **deduction system**.[23] A deduction system proves **items** V using **deduction rules**. Items represent propositions; the rules are used to prove all propositions that are true. A deduction rule is of the form

$$\textsc{Example:}\ \frac{U_1 \qquad U_2 \qquad \cdots}{V}$$

where Example is the name of the rule, the 0 or more items above the bar are called **antecedents**, and the single item below the bar is called a **consequent**. Antecedents may also be written to the side of the bar; these are called **side conditions** and will be handled differently for weighted parsing (as we will explain below). **Axioms** (listed separately) are merely rules that have no antecedents; as a shorthand, we omit the bar in this case and simply write the consequent.

Our unweighted recognizer will determine whether a certain **goal item** is provable by a certain set of deduction rules from axioms that encode $\mathfrak{G}$ and $\boldsymbol{y}$. The deduction system is set up so that this is the case iff $\mathrm{S} \stackrel{*}{\Rightarrow} \boldsymbol{y}$. The provability of an item under a deduction system can be generically solved through forward-chaining. A parser additionally returns one or more actual proofs of the goal item, which are tree-structured. In general, a **proof tree** (or just proof) of an item V is a tree rooted at V, whose nodes are items and whose leaves are axioms.

The proof trees of Earley's algorithm are in one-to-one correspondence with the derivation trees that derive $\boldsymbol{y}$. There are axioms $\mathrm{A} \rightarrow \boldsymbol{\alpha}$ for each production rule in the grammar, as well as axioms on the form $[j, j+1, a]$ which are true iff the jth position of the input string is equal to the terminal symbol a. All other items take the form:

$$[i, j, \mathrm{A} \rightarrow \boldsymbol{\alpha} \bullet \boldsymbol{\beta}],$$

[23] The other parsing algorithms described above can be described with deduction systems as well. In an exercise, we ask you to write CKY as a deduction system.

with $\boldsymbol{\alpha}, \boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^*$. The item $[i, j, \mathrm{A} \rightarrow \boldsymbol{\alpha} \bullet \boldsymbol{\beta}]$ is derivable only if there is a rule $\mathrm{A} \rightarrow \boldsymbol{\alpha}\,\boldsymbol{\beta}$ in the grammar such that $\boldsymbol{\alpha} \stackrel{*}{\Rightarrow} \boldsymbol{y}_{i:j}$. The dot $\bullet$ thus represents the progress that has been made on this particular production rule. In a sense, it can be seen as a generalization of the binarization done in CKY – the $\bullet$ splits the rule in two separate constituents. We say that an item is **completed** if $\bullet$ is at the rightmost position. The set of all items with a shared right index j is called the **item set** of j, denoted $\mathcal{S}_j$. Earley's processes all items in a given item set $\mathcal{S}_j$ before proceeding to the next item set $\mathcal{S}_{j+1}$.

Let us give a couple of examples corresponding to the rules of the derivation tree in *Fig.* 1.12. The item

$$[0, 0, \text{Nominal} \rightarrow \bullet\, \text{Det NP}]$$

represents the start of the Earley algorithm on this particular grammar. The first 0 indicates that the tree rooted at Nominal spans the start of the input string. The second index being equal to the first (0) indicates that no progress has been made on this production rule, reflecting that $\bullet$ is on the 0th position. The position of $\bullet$ tells us that we are expecting to process a rule with left-hand side Det next. Another item

$$[1, 2, \text{NP} \rightarrow \text{Adj} \bullet \text{NP}]$$

states that we have found a constituent Adj that derives $\boldsymbol{y}_{1:2}$.

Including $[0, 0, \mathrm{S}' \rightarrow \bullet\, \mathrm{S}]$ as an axiom in the system effectively causes forward-chaining to start looking for a derivation at position 0. The system has proved $\mathrm{S} \stackrel{*}{\Rightarrow} \boldsymbol{y}$ if it can prove the goal item $[0, N, \mathrm{S}' \rightarrow \mathrm{S} \bullet]$, where $N = \boldsymbol{y}$.

There are three deduction rules: Predict, Scan and Complete. Scan consumes the next single input symbol, Predict calls a subroutine to consume an entire constituent of a given nonterminal type by recursively consuming its subconstituents, and Complete returns from that subroutine.

Predict To look for constituents rooted at a nonterminal B starting at position j, using the rule $\mathrm{B} \rightarrow \rho$, we need to prove $[j, j, B \rightarrow \bullet\, \rho]$. Earley's algorithm imposes $[i, j, A \rightarrow \mu \bullet B\, \nu]$ as a side condition, so that we only start looking if such a constituent could be combined with some item to its left.

$$\textsc{Pred:}\ \frac{\mathrm{B} \rightarrow \rho}{[j, j, B \rightarrow \bullet\, \rho]}\ [i, j, A \rightarrow \mu \bullet B\, \nu]$$

Taking the grammar and string in Example 1.4.1, applying Predict to $[0, 0, \text{Nominal} \rightarrow \bullet\, \text{Det NP}]$, derives the new items $[0, 0, \text{Det} \rightarrow \bullet\, \text{a}]$ and $[0, 0, \text{Det} \rightarrow \bullet\, \text{the}]$.

Scan If we have proved an incomplete item $[i, j, A \rightarrow \mu \bullet a\, \nu]$, we can advance the dot if the next terminal symbol is a:

$$\textsc{Scan:}\ \frac{[i, j, A \rightarrow \mu \bullet a\, \nu] \quad [j, k, a]}{[i, k, A \rightarrow \mu\, a \bullet \nu]}$$

This makes progress toward completing the A. Note that Scan pushes the antecedent to a subsequent item set $\mathcal{S}_k$. Since terminal symbols have a span width of 1, it follows that $j = k - 1$.

As an example, applying Scan to $[0, 0, \text{Det} \rightarrow \bullet\, \text{the}]$ results in $[0, 1, \text{Det} \rightarrow \text{the} \bullet]$, since the word "the" is indeed found at position $[0, 1]$ of the input string.

Complete Recall that having $[i, j, A \to \mu \bullet B\, \nu]$ allowed us to start looking for a B at position j (PRED). Once we have found a complete B by deriving $[j, k, B \to \rho \bullet]$, we can advance the dot in the former rule:

$$\textsc{Comp:}\ \frac{[i, j, A \to \mu \bullet B\, \nu] \quad [j, k, B \to \rho \bullet]}{[i, k, A \to \mu\, B \bullet \nu]}$$

Considering again the example above, the completed item $[0, 1, \text{Det} \to \text{the} \bullet]$ in $\mathcal{S}_1$ will be applied to COMPLETE and result in the new item $[0, 1, \text{Nominal} \to \text{Det} \bullet \text{NP}]$, added to $\mathcal{S}_1$. That is, since $[0, 0, \text{Nominal} \to \bullet\, \text{Det NP}]$ was found in $\mathcal{S}_0$.

Weighted deduction So far we have not mentioned the string sums, but rather laid the groundwork for a recognition algorithm. We first observe that by design, the derivation trees of the CFG are in 1-1 correspondence with the proof trees of our deduction system that are rooted at the goal item. Furthermore, the weight of a derivation subtree can be found as the weight of the corresponding proof tree, if the weight of any proof tree rooted at an item V is defined recursively as follows.

Base case: The proof tree may be a single node, i.e., V is an axiom. If V has the form $A \to \rho$, then its weight is the corresponding grammar production, i.e., $w\,(A \to \rho)$. All other axiomatic proof trees have weight $\mathbf{1}$.

Recursive case: If the final step of the proof tree uses the deduction rule EXAMPLE to prove V from U_1 and U_2, then the weight of the proof tree rooted at V is the weight of the proof tree rooted at U_1 $\otimes$-multiplied with the weight of the proof tree rooted at U_2. Generalizing this to other rules, the factors in this product include only the weights of the antecedents written above the bar, *not the side conditions*.

Like for the parsing algorithms presented above, we also associate a weight with each proved item V, denoted $\boldsymbol{\beta}(V)$, which is the *total* weight of *all* its proof trees. By the distributive property, we can obtain that weight as an $\oplus$-sum over all one-step proofs of V from previously proved antecedents and side conditions. In this $\oplus$-sum, each one-step proof (via an instantiated rule) contributes the $\otimes$-product of the weights of its antecedent items. Thus, we get the following weight updates for the three deduction rules:

$$\begin{aligned}
&\textsc{Pred}: && \boldsymbol{\beta}([j, j, B \to \bullet\, \rho]) \oplus\!= \boldsymbol{\beta}(\text{B} \to \rho) \\
&\textsc{Scan}: && \boldsymbol{\beta}([i, k, A \to \mu\, a \bullet \nu]) \oplus\!= \boldsymbol{\beta}([i, j, A \to \mu \bullet\, a\, \nu]) \otimes \boldsymbol{\beta}([j, k, a]) \\
&\textsc{Comp}: && \boldsymbol{\beta}([i, k, A \to \mu\, B \bullet \nu]) \oplus\!= \boldsymbol{\beta}([i, j, A \to \mu \bullet B\, \nu]) \otimes \boldsymbol{\beta}([j, k, B \to \rho \bullet])
\end{aligned}$$

$\boldsymbol{\beta}$ of the goal item will by definition be the total weight of all proof trees of the goal item, which corresponds to the desired $Z_{\mathfrak{G}}(\boldsymbol{y})$.

A note on cycles The deduction system works for any semiring-weighted CFG. Unfortunately, the forward-chaining algorithm for weighted deduction may not terminate if the system permits *cyclic* proofs, where an item can participate in one of its own proofs. In this case, the algorithm will merely approach the correct value of $Z_{\mathfrak{G}}(\boldsymbol{y})$ as it discovers deeper and deeper proofs of the goal item. Cyclicity in our system can arise from sets of unary productions such as $\{\text{A} \to \text{B}, \text{B} \to \text{A}\} \subseteq \mathcal{P}$, or equivalently, from $\{\text{A} \to \text{E B E}, \text{B} \to \text{A}\} \subseteq \mathcal{P}$ where $\text{E} \overset{*}{\Rightarrow} \varepsilon$ (which is possible if $\mathcal{P}$ contains $\text{E} \to \varepsilon$ or other nullary productions). To avoid these issues, we can eliminate unary and nullary production as described in §1.3.

1.5 Pushdown Automata

1.5.1 Pushdown Automata

Now we introduce a new type of computational device, called *pushdown automaton*. A pushdown automaton is very similar to a finite-state automaton but it has an extra component called a *stack*. The stack provides some additional memory beyond the finite set of states of the machine, enabling them to recognize context-free languages. Pushdown automata[24] have equivalent expressive power to context-free grammars, both characterizing context-free languages. Pushdown automata *recognize* context-free languages while context-free grammars *generate* them.

Definition 1.5.1. *A **pushdown automaton** (PDA) is a* 6*-tuple* $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of input symbols, called the* input alphabet*;*
- Γ *is a finite set of stack symbols, called the* stack alphabet*;*
- $\delta \subseteq Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$ *is a finite multi-set, with elements that are generally called* transitions*;*
- $(q_I, \boldsymbol{\gamma}_I)$ *and* $(q_F, \boldsymbol{\gamma}_F)$*, where* $q_I, q_F \in Q$*,* $\boldsymbol{\gamma}_I, \boldsymbol{\gamma}_F \in \Gamma^*$*, are called the* initial *and* final configuration*.*

A stack is a memory structure that allows access only to its top elements. Therefore, a pushdown automaton can push and pop sequences of stack symbols to and from the top of the stack.

Definition 1.5.2. *A **stack** $\boldsymbol{\gamma} \in \Gamma^*$ is a sequence of stack symbols.*

Stacks are represented as strings over Γ, from the bottom to top. If $\boldsymbol{\gamma} = X_1 \cdots X_n$ is a stack, the symbol X_1 is at the bottom while X_n is at the top.

Definition 1.5.3. *A **configuration** of a PDA is a pair* $(q, \boldsymbol{\gamma})$*, where* $q \in Q$ *is the current state and* $\boldsymbol{\gamma} \in \Gamma^*$ *is the current contents of the stack.*

The initial and final configurations are examples of configurations. It is possible to generalize the initial and final stacks to (say) regular expressions over Γ but the current definition suffices for our purposes. The reason we can do this, as we will see later, is because the stack sequences (strings in Γ^*) of a PDA form a regular language. We will look at this in more detail in a later section. Unless stated otherwise, we will assume that the initial and final configurations are $(q_I, \boldsymbol{\gamma}_I) = (q_I, \varepsilon)$ and $(q_F, \boldsymbol{\gamma}_F) = (q_I, \varepsilon)$ for some $q_I, q_F \in Q$. We will later look at some subclasses of PDAs that have different initial and final configurations.

Recall that a finite-state automaton moves from one state to another by taking transitions from δ. A pushdown automaton behaves similarly, moving from one configuration to another by taking transitions from δ. We represent a transition $\tau = (p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2)$ using the more intuitive notation $\tau = p \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2} q$, which represents a move from state p to state q while popping the sequence of symbols $\boldsymbol{\gamma}_1 \in \Gamma^*$ from the top of the stack and pushing the sequence $\boldsymbol{\gamma}_2 \in \Gamma^*$. This means that whenever the PDA is in a configuration $(p, \boldsymbol{\gamma}\boldsymbol{\gamma}_1)$, it can take the transition τ and end up in the configuration $(q, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$. We use the notation $(p, \boldsymbol{\gamma}\boldsymbol{\gamma}_1) \overset{\tau}{\Longrightarrow} (q, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$ to denote such a move.

[24] This is only true for non-deterministic pushdown automata.

Definition 1.5.4. *Whenever a transition is of the form $\tau = (p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2)$, where $a \in \Sigma$, we call τ* ***scanning****; otherwise, we call in* ***non-scanning****.*

Definition 1.5.5. *We call a transition $\tau = (p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2)$* ***$k$-pop, l-push*** *if $|\boldsymbol{\gamma}_1| = k$ and $|\boldsymbol{\gamma}_2| = l$.*

Definition 1.5.6. *A* ***run*** *of a PDA $\mathfrak{P}$ is a sequence of transitions and configurations*

$$\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \tau_1, (q_1, \boldsymbol{\gamma}_1), \cdots, \tau_n, (q_n, \boldsymbol{\gamma}_n), \tag{1.72}$$

where for $i = 1, \cdots, n$, we have $(q_{i-1}, \boldsymbol{\gamma}_{i-1}) \overset{\tau_i}{\Longrightarrow} (q_i, \boldsymbol{\gamma}_i)$. If $(q_0, \boldsymbol{\gamma}_0) = (q_I, \boldsymbol{\gamma}_I)$ and $(q_n, \boldsymbol{\gamma}_n) = (q_F, \boldsymbol{\gamma}_F)$, $\boldsymbol{\pi}$ is called ***accepting****. If τ_i scans the symbol a_i, then $\boldsymbol{\pi}$ scans the string $\boldsymbol{y} = a_1 \cdots a_n$, which we denote by $\boldsymbol{y} = \mathrm{s}(\boldsymbol{\pi})$. If $\boldsymbol{\pi}$ is accepting, we furthermore say that $\mathfrak{P}$* ***recognizes*** *(or* ***accepts****) the string $\boldsymbol{y}$.*

It will sometimes be more convenient to treat a run as a sequence of configurations only or a sequence of transitions only. Similar to a path in an FSA, the length of a run in a PDA, denoted by $|\boldsymbol{\pi}|$ is the number of transitions it contains.

A pushdown automaton can be graphically represented as a directed multi-graph, as shown in Example 1.5.1. Just like in the FSA case, a graph's nodes represent the states and the edges represent its transitions. However, apart from the input symbol, for PDA we also indicate the symbols popped and pushed from and to the stack.

We define the following notation for the sets of accepting runs of a PDA $\mathfrak{P}$:

- $\Pi(\mathfrak{P})$ represents the set of all accepting runs of $\mathfrak{P}$;
- $\Pi(\boldsymbol{y}, \mathfrak{P})$ represents the set of all accepting runs of $\mathfrak{P}$ that scan $\boldsymbol{y}$;

Definition 1.5.7. *The* ***language*** *recognized by a PDA $\mathfrak{P}$ is the set of strings scanned by all its accepting runs,*

$$L(\mathfrak{P}) = \{\mathrm{s}(\boldsymbol{\pi}) \mid \boldsymbol{\pi} \in \Pi(\mathfrak{P})\}. \tag{1.73}$$

Example 1.5.1. *Fig. 1.14a shows an example of a pushdown automaton $\mathfrak{P}$ accepting the language $L(\mathfrak{P}) = \{a^n b^n \mid n \in \mathbb{N}\}$. $\left(1 \xrightarrow{a,\varepsilon\to \mathrm{X}} 1, 1 \xrightarrow{\varepsilon,\varepsilon\to\varepsilon} 2\right)$ is a run of $\mathfrak{P}$; $\left(1 \xrightarrow{a,\varepsilon\to \mathrm{X}} 1, 1 \xrightarrow{\varepsilon,\varepsilon\to\varepsilon} 2, 2 \xrightarrow{b,\mathrm{X}\to\varepsilon} 2\right)$ is an* accepting *run of $\mathfrak{P}$.*

(a) A PDA $\mathfrak{P}$ that recognizes the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

(b) A PDA $\mathfrak{P}'$ that recognizes the language $\{ww^{\mathrm{R}} \mid w \in \{a, b\}^*\}$.

Figure 1.14: Examples of pushdown automata.

A note on recognition. Traditionally, a PDA is defined with initial and final states rather than initial and final configurations. Our definition is slightly different in order to generalize multiple definitions found in the literature. Other definitions introduce the notions of *recognition by final state*, *recognition by empty stack*, or *recognition by final state and empty stack*. A PDA that recognizes strings by final state has a set of final states and recognizes strings when it ends its computation in a final state, regardless of the contents of the stack. Similarly, a PDA that recognizes strings by an empty stack can end its computation whenever the stack is empty, regardless of the current state. Finally, a PDA that recognizes strings by final state and empty stack can end its computation when it is *both* in a final state and the current stack is empty. Generally, under these definitions, a PDA starts its computation in an initial state and can have on the stack a single initial symbol. Under our definition, a PDA can only start its computation in an initial configuration and stop when it reaches the final configuration. However, one can easily construct transformations that convert such a PDA into other types of PDAs.

1.5.2 Weighted Pushdown Automata

We can now generalize PDAs to their semiring-weighted counterparts.

Definition 1.5.8. *A **weighted pushdown automaton** (WPDA) over a semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is a 6-tuple $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ where*

- *Q is a finite set of states;*
- *Σ is a finite set of input symbols, called the* input alphabet*;*
- *Γ is a finite set of stack symbols, called the* stack alphabet*;*
- *$\delta \colon Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^* \to \mathbb{K}$ is a* transition weighting function*;*
- *$(q_I, \boldsymbol{\gamma}_I)$ and $(q_F, \boldsymbol{\gamma}_F)$, where $q_I, q_F \in Q$, $\boldsymbol{\gamma}_I, \boldsymbol{\gamma}_F \in \Gamma^*$, are called the* initial *and* final configuration*.*

Unlike the initial and final states in a WFSA, the initial and final configurations of a WPDA are *not* weighted. Similar to the unweighted version, we will denote a transition such that $\delta(p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2) = w$ by $p \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2 / w} q$ and we will only show the transitions with non-$\boldsymbol{0}$ weight when representing a WPDA graphically.

Definition 1.5.9. *Let $\mathfrak{P}$ be a WPDA. The **weight** of a run $\boldsymbol{\pi}$ of $\mathfrak{P}$, denoted by $w(\boldsymbol{\pi})$, is the $\otimes$-multiplication of the weights of its transitions,*

$$w(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \bigotimes_{\tau \in \boldsymbol{\pi}} \delta(\tau). \tag{1.74}$$

Definition 1.5.10. *Let $\mathfrak{P}$ be a WPDA and $\boldsymbol{y}$ an input string. The **stringsum** of $\boldsymbol{y}$ in $\mathfrak{P}$, denoted by $w(\mathfrak{P}, \boldsymbol{y})$, is the total weight of the accepting runs scanning $\boldsymbol{y}$,*

$$w(\mathfrak{P}, \boldsymbol{y}) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathfrak{P}, \boldsymbol{y})} w(\boldsymbol{\pi}). \tag{1.75}$$

Definition 1.5.11. *Let $\mathfrak{P}$ be a weighted pushdown automaton. The **language** of $\mathfrak{P}$ is the set of strings $\boldsymbol{y} \in \Sigma^*$ with non-$\boldsymbol{0}$ weight,*

$$L(\mathfrak{P}) = \{\boldsymbol{y} \in \Sigma^* \mid w(\mathfrak{P}, \boldsymbol{y}) > \boldsymbol{0}\}. \tag{1.76}$$

Definition 1.5.12. *Let $\mathfrak{P}$ be a WPDA. The **allsum** of $\mathfrak{P}$, denoted by $w(\mathfrak{P})$, is the total weight of the accepting runs of $\mathfrak{P}$,*

$$w(\mathfrak{P}) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathfrak{P})} w(\boldsymbol{\pi}). \tag{1.77}$$

The allsum of a WPDA is analogous to the treesum of a WCFG or the pathsum of a WFSA.

1.5.3 Subclasses of WPDAs

So far we used a general definition of WPDAs that allows transitions which can pop and push any number of stack symbols. We now introduce two subclasses of WPDAs, called **bottom-up** and **top-down** that appear repeatedly in the literature. The names are suggestive of the fact that these types of WPDAs can be used as bottom-up or top-down parsers for WCFGs. Sometimes it will be more convenient to derive certain constructions or algorithms for these special types of PDAs but they have equivalent expressive power to each other and the WPDA from Def. 1.5.1.

Definition 1.5.13. *A WPDA is called **bottom-up** if it only has 1-push transitions. Moreover, the initial configuration is (q_I, ε) and the final configuration is (q_F, S) for some $q_I, q_F \in Q$ and $S \in \Gamma$.*

In other words, each transition of a bottom-up WPDA pushes *exactly* one stack symbol but it can pop any number of symbols. Additionally, it can only start with an empty stack and end its computation when it places a distinguished stack symbol S on the stack.

Proposition 1.5.1. *Every WPDA is equivalent[25] to some bottom-up WPDA.*

Proof. We assume that $\mathfrak{P}$ is a WPDA as defined in Def. 1.5.1 and construct a bottom-up WPDA $\mathfrak{P}'$ that recognizes the same language. $\mathfrak{P}'$ has the set of states $Q' = Q \cup \{q'_I, q'_F\}$, the set of stack symbols $\Gamma' = \Gamma \cup \{S'\}$, the initial configuration (q'_I, ε) and the final configuration (q'_F, S'). If $\boldsymbol{\gamma}_I = X_1 \cdots X_k$, where $k > 0$, add the new states $r_1, \cdots, r_{k-1}$ to Q and the following transitions to δ':

$$\left\{ \begin{array}{ll} q'_I \xrightarrow{\varepsilon, \varepsilon \to S'/\mathbf{1}} r_1, & \\ r_{i-1} \xrightarrow{\varepsilon, \varepsilon \to X_i/\mathbf{1}} r_i, & i = 1, \cdots, k-1 \\ r_{k-1} \xrightarrow{\varepsilon, \varepsilon \to X_k/\mathbf{1}} q_I, & \\ q_F \xrightarrow{\varepsilon, S'\boldsymbol{\gamma}_F \to S'/\mathbf{1}} q'_F. & \end{array} \right\} \tag{1.78}$$

The first transition places the new symbol S' on the stack. This symbol will always be at the bottom of the stack so that we can modify k-pop, 0-push transitions that pop the whole stack into $k+1$-pop, 1-push transitions. The second and third types of transitions place sequentially each symbol from the old initial stack and the last transition replaces the old final stack with S'. For each k-pop, 1-push transition $p \xrightarrow{a, \boldsymbol{\gamma} \to X_1 \cdots X_l / w} q \in \delta$ such that $l > 1$, we add $l - 1$ new states $r_1, \cdots, r_{l-1}$ to Q and the following transitions to δ':

$$\left\{ \begin{array}{ll} p \xrightarrow{\varepsilon, \boldsymbol{\gamma} \to X_1/\mathbf{1}} r_1, & \\ r_{i-1} \xrightarrow{\varepsilon, \varepsilon \to X_i/\mathbf{1}} r_i, & i = 2, \cdots, l-1 \\ r_{k-1} \xrightarrow{\varepsilon, \varepsilon \to X_k/\mathbf{1}} q. & \end{array} \right\} \tag{1.79}$$

[25] We will see later in the chapter that there exist multiple definitions of equivalence between WPDAs. For now, we will assume that two machines are equivalent if they recognize the same weighted language. We will see later that this transformation is actually semantics preserving, meaning that every derivation in the original machine is preserved in the output machine, with the same weight. This further implies that the languages of the two machines are identical.

For each k-pop, 0-push transition $p \xrightarrow{a,\boldsymbol{\gamma}\to\varepsilon/w} q \in \delta$, add to δ' the transition $p \xrightarrow{a,X\boldsymbol{\gamma}\to X/w} q$ for all $X \in \Gamma'$. □

The mirror image of a bottom-up WPDA is a top-down WPDA, whose transitions pop exactly one stack symbol.

Definition 1.5.14. *A WPDA is called **top-down** if it only has 1-pop transitions. Moreover, the initial configuration is (q_I, S) and the final configuration is (q_F, ε) for some $q_I, q_F \in Q$ and $S \in \Gamma$.*

Proposition 1.5.2. *Every WPDA is equivalent to some top-down WPDA.*

The proof follows closely the proof for the bottom-up case. We leave it as an exercise.

Definition 1.5.15 (Push computation)**.** *Let $\mathfrak{P}$ be a bottom-up WPDA. A **push computation** of $\mathfrak{P}$ that pushes X is a run $\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \cdots, (q_m, \boldsymbol{\gamma}_m)$, where $\boldsymbol{\gamma}_m = \boldsymbol{\gamma}_0 X$, and for all $l > 0$, $|\boldsymbol{\gamma}_l| \geq |\boldsymbol{\gamma}_m|$.*

Intuitively, a push computation that pushes X is a run $\boldsymbol{\pi}$ that places X at the top of the stack without touching the sequence stack symbols below, as shown in Fig. 1.15. Each other symbol that is pushed to the stack between q_0 and q_m must be removed later by another transition. If the last transition of $\boldsymbol{\pi}$ pops $Y_1 Y_2 \cdots Y_k$ and pushes X, it means that the symbols $Y_1 Y_2 \cdots Y_k$ were placed on the stack by some other push computations. Therefore, each push computation consists of k shorter push computations and a final k-pop, 1-push transition.

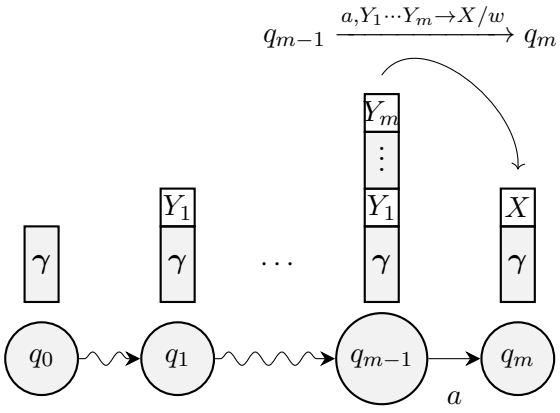


Figure 1.15: Example of push computation that pushes X to the stack. The curly edges indicate sequences of transitions while the straight edge indicates a single transition.

The mirror image of a push computation is a pop computation in top-down WPDAs. The pop computations have the same structure as push computations but the transitions happen in reverse order. A pop computation always consists of a 1-pop, k-push transition, followed by k pop computations.

Definition 1.5.16 (Pop computation)**.** *Let $\mathfrak{P}$ be a top-down WPDA. A **pop computation** of $\mathfrak{P}$ that pops X is a run $\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \cdots, (q_m, \boldsymbol{\gamma}_m)$, where $\boldsymbol{\gamma}_0 = \boldsymbol{\gamma}_m X$, and for all $l > 0$, $|\boldsymbol{\gamma}_l| \leq |\boldsymbol{\gamma}_m|$.*

1.5.4 PDA-CFG Conversions

We have already mentioned in this chapter that CFGs and PDAs have equivalent expressive power, both being formalisms for characterizing context-free languages. In this section we will have a

further look at this equivalence in expressive power of CFGs and PDAs. We can show that for any top-down or bottom-up WPDA, we can construct a WCFG that generates the same language. We will give algorithms for top-down and bottom-up WPDAs only, but, as we've seen earlier, any WPDA can be converted into such a form. The pop and push computations are essential in the construction of these algorithms.

Theorem 1.5.1. *A language L is context-free if and only if there exists a pushdown automaton that recognizes it.*

The first direction of the proof (i.e., if a language L is context-free, there exists a non-deterministic PDA that recognizes it) will be discussed in §1.5.7. The second part can be proven by construction, that is, by converting a PDA to an equivalent CFG.

When converting a top-down PDA into an equivalent grammar, the non-terminals will correspond to pop computations. Therefore, the non-terminals will be of the form $[pXq]$, where $p, q \in Q$ and $X \in \Gamma$. Since all accepting runs of a top-down PDA are pop computations of S from q_I to q_F, the distinguished start symbol will be $[q_I S q_F]$. Similarly, when converting a bottom-up PDA to a CFG, the non-terminals of the grammar will be of the same form but correspond to push computations. Alg. 13 and Alg. 14 formalize this intuition for the top-down and bottom-up cases, respectively.

Algorithm 13 Top-down PDA-to-CFG conversion. We assume that the input WPDA is top-down.

def top-down-PDA-to-CFG($\mathfrak{P}$):
  $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ ▷*Initialize CFG $\mathfrak{G}$ over the same semiring and input alphabet.*
  $\mathcal{N} \leftarrow \{[qXp] \mid q, p \in Q, X \in \Gamma\}$ ▷*Non-terminals correspond to pop computations.*
  $\mathrm{S} \leftarrow [q_I S q_F]$
  for $p \xrightarrow{a, X \to \gamma / w} q \in \delta$:
    $k \leftarrow |\gamma|$
    if $k = 0$:
      add $[pXq] \to a$ to $\mathcal{P}$ with weight w
    else
      $Y_1 \dots Y_k \leftarrow \gamma$
      for $r_1, \cdots, r_k \in Q$: ▷*All possible combinations of states $r_1, \cdots, r_k$.*
        add $[pXr_k] \to a[qY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$ to $\mathcal{P}$ with weight w
  return $\mathfrak{G}$

Example 1.5.2. *Consider the unweighted top-down PDA $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q, S), (q, \varepsilon))$ where $Q = \{q\}$, $\Sigma = \{a, b\}$, $\Gamma = \{S, X, Y\}$, and δ has the following transitions:*

$$q \xrightarrow{a, X \to \varepsilon} q, q \xrightarrow{b, Y \to \varepsilon} q, q \xrightarrow{\varepsilon, S \to \varepsilon} q$$
$$q \xrightarrow{a, S \to SY} q, q \xrightarrow{b, S \to SX} q, q \xrightarrow{a, Y \to YY} q, q \xrightarrow{b, X \to XX} q$$

The corresponding context-free grammar $\mathfrak{G}$ has the alphabet $\Sigma = \{a, b\}$, the set of non-terminals

$\mathcal{N} = \{S, [qSq], [qXq], [qYq]\}$, *the start symbol* $[qSq]$ *and the following set of production rules:*

$$\mathcal{P} = \left\{ \begin{array}{c} [qSq] \to b[qXq][qSq], \\ [qSq] \to a[qYq][qSq], \\ [qXq] \to b[qXq][qXq], \\ [qYq] \to a[qYq][qYq], \\ [qXq] \to a, \\ [qYq] \to b, \\ [qSq] \to \varepsilon \end{array} \right\}$$

In order to prove the correctness of the algorithm, we will prove that the interpretation of the non-terminals in the constructed CFG is correct.

Lemma 1.5.1. *Let* $\mathfrak{P}$ *be a top-down WPDA and* $\mathfrak{G}$ *the WCFG outputted by Alg. 13. The non-terminal* $[pXq]$ *of* $\mathfrak{G}$ *derives the string* $\boldsymbol{y}$ *with weight* w *if and only if* $\mathfrak{P}$ *has a pop computation of* X *from state* p *to state* q *that scans* $\boldsymbol{y}$ *with weight* w.

Proof. ($\Rightarrow$) Suppose there exists a pop computation of $\mathfrak{P}$ that pops X from p to q, which we call a pop computation of type $[pXq]$, and scans $\boldsymbol{y}$ with weight w. We prove that the non-terminal $[pXq]$ derives $\boldsymbol{y}$ with the weight w by induction on the length of the pop computation.

Base Case. When the pop computation is of length 1, it must consist of a single 1-pop, 0-push transition $p \xrightarrow{a, X \to \varepsilon / w} q$. Therefore, $\mathcal{P}$ will contain the production $[pXq] \to a$ with weight w.

Inductive Step. We assume that $\mathfrak{P}$ has a pop computation of X scanning $\boldsymbol{y}$ from p to q with weight w that has length ℓ. Additionally, we assume that the statement holds for all pop computations of length at most $\ell - 1$. The first transition of the pop computation must be of the form $p \xrightarrow{a, X \to Y_1 \cdots Y_k / w'} r_1$, followed by $\ell - 1$ pop computations of type $[r_1 Y_1 r_2]$, $[r_2 Y_2 r_3]$, $\cdots$ and $[r_{k-1} Y_k q]$, respectively. All these pop computations must have length at most $\ell - 1$, thus the inductive hypothesis applies to them. We assume that these pop computations scan the substrings $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_k$ and have weights $w_1, w_2, \cdots, w_k$. Then the non-terminals $[r_1 Y_1 r_2]$, $[r_2 Y_2 r_3]$, $\cdots$ and $[r_{k-1} Y_k q]$ derive the substrings $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_k$ with weights $w_1, w_2, \cdots, w_k$. The push computation of type $[pXq]$ scans the string $\boldsymbol{y} = a \circ \boldsymbol{y}_1 \circ \boldsymbol{y}_2 \circ \cdots \circ \boldsymbol{y}_k$ and has weight $w = w' \otimes w_1 \otimes w_2 \otimes \cdots \otimes w_k$. It is easy to notice that the non-terminal $[pXq]$ derives the string $\boldsymbol{y}$ with weight w.

($\Leftarrow$) Suppose that the non-terminal $[pXq]$ derives the string $\boldsymbol{y}$ with weight w. Then $\mathfrak{P}$ has a pop computation of type $[pXq]$ that scans $\boldsymbol{y}$ and has weight w. We prove the statement by induction on the length of the derivation.

Base Case. When the derivation has a length of 1, it consists of a single production $[pXq] \to a$ with weight w. Therefore, $\mathfrak{P}$ has the push computation consisting of the single transition $p \xrightarrow{a, X \to \varepsilon / w} q$.

Inductive Step. We assume that the statement holds for any derivation of length at most $\ell - 1$ and there is a derivation d of length ℓ of the string $\boldsymbol{y}$ starting from the non-terminal $[pXq]$ with weight w. By construction, the first transition of this derivation must be of the form $[pXq] \to a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kq]$ and have some weight w'. This production was added by Alg. 13 because $\mathfrak{P}$ has the transition $p \xrightarrow{a, X \to Y_1 \cdots Y_k / w'} r$. We assume that each of the non-terminals derives the substring $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_k$ with the weight $w_1, w_2, \cdots, w_k$, thus d derives the string $\boldsymbol{y} = a \circ \boldsymbol{y}_1 \circ \boldsymbol{y}_2 \circ \cdots \circ \boldsymbol{y}_k$ with the weight $w = w' \otimes w_1 \otimes w_2 \otimes \cdots \otimes w_k$. By the inductive hypothesis, $[rY_1r_1], [r_1Y_2r_2], \cdots, [r_{k-1}Y_kq]$ are pop computations of $\mathfrak{P}$ that scan $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_k$ and have weights $w_1, w_2, \cdots, w_k$. Therefore, $\mathfrak{P}$ has a pop computation of type $[pXq]$ that scans $\boldsymbol{y}$ and has weight w. □

In a top-down WPDA all accepting runs are pop computations of S from q_I to q_F. Thus, by the previous lemma, all the strings scanned by these runs will be derived by the distinguished start symbol $[q_I S q_F]$ with the same weight.

Since the mirror image of a top-down WPDA is a bottom-up WPDA, the previous algorithm can be modified slightly in order to allow the conversion of bottom-up WPDAs into WCFGs (Alg. 14). We leave the correctness of this algorithm as an exercise.

Algorithm 14 Bottom-up PDA-to-CFG conversion. We assume that the input WPDA is bottom-up.

```
1.  def bottom-up-PDA-to-CFG(𝔓):
2.    𝔊 = (𝒩, Σ, S, 𝒫)                ▷Initialize CFG 𝔊 over the same semiring and input alphabet.
3.    𝒩 ← {[qXp] | q, p ∈ Q, X ∈ Γ}    ▷Non-terminals correspond to push computations.
4.    S ← [q_I S q_F]
5.    for p --a,γ→X/w--> q ∈ δ :
6.      k ← |γ|
7.      if k = 0 :
8.        add [pXq] → a to 𝒫 with weight w
9.      else
10.       Y_1 ... Y_k ← γ
11.       for r_1, ··· , r_k ∈ Q :   ▷All possible combinations of states r_1, ··· , r_k.
12.         add [r_1 X q] → [r_1 Y_1 r_2] ··· [r_{k-2} Y_2 r_{k-1}][r_{k-1} Y_k p]a to 𝒫 with weight w
13.   return 𝔊
```

Lemma 1.5.2. *Let $\mathfrak{P}$ be a bottom-up WPDA and $\mathfrak{G}$ the WCFG outputted by Alg. 14. The non-terminal $[pXq]$ of $\mathfrak{G}$ derives the string $\boldsymbol{y}$ with weight w if and only if $\mathfrak{P}$ has a push computation of X from state p to state q that scans $\boldsymbol{y}$ with weight w.*

Proof. The proof is left as an exercise. □

1.5.5 Determinism

Just like in the case of finite-state automata, PDAs can also be deterministic or non-deterministic. However, there is an important difference in the case of pushdown automata. While deterministic and non-deterministic FSAs have equivalence expressive power and one can determinize any non-deterministic FSA, this doesn't hold for PDAs. The class of languages recognized by deterministic PDAs is called the class of **deterministic context-free languages** and is a proper subset of the class of context-free languages.

Definition 1.5.17. *A PDA $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ is called **deterministic** if*

- *For every $(q, a, \boldsymbol{\gamma}) \in Q \times \Sigma \cup \{\varepsilon\} \times \Gamma^*$, there is at most one transition $(q, a, \boldsymbol{\gamma}, p, \boldsymbol{\gamma}') \in \delta$;*
- *If there is a transition $(q, a, \boldsymbol{\gamma}, p, \boldsymbol{\gamma}') \in \delta$ for some $a \in \Sigma$, there is no transition of the type $(q, \varepsilon, \boldsymbol{\gamma}, p', \boldsymbol{\gamma}'')$.*

Theorem 1.5.2. *A language $L \subseteq \Sigma^*$ is context-free if and only if it can be recognized by a non-deterministic pushdown automaton.*

We give this theorem without proof for now but we will come back to it in a later section and explore in detail the relation between context-free grammars (which, as we saw already, generate context-free languages) and pushdown automata.

Example 1.5.3. *The context-free language $L = \{ww^R \mid w \in \{a,b\}^*\}$ cannot be recognized by any deterministic pushdown automaton. However, there exist non-deterministic PDAs that can recognize it. One example is the PDA shown in Fig. 1.14b. It is not deterministic because when it is in state* 1*, it can either read the next symbol and push a symbol to the stack or move to state* 2*, without reading any symbol or modifying the stack. Therefore, it needs to "guess" when it has reached the center of the string. If the words in L contained a center marker, then they could be recognized by a deterministic PDA that pushes symbols to the stack until it scans the center of the string, then pops the stack symbols until it empties the stack. The automaton shown in Fig. 1.16 recognizes the language $L' = \{wcw^R \mid w \in \{a,b\}^*, c \in \Sigma\}$.*

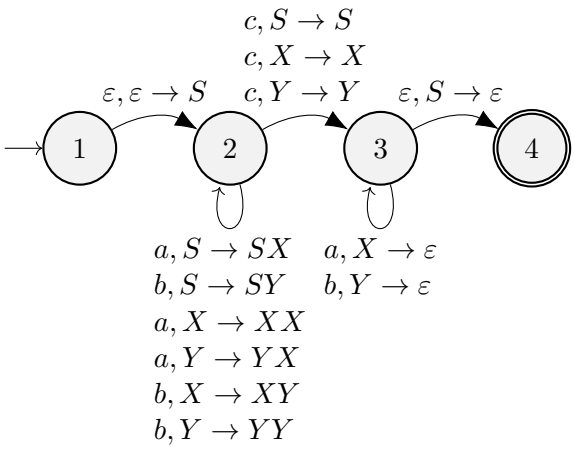


Figure 1.16: A deterministic PDA $\mathfrak{P}'$ that recognizes the language $\{wcw^{\mathrm{R}} \mid w \in \{a,b\}^*\}$.

The languages that deterministic PDAs accept have *unambiguous* grammars. However, the converse of the statement does not hold, i.e., the languages recognized by deterministic PDAs are not exactly equal to the subset of the context-free languages that are not inherently ambiguous.

Example 1.5.4. *For instance, the language $L = \{ww^R \mid w \in \{a,b\}^*\}$ has the unambiguous grammar $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$, where $\Sigma = \{a,b\}$, $\mathcal{N} = \{S\}$, $\mathrm{S} = S$ and*

$$\mathcal{P} = \left\{ \begin{array}{c} S \to aSa \\ S \to bSb \\ S \to \varepsilon \end{array} \right\},$$

even though it is not a deterministic context-free language.

Theorem 1.5.3. *If a language L is deterministic context-free, it has an unambiguous context-free grammar.*

Proof. We will prove that the context-free grammar outputted by Alg. 13 is unambigous when the input PDA is deterministic. Recall that it suffices to prove that a context-free grammar has unique left-most derivations in order to prove that it is unambiguous.

Assume that L is the language recognized by a deterministic PDA $\mathfrak{P}$ and it has a grammar $\mathfrak{G} = \mathsf{top\text{-}down\text{-}PDA\text{-}to\text{-}CFG}(\mathfrak{P})$. We assume without loss of generality that $\mathfrak{P}$ is a top-down PDA. When $\mathfrak{P}$ accepts a string $\boldsymbol{y}$, it does so by following a unique sequence of moves since it is deterministic. Knowing this sequence of moves, we can determine the single choice of a production in a left-most derivation whenever $\mathfrak{G}$ derives $\boldsymbol{y}$. Even though a transition $p \xrightarrow{a, X \to Y_1 \cdots Y_k / w} q$ of $\mathfrak{P}$ might generate

multiple production rules in $\mathfrak{G}$ in Alg. 13 (with different states in the positions that reflect the states of $\mathfrak{P}$ after popping each of $Y_1, \cdots, Y_k$), only one of these productions will be consistent with the transitions of $\mathfrak{P}$, therefore only one of these productions will lead to a valid derivation of $\boldsymbol{y}$. □

1.5.6 Stack Language & PPDAs

Recall that the stack of a pushdown automaton has infinite size, thus the *stack language*, i.e., the set of strings that is written on the stack at any point in the computation, is also infinite. This further implies that the number of configurations of a PDA is infinite. However, it turns out that the stack language of a pushdown automaton is *regular*. As we will see later in this section, this fact is essential for defining probabilistic pushdown automata.

Theorem 1.5.4. *The stack language of a pushdown automaton is a* ***regular language****.*

We give for simplicity a proof in the unweighted case but a very similar proof holds for the weighted case. Autebert et al. (1997) give an alternative proof for top-down PDAs by constructing left-linear and right-linear CFGs, which we already know that generate regular languages. Here, we make use again of the push computations.

Proof. We assume that $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \varepsilon), (q_F, S))$ is an unweighted bottom-up PDA. Then we can define an FSA $\mathcal{A} = (\Gamma, Q, \{q_I\}, Q, \delta')$ where δ' contains a transition $p \xrightarrow{X} q$ if and only if there is a push computation of X from p to q in $\mathfrak{P}$. In words, the automaton $\mathcal{A}$ is defined over the stack alphabet of $\mathfrak{P}$ and has the same set of states. The single initial state of $\mathcal{A}$ is $\mathfrak{P}$'s initial state q_I and all states can be final.

At each point in the computation of $\mathfrak{P}$, the stack contains a sequence of symbols $\boldsymbol{\gamma} = X_1 X_2 \cdots X_k$. Each X_i has been pushed as a result of a push computation, thus by construction $\mathcal{A}$ has a transition labeled with X_i. Since all states are accepting and $\mathfrak{P}$ starts its computation with an empty stack, $\mathcal{A}$ must have an accepting path yielding $\boldsymbol{\gamma}$.

□

When the PDA is top-down we can give a mirror image of the proof. The transitions of the automaton $\mathcal{A}$ must correspond to pop computations of $\mathfrak{P}$ and an accepting path of $\mathcal{A}$ corresponds to emptying the stack $\boldsymbol{\gamma}$. It is easy to see that all states must be initial and $\mathcal{A}$ has a single final state q_F ($\mathfrak{P}$'s final state). We leave the details as an exercise.

Now that we have determined what class of languages the stack language belongs to, we are ready to define an important type of semiring-weighted PDAs, namely probabilistic PDAs (or PPDAs). As the name suggests, the weights of a PPDA must form a probability distribution. Naturally, in the case of probabilistic FSAs, the weights of outgoing edges of any state must form a probability distribution. Similarly, in the case of probabilistic CFGs, the weights of the productions having some non-terminal X of the left-hand side must form a probability distribution. However, in the case of probabilistic PDAs, there are some subtleties involved. An intuitive way of defining a probability distribution would be over the next possible actions for a given configuration. Recall however, that a PDA can have an infinite number of possible configurations. The transition function δ, however, is still finite. Therefore, there is a finite number of actions we can ever do. When defining a probabilistic PDA, we must limit ourselves to a finite number of configurations over which we define probability distributions over the next possible actions. This is where the regularity of the stack language comes into play. Recall that a regular language has a finite number of equivalence classes. Therefore, a very natural way to limit ourselves to a finite set of contexts for defining probability distributions in a probabilistic PDA is in terms of a general equivalence relation.

Definition 1.5.18. *Let R be an equivalence relation over Γ^* with a finite number of equivalence classes and $\{[\boldsymbol{\gamma}] \mid \boldsymbol{\gamma} \in \Gamma^*\}$ the set of its equivalence classes. A WPDA $\mathfrak{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ is called **probabilistic** if for any configuration $(q, \boldsymbol{\gamma})$ it holds that*

$$\forall q \xrightarrow{a,[\boldsymbol{\gamma}]\to[\boldsymbol{\gamma}']/w} r \in \delta \colon w \geq 0 \tag{1.80}$$

and

$$\sum_{q \xrightarrow{a,[\boldsymbol{\gamma}]\to[\boldsymbol{\gamma}']/w} r} w = 1. \tag{1.81}$$

1.5.7 Shift-reduce parsing

One of the most prominent use cases of pushdown automata—both in NLP and in compiler theory—is the construction of linear-time parsing algorithms. Recall from our discussion on parsing that the exact parsing, e.g. using the CKY algorithm, generally have a runtime of $\mathcal{O}\left(n^3\right)$ where n is the length of the input to be parsed. PDAs allow us to construct linear-time parsing algorithm by scanning the input string in one pass over the text, without backtracking. When the language is deterministic context-free, we can parse the grammar exactly in linear time. However, even for context-free languages that are not deterministic, it still beneficial to consider linear-time parsers. Even though the parsing algorithm will not run in linear time, in practice it will run in a time that is close to linear as the runtime is proportional to the degree of non-determinism in the grammar. The general paradigm we will discuss is called *shift–reduce parsing*. This involves creating a special type of PDA from the CFG, in which there is only one state and there are only two types of transitions: SHIFT and REDUCE. We denote this automaton as the **Shift-Reduce WPDA**. There are two general conversion strategies: the **bottom-up** shift–reduce strategy and the **top-down** shift–reduce strategy. As the name suggests, the bottom-up strategy constructs a bottom-up WPDA while the top-down strategy constructs a top-down WPDA.

Bottom-up Shift-Reduce Parsing

The general idea of the bottom-up shift–reduce parsing algorithm is to reduce the input string back to the start symbol S. The parser builds up the parse tree incrementally, bottom-up, left-to-right, without guessing of backtracking. At every point in this pass, the parser has accumulated a list of subtrees that have already been parsed, which will eventually be merged using some rules of the grammar. As mentioned earlier the parser works by iteratively performing two types of actions, a SHIFT or a REDUCE.

- A SHIFT transition reads in the nnext input symbol and places a single symbol at the top of the stack. This symbol corresponds to a new single-node parse tree. For example, a transition SHIFT$(X \to x)$ scans x and pushes X onto the stack.
- REDUCE are ε-transitions that combine several stack symbols by replacing the right-hand side of production by its left-hand side. This step corresponds to merging several subtrees of the derivation tree into a single one, rooted at the most recent symbol pushed on the stack, i.e., the left-hand side of the production. For example, REDUCE$(X \to YZ)$ pops YZ from the stack and pushes X.

The parser applies successively shift and reduce transitions until the whole input has been consumed, at which point the full parse tree has been constructed. It turns out that applying transitions of this form corresponds to visiting the nodes of the derivation tree in post order.

Algorithm 15 Bottom-up Shift–Reduce Parsing.

```
1. def bottom-up-shift-reduce(𝔊):
2.     ▷ Initialize single-state bottom-up WPDA over the same semiring. The non-terminals become the stack symbols.
3.     𝔓 = ({q}, Σ, 𝒩, (q, ε), (q, S))
4.     for (X --w--> α) ∈ 𝒫 :
5.         if α ∈ Σ :                                   ▷translate X → x
6.             add q --α,ε→X/w--> q into δ
7.         else                                         ▷translate X → Y_1 Y_2 ⋯ Y_k
8.             add q --ε,α→X/w--> q into δ
9.     return 𝔓
```

Example 1.5.5. *Consider the WCFG $\mathfrak{G} = (\mathcal{N}, \Sigma, \mathrm{S}, \mathcal{P})$ in Chomsky Normal Form, where $\mathcal{N} =$ {Adj, Det, N, S, NP}, $\Sigma =$ {a, big, female, giraffe, male, tall, the}, $\mathrm{S} = S$ and the production rules $\mathcal{P}$ are defined in Tab. 1.1. We can translate each rule in $\mathfrak{G}$ to a transition in a WPDA $\mathfrak{P}$.*

Rules in $\mathfrak{G}$	Transitions in $\mathfrak{P}$	
	bottom-up-shift-reduce	top-down-shift-reduce
$\mathrm{S} \xrightarrow{1} \mathrm{Det\ NP}$	$q \xrightarrow{\varepsilon,\mathrm{Det\ NP}\to\mathrm{S}/1} q$	$q \xrightarrow{\varepsilon,\mathrm{S}\to\mathrm{Det\ NP}/1} q$
$\mathrm{NP} \xrightarrow{.4} \mathrm{N}$	$q \xrightarrow{\varepsilon,\mathrm{N}\to\mathrm{NP}/.4} q$	$q \xrightarrow{\varepsilon,\mathrm{NP}\to\mathrm{N}/.4} q$
$\mathrm{NP} \xrightarrow{.6} \mathrm{Adj\ NP}$	$q \xrightarrow{\varepsilon,\mathrm{Adj\ N}\to\mathrm{NP}/.6} q$	$q \xrightarrow{\varepsilon,\mathrm{NP}\to\mathrm{Adj\ N}/.6} q$
$\mathrm{Det} \xrightarrow{.5} \mathrm{a}$	$q \xrightarrow{\mathrm{a},\varepsilon\to\mathrm{Det}/.5} q$	$q \xrightarrow{\mathrm{a},\mathrm{Det}\to\varepsilon/.5} q$
$\mathrm{Det} \xrightarrow{.5} \mathrm{the}$	$q \xrightarrow{\mathrm{the},\varepsilon\to\mathrm{Det}/.5} q$	$q \xrightarrow{\mathrm{the},\mathrm{Det}\to\varepsilon/.5} q$
$\mathrm{N} \xrightarrow{.2} \mathrm{female}$	$q \xrightarrow{\mathrm{female},\varepsilon\to\mathrm{N}/.2} q$	$q \xrightarrow{\mathrm{female},\mathrm{N}\to\varepsilon/.2} q$
$\mathrm{N} \xrightarrow{.5} \mathrm{giraffe}$	$q \xrightarrow{\mathrm{giraffe},\varepsilon\to\mathrm{N}/.5} q$	$q \xrightarrow{\mathrm{giraffe},\mathrm{N}\to\varepsilon/.5} q$
$\mathrm{N} \xrightarrow{.3} \mathrm{male}$	$q \xrightarrow{\mathrm{male},\varepsilon\to\mathrm{N}/.3} q$	$q \xrightarrow{\mathrm{male},\mathrm{N}\to\varepsilon/.3} q$
$\mathrm{Adj} \xrightarrow{.2} \mathrm{big}$	$q \xrightarrow{\mathrm{big},\varepsilon\to\mathrm{Adj}/.2} q$	$q \xrightarrow{\mathrm{big},\mathrm{Adj}\to\varepsilon/.2} q$
$\mathrm{Adj} \xrightarrow{.3} \mathrm{female}$	$q \xrightarrow{\mathrm{female},\varepsilon\to\mathrm{Adj}/.3} q$	$q \xrightarrow{\mathrm{female},\mathrm{Adj}\to\varepsilon/.3} q$
$\mathrm{Adj} \xrightarrow{.4} \mathrm{male}$	$q \xrightarrow{\mathrm{male},\varepsilon\to\mathrm{Adj}/.4} q$	$q \xrightarrow{\mathrm{male},\mathrm{Adj}\to\varepsilon/.4} q$
$\mathrm{Adj} \xrightarrow{.1} \mathrm{tall}$	$q \xrightarrow{\mathrm{tall},\varepsilon\to\mathrm{Adj}/.1} q$	$q \xrightarrow{\mathrm{tall},\mathrm{Adj}\to\varepsilon/.1} q$

Table 1.1: Production rules of a WCFG $\mathfrak{G}$ in CNF together with the transitions of of the WPDAs resulting from bottom-up and top-down shift-reduce parsing.

Theorem 1.5.5. *Let $\mathfrak{G}$ be a grammar in CNF.*[26] *Then, the derivations $\mathcal{D}_{\mathfrak{G}}$ of $\mathfrak{G}$ and the runs $\Pi(\mathfrak{P})$ of $\mathfrak{P} =$ bottom-up-shift-reduce$(\mathfrak{G})$ are in one-to-one correspondence. Moreover, the weights are preserved.*

Proof. ($\Rightarrow$) Suppose that $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}$ is a derivation tree of $\mathfrak{G}$. Since $\mathfrak{G}$ is in CNF, all productions in $\boldsymbol{t}$, have the form $X \to Y\ Z$ or $X \to x$ where $X, Y, Z \in \mathcal{N}$ and $x \in \Sigma$. The tree $\boldsymbol{t}$ may be converted

[26] This assumption is not strictly necessary, but it makes the proof considerably simpler.

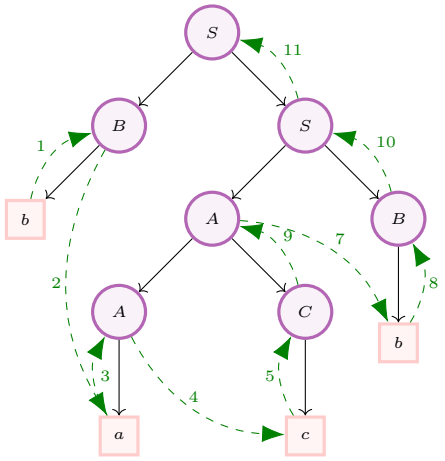
into a run of $\mathfrak{P}$ as follows: when traversing the nodes of $\boldsymbol{t}$ in post order, if the node has two other non-terminals as children, then it corresponds to a REDUCE$(X \to Y\ Z)$ transition; otherwise, it corresponds to a SHIFT$(X \to x)$ transition. ($\Leftarrow$)The proof for the other direction is trivial. □

Example 1.5.6. *Consider the grammar $\mathfrak{G}$ in CNF, where $\mathcal{N} = \{S, A, B, C\}$, $\Sigma = \{a, b, c\}$, $\mathrm{S} = S$ and the set $\mathcal{P}$ contains the following production rules:*

$$\mathcal{P} = \left\{ \begin{array}{l} S \to BS \mid AB \\ A \to AC \mid a \\ B \to b \\ C \to c \end{array} \right\} \tag{1.82}$$

Fig. 1.18a shows a derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$ of the string $\boldsymbol{y} = bacb$. The post-order traversal of the nodes of $\boldsymbol{t}$ is BACABSS, which is exactly the order in which the nodes are added to the stack. Fig. 1.17b shows the corresponding run in $\boldsymbol{\pi} \in \Pi(\mathfrak{P})$, where $\mathfrak{P} =$ bottom-up-shift-reduce$(\mathfrak{G})$.

(a) The derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$. The arrows denote its post-order traversal.



(b) Bottom-up Shift-Reduce Parsing. $\top$ denotes an empty stack.

Stack	Input	Transition
$\top$	bacb	-
B	acb	SHIFT(B →b)
BA	cb	SHIFT(A →a)
BAC	b	SHIFT(C →c)
BA	b	REDUCE(A →AC)
BAB	ε	SHIFT(B →b)
BS	ε	REDUCE(S →AB)
S	ε	REDUCE(S →BS)

Figure 1.17: Derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}$ of the string *bacb*, together with the post-order traversal of the nodes and the corresponding transitions of the shift–reduce WPDA .

Top-Down Shift–Reduce Parsing

In top-down shift–reduce parsing, we start with the start symbol S and derive the input string by systematic application of the productions. In top-down shift-reduce parsing, the SHIFT and REDUCE actions have opposite definitions to those in bottom-up parsing. A SHIFT action scans a terminal symbol and pops a single stack symbol from the stack. For instance, SHIFT$(X \to x)$ scans x and pops X from the stack. A transition of the form REDUCE$(X \to \boldsymbol{\gamma})$ replaces the left-hand symbol X with the sequence of symbols $\boldsymbol{\gamma}$ on the right-hand side. The general converting strategy is as follows: when encountering a terminal symbol, see if the next input matches the stack top (SHIFT); when encountering a non-terminal symbol, match the stack top to a rule in $\mathcal{P}$, and pop stack and push RHS of the rule onto the stack (REDUCE). The formal presentation of the algorithm is given Alg. 16. Perhaps not surprisingly, a top-down shift–reduce parser traverses the nodes in pre-order.

Algorithm 16 Top-down Shift–Reduce Parsing.

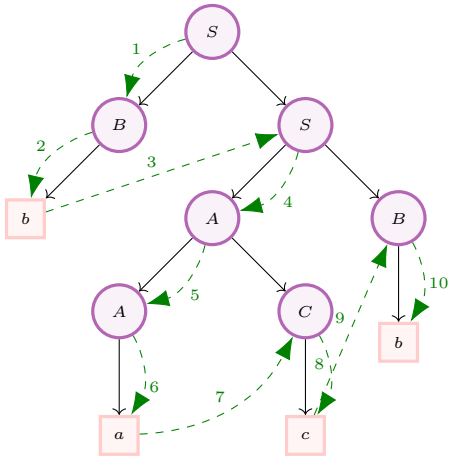
```
1. def top-down-shift-reduce(𝔊):
2.     ▷ Initialize single-state top-down WPDA over the same semiring. The non-terminals become the stack symbols of the WPDA.
3.     𝔓 = ({q}, Σ, 𝒩, (q, S), (q, ε))
4.     for (X --w--> α) ∈ 𝒫 :
5.         if α ∈ Σ :                                   ▷ Translate X → x
6.             add q --α, X→ε/w--> q into δ
7.         else                                         ▷ Translate X → Y_1 Y_2 ⋯ Y_k
8.             add q --ε, X→α/w--> q into δ
9.     return 𝔓
```

Theorem 1.5.6. *Let $\mathfrak{G}$ be a grammar in CNF.*[27] *Then, the derivations $\mathcal{D}_{\mathfrak{G}}$ of $\mathfrak{G}$ and the runs $\Pi(\mathfrak{P})$ of $\mathfrak{P} = \textit{top-down-shift-reduce}(\mathfrak{G})$ are in one-to-one correspondence. Moreover, the weights are preserved.*

Proof. The proof is very similar to the proof for the bottom-up case. We leave it as an exercise. □

Example 1.5.7. *Consider the same grammar $\mathfrak{G}$ and the same derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$ from Example 1.5.6. The WPDA begins with the start symbol* S *and guesses substitutions. When done, it compares the stack top with the next input. In this example, we always perform an action on the left-most non-terminal, which would correspond to a left-most derivation in $\mathfrak{G}$.*

(a) The derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}(\boldsymbol{y})$. The arrows denote its pre-order traversal.



(b) Top-down Shift-Reduce Parsing. ⊤ denotes an empty stack.

Stack	Input	Transition
S	bacb	REDUCE(S →BS)
SB	bacb	SHIFT(B →b)
S	bacb	REDUCE(S →AB)
BA	acb	REDUCE(A →AC)
BCA	acb	SHIFT(A →a)
BC	cb	SHIFT(C →c)
B	b	SHIFT(B →b)
⊤	ε	-

Figure 1.18: Derivation tree $\boldsymbol{t} \in \mathcal{D}_{\mathfrak{G}}$ of the string *bacb*, together with the pre-order traversal of the nodes and the corresponding transitions of the shift–reduce WPDA .

1.5.8 Computing Allsums

In this section, we will derive a system of non-linear equations for computing allsums in bottom-up and top-down WPDAs, similar to the way we computed treesums in WCFGs. In §1.5.3 we saw that

[27] This assumption is not strictly necessary, but it makes the proof considerably simpler.

every push computation consists of a sequence of k push computations of smaller length, followed by a k-pop, 1-push transition. This means that the weight of push computations can be derived solely from the weights of these shorter push computations and the weight of the final transition. Recall that in a bottom-up WPDA the initial configuration is (q_I, ε) and the final configuration is (q_F, S). Therefore, all accepting paths will be push computations of S from q_I to q_F.

$$w(p, X, q) = \bigoplus_{a \in \Sigma \cup \{\varepsilon\}} \delta\left(p \xrightarrow{a, \varepsilon \to X} q\right) \tag{1.83}$$

$$\oplus \bigoplus_{\substack{Y \in \Gamma \\ r \in Q \\ a \in \Sigma \cup \{\varepsilon\}}} w(p, Y, r) \otimes \delta\left(r \xrightarrow{a, Y \to X} q\right) \tag{1.84}$$

$$\oplus \bigoplus_{\substack{Y, Z \in \Gamma \\ r, s \in Q \\ a \in \Sigma \cup \{\varepsilon\}}} w(p, Y, r) \otimes w(r, Z, s) \otimes \delta\left(s \xrightarrow{a, YZ \to X} q\right) \tag{1.85}$$

$$\oplus \bigoplus_{\substack{Y, Z, W \in \Gamma \\ r, s, t \in Q \\ a \in \Sigma \cup \{\varepsilon\}}} w(p, Y, r) \otimes w(r, Z, s) \otimes w(s, W, t) \otimes \delta\left(t \xrightarrow{a, YZW \to X} q\right) \tag{1.86}$$

$$\oplus \quad \cdots \tag{1.87}$$

In general, the weights of the push computations cannot be derived recursively as they may depend on weights that are yet to be computed. Similar to computing treesums in WCFGs, we can use either fixed-point iteration or the semiring generalization of Newton's method for computing these weights. As mentioned earlier, the treesum of a WPDA $\mathfrak{P}$ is the weight $w(q_I, S, q_F)$.

1.6 PDA Transforms

Before presenting parsing algorithms for pushdown automata, we must take a detour and look at PDA transformations. Perhaps not surprisingly, just like in the CFG case, a PDA transformation is just a mapping from some (weighted) PDA to another (weighted) PDA. Although one can come up with many PDA transformations, we will focus mostly on three particular transformations: binarization, nullary removal and unary removal. These transformations will help us derive a normal form for PDAs that is analogous to the Chomsky Normal form for context-free grammars and allows computing stringsums efficiently.

Definition 1.6.1 (PDA Transform)**.** *A **PDA transform** $\mathcal{T}$ is a mapping that takes as input a pushdown automaton $\mathfrak{P}$ and returns another pushdown automaton $\mathcal{T}(\mathfrak{P})$. The input and output can be either weighted or unweighted.*

Sometimes we are only interested in PDA transforms that have certain properties. For instance, when applying a transformation $\mathcal{T}$ to a WPDA $\mathfrak{P}$ so that a stringsum algorithm can be run efficiently, we must ensure that the output of the transformation preserves the stringsums from the original automaton. In the unweighted case, this is equivalent to saying that the input and output of the transform recognize the same language. The following definition formalizes this intuition.

Definition 1.6.2 (Weak equivalence)**.** *Two (weighted) PDAs $\mathfrak{P}$ and $\mathfrak{P}'$ are called **weakly equivalent** if they have the same (weighted) languages,*

$$L(\mathfrak{P}) = L(\mathfrak{P}'). \tag{1.88}$$

A transformation that ensures weak equivalence only might collapse several runs of the original machine into a single one or it might introduce new runs, while still preserving the same (weighted) language. The weak equivalence requirement might be too weak sometimes as the output of the transformation loses some structure of the runs in the original automaton. For this reason, we introduce a stricter notion of equivalence.

Definition 1.6.3 (Semantics preserving)**.** *Let $\mathcal{W}$ be some semiring, $\mathfrak{P}$ a WPDA defined over $\mathcal{W}$ and $\mathfrak{P}' = \mathcal{T}(\mathfrak{P})$ the output of a transformation $\mathcal{T}$. We call the transformation $\mathcal{T}$ **semantics preserving** if there is a bijection $\phi\colon \Pi(\mathfrak{P}) \to \Pi(\mathfrak{P}')$ such that for every $\boldsymbol{\pi} \in \Pi(\mathfrak{P})$, it holds that*

$$w(\boldsymbol{\pi}) = w(\phi(\boldsymbol{\pi})). \tag{1.89}$$

Definition 1.6.4 (d-Weak equivalence)**.** *Two (weighted) PDAs $\mathfrak{P}$ and $\mathfrak{P}'$ are called **d-weakly equivalent**[28] if there is some semantics preserving transformation $\mathcal{T}$ such that $\mathfrak{P}' = \mathcal{T}(\mathfrak{P})$.*

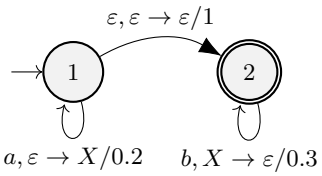
Example 1.6.1. *Consider the two WPDAs shown in Fig. 1.19. Both WPDAs have the initial configuration $(1, \varepsilon)$. The WPDA $\mathfrak{P}$ has the final configuration $(2, \varepsilon)$ while the WPDA $\mathfrak{P}'$ has the final configuration $(3, \varepsilon)$. Every accepting run of $\mathfrak{P}$ is preserved in $\mathfrak{P}'$ with the same weight.*

Lemma 1.6.1. *D-weak equivalence implies weak equivalence.*

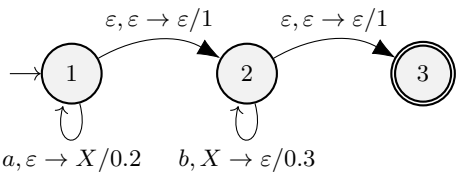
Proof. The weighted language of a WPDA is the set of weighted strings it derives. For any string $\boldsymbol{y} \in \Sigma^*$, a WPDA $\mathfrak{P}$ recognizes $\boldsymbol{y}$ with weight

$$w(\mathfrak{P}, \boldsymbol{y}) = \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathfrak{P}, \boldsymbol{y})} w(\boldsymbol{\pi}). \tag{1.90}$$

[28] The name comes from the term "weak derivational equivalence".



(a) Example of WPDA $\mathfrak{P}$ over the real semiring.



(b) Output $\mathfrak{P}' = \mathcal{T}(\mathfrak{P})$ of a semantics preserving transformation on the WPDA $\mathfrak{P}$.

Figure 1.19: Example of a WPDA $\mathfrak{P}$ accepting the language $L(\mathfrak{P}) = \{a^n b^n/(0.06)^n \mid n \in \mathbb{N}\}$ and a semantics preserving transformation $\mathcal{T}$ that outputs the d-weakly equivalent WPDA $\mathfrak{P}'$. $\mathfrak{P}'$ also recognizes $L(\mathfrak{P})$ and every run of $\mathfrak{P}$ is preserved in $\mathfrak{P}'$ with the same weight.

If two WPDAs $\mathfrak{P}$ and $\mathfrak{P}'$ are d-weakly equivalent, it holds that

$$w(\boldsymbol{\pi}) = w(\phi(\boldsymbol{\pi})), \tag{1.91}$$

for some bijection $\phi\colon \Pi(\mathfrak{P}, \boldsymbol{y}) \to \Pi(\mathfrak{P}', \boldsymbol{y})$ and every $\boldsymbol{\pi} \in \Pi(\mathfrak{P}, \boldsymbol{y}), \phi(\boldsymbol{\pi}) \in \Pi(\mathfrak{P}', \boldsymbol{y})$. Therefore,

$$\begin{aligned} w(\mathfrak{P}, \boldsymbol{y}) &= \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathfrak{P}, \boldsymbol{y})} w(\boldsymbol{\pi}) && (1.92)\\ &= \bigoplus_{\phi(\boldsymbol{\pi}) \in \Pi(\mathfrak{P}', \boldsymbol{y})} w(\phi(\boldsymbol{\pi})) && (1.93)\\ &= w(\mathfrak{P}', \boldsymbol{y}), && (1.94) \end{aligned}$$

which means that the two machines have the same weighted languages. □

The definition of d-weak equivalence requires only that the runs in the two machines must exist in one-to-one correspondence but it does not say anything about the structure of these runs. This means that the elements (transitions and configurations) of these runs might not be in one-to-one correspondence. For instance, each run of the WPDA $\mathfrak{P}'$ from Fig. 1.19 has one more transition and one more configuration than its corresponding run in $\mathfrak{P}$.

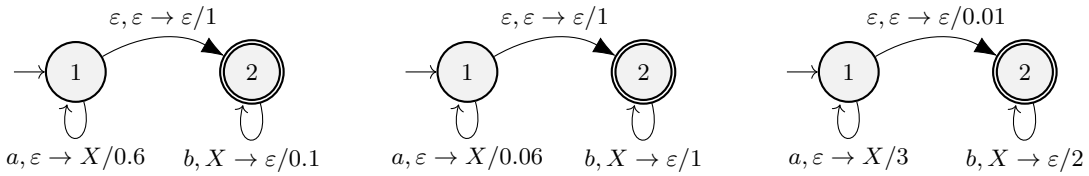
Definition 1.6.5 (Strong equivalence)**.** *Two WPDAs $\mathfrak{P}$ and $\mathfrak{P}'$ are called **strongly equivalent** if they have the same set of runs up to a redistribution of the weights along the run.*

Therefore, the runs of two strongly equivalent WPDAs are not only in one-to-one correspondence, but they also have the same structure. In the boolean semiring two strongly equivalent PDAs have identical sets of runs.

Example 1.6.2. *The WPDAs shown in Fig. 1.20 are strongly equivalent to the WPDA from Fig. 1.19a. The runs in each of these WPDAs are in one-to-one correspondence but their weights are redistributed along the runs. Moreover, the transitions of each of these runs are the same, up to a weight relabeling.*

Lemma 1.6.2. *Strong equivalence implies d-weak equivalence.*

Proof. By definition, the sets of runs of two strongly equivalent WPDAs are the same, thus the identity function (when ignoring the weights) is a valid bijection between the derivation sets. Moreover, the weights are preserved. □



(a) (b) (c)

Figure 1.20: Example of WPDAs that are strongly equivalent to the WPDA defined in Fig. 1.19a.

Definition 1.6.6. *A bottom-up WPDA is in* ***normal form*** *if all its scanning transitions are k-pop, 1-push for $k \leq 2$ and all its non-scanning transitions are 2-pop, 1-push. Similarly, a top-down WPDA is in* ***normal form*** *if all its scanning transitions are 1-pop, k-push for $k \leq 2$ and all its non-scanning transitions are 1-pop, 2-push.*

In the remainder of this section we will give transformations for converting bottom-up WPDAs into the normal form: binarization, nullary removal and unary removal. The transformations must be applied in this order because nullary removal, as we will see later, can generate new unary transitions. Mirror images of these transformations can be easily constructed for top-down WPDAs. We leave these as an exercise.

1.6.1 Binarization

The binarization transformation is entirely analogous to the binarization in CFGs and is symmetric for bottom-up and top-down CFGs. In the CFG case, binarization ensures that derivation trees are binary, therefore the weight of the derivations of a nonterminal depends only on the weights of the derivations of two children. In the case of bottom-up WPDAs, it ensures that at most two stack symbols are popped by each transition. Since the stringsum algorithm is a dynamic programming algorithm that computes weights of runs by reusing the weights of shorter runs, this means that in a binarized WPDA it will reuse the weights of only two such runs. We defer the details to a later section.

Whenever the WPDA has a transition that pops more than two stack symbols, i.e.

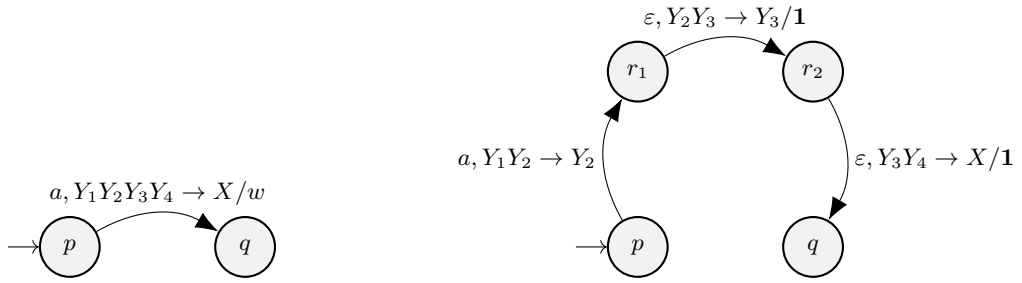
$$p \xrightarrow{a, Y_1 \cdots Y_k \to X/w} q, \quad k > 2, \tag{1.95}$$

it needs to be replaced with a sequence of transitions, each of which pops exactly two symbols. Fig. 1.21 shows an example of a transition in the original automaton and the sequence of transitions it gets replaced with.

1.6.2 Nullary Removal

In the case of PDA binarization, the construction resembled closely the binarization transformation for CFGs. Although nullary transitions are analogous to CFG nullary productions, the nullary removal transformation for PDAs does not have an exact analogous for CFGs. The PDA nullary removal transformation consists of three steps: *partitioning*, *precomputation* and *removal*. Alg. 18 formalizes the partitioning and removal steps.

Definition 1.6.7. *A transition of a bottom-up WPDA is called* ***nullary*** *if it is of the form $p \xrightarrow{\varepsilon, \varepsilon \to X/w} q$.*



(a) k-pop, 1-push WPDA transition for $k > 2$.

(b) Sequence of binary transitions resulting from binarization.

Figure 1.21: Example of binarization in a WPDA.

Algorithm 17 The WPDA binarization algorithm.

```
1. def PDABinarize(𝔓):
2.   𝔓′ ← 𝔓                                          ▷Make a copy of 𝔓.
3.   for p --a,Y1···Yk→X/w--> q ∈ δ :
4.     if k > 2 :
5.       remove p --a,Y1···Yk→X/w--> q from δ
6.       add a new state r1 to Q
7.       add a new transition p --a,Y1Y2→Y2/w--> r1 to δ
8.       for i = 2 . . . k − 2 :
9.         add a new state ri to Q
10.        add a new transition ri−1 --ε,YiYi+1→Yi+1/1--> ri to δ
11.      add a new transition rk−2 --ε,Yk−1Yk→X/1--> q to δ
12.  return 𝔓′
```

Partitioning For each stack symbol $X \in \Gamma$, we replace X with two stack symbols, X^{ε} and $X^{\notin}$, with the following meaning: a push computation that pushes X^{ε} scans ε while a push computation that pushes $X^{\notin}$ scans some string $\boldsymbol{y} \in \Sigma^*$. Now we need to replace the stack symbols in each transition. For each transition $p \xrightarrow{a,Y_1 \cdots Y_k \to X/w} q$, we remove it and replace it with 2^k new transitions of the form $p \xrightarrow{a,Y_1^{\nu_1} \cdots Y_k^{\nu_k} \to X^{\nu}/w} q$, where $\nu_i \in \{\varepsilon, \notin\}$ and $\nu = \varepsilon$ if $\nu_i = \varepsilon$ for all $i = 1 \ldots k$ and $a = \varepsilon$. The goal is to remove all the symbols of the form X^{ε} and account for the weights of their corresponding push computations.

Precomputation We compute the weights of all non-scanning push computations by solving the system of quadratic equations:

$$w_{pXq} = \delta\left(p \xrightarrow{\varepsilon,\varepsilon \to X} q\right) \tag{1.96}$$

$$\oplus \bigoplus_{r,Y} w_{pYr} \otimes \delta\left(r \xrightarrow{\varepsilon,Y \to X} q\right) \tag{1.97}$$

$$\oplus \bigoplus_{r,s,Y,Z} w_{pYr} \otimes w_{rZs} \otimes \delta\left(s \xrightarrow{\varepsilon,YZ \to X} q\right) \tag{1.98}$$

Additionally, define the weights of the form w_{pXYq} as

$$w_{pXYq} = \bigoplus_r w_{pXr} \otimes w_{rYq}. \tag{1.99}$$

Removal First, we can delete all transitions that push a symbol of the form X^{ε}. This can be safely done as non-scanning push computations will be removed and their weights will be accounted for. If the WPDA has already been binarized, there are only several types of transitions that need to be replaced. Sometimes a symbol of the type X^{ε} is popped immediately after it is pushed, with no symbols scanned in between. For the following types of transitions, we create new versions in which the popped X^{ε} symbols are removed and the weights of the corresponding push computations are multiplied in:

$$\begin{array}{ll} \text{For each:} & \text{Replace with } (\forall t \in Q): \\ p \xrightarrow{a, Y^{\varepsilon} \to X^{\not\varepsilon}/w} q & t \xrightarrow{a, \varepsilon \to X^{\not\varepsilon}/w_{tYp} \otimes w} q \\ p \xrightarrow{a, Y^{\varepsilon} Z^{\varepsilon} \to X^{\not\varepsilon}/w} q & t \xrightarrow{a, \varepsilon \to X^{\not\varepsilon}/w_{tYZp} \otimes w} q \\ p \xrightarrow{a, Y^{\not\varepsilon} Z^{\varepsilon} \to X^{\not\varepsilon}/w} q & t \xrightarrow{a, Y^{\not\varepsilon} \to X^{\not\varepsilon}/w_{tZp} \otimes w} q \end{array} \tag{1.100}$$

Now we are only left with transitions of the type $p \xrightarrow{a, Y^{\varepsilon} Z^{\not\varepsilon} \to X^{\not\varepsilon}/w} q$, which correspond to a non-scanning push computation that pushes Y, followed by a scanning push computation that pushes Z. For this reason, we cannot replace these transitions as we did for the previous types. We create new stack symbols of the type X_{rs} for all $X \in \Gamma$ and $r, s \in Q$. These symbols stand for a sequence of zero or more non-scanning push computations, followed by a push computation that pushes X (this push computation must be 0-pop, 1-push, because all the other types of transitions expect a symbol of the form $X^{\not\varepsilon}$ on the top of the stack). Then we need to modify (the transitions that we modified in the previous step) the 0-pop transitions to first simulate 0 or more nullary transitions.

$$\begin{array}{ll} \text{For each:} & \text{Replace with } (\forall s \in Q): \\ t \xrightarrow{a, \varepsilon \to X^{\not\varepsilon}/w_{tYp} \otimes w} q & s \xrightarrow{a, \varepsilon \to X_{st}/w_{tYp} \otimes w} q \\ t \xrightarrow{a, \varepsilon \to X^{\not\varepsilon}/w_{tYZp} \otimes w} q & s \xrightarrow{a, \varepsilon \to X_{st}/w_{tYZp} \otimes w} q \end{array} \tag{1.101}$$

Finally, we need to replace all transitions of the type $p \xrightarrow{a, Y^{\varepsilon} Z^{\not\varepsilon} \to X^{\not\varepsilon}/w} q$. For all $r, s, t \in Q$,

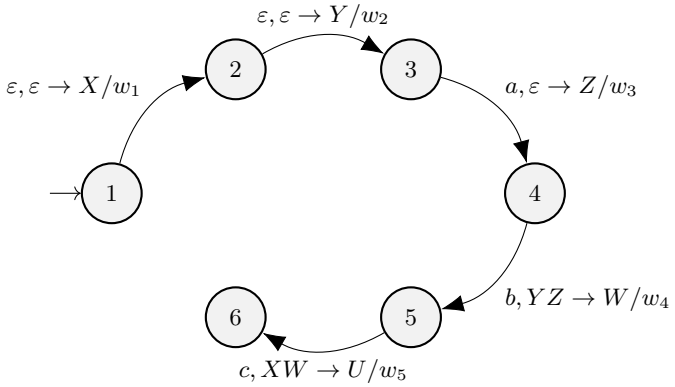
$$\begin{array}{ll} \text{For each:} & \text{Replace with } (\forall s \in Q): \\ p \xrightarrow{a, Y^{\varepsilon} Z^{\not\varepsilon} \to X^{\not\varepsilon}/w_{sYt} \otimes w} q & p \xrightarrow{a, Z_{rt} \to X_{rs}/w} q. \end{array} \tag{1.102}$$

And, for all $X \in \Gamma, p, q \in Q$, we add the following transitions to remove the state annotations:

$$q \xrightarrow{a, X_{pp} \to X^{\not\varepsilon}/\mathbf{1}} q. \tag{1.103}$$

Fig. 1.22 shows a sequence of transitions in a WPDA that gets modified by the nullary removal transformation.

(a) Subset of the nodes and transitions in a WPDA $\mathfrak{P}$ with nullary transitions.



(b) Modified transitions in $\mathfrak{P}$ after nullary removal.

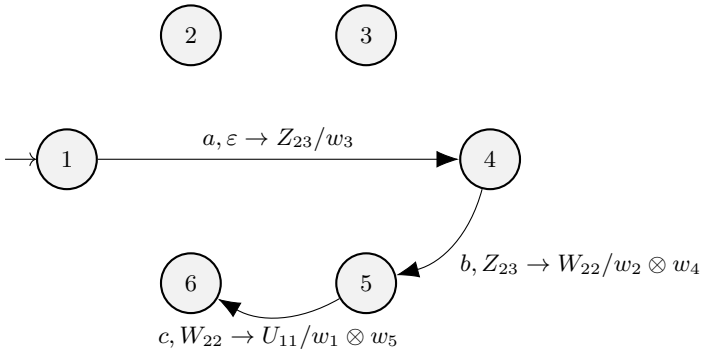


Figure 1.22: Example nullary removal in a WPDA.

1.6.3 Unary Removal

Before we dive into the unary removal algorithm, let's start by defining what a unary transition in a WPDA.

Definition 1.6.8. *A transition of a bottom-up WPDA is called* ***unary*** *if it is of the form* $p \xrightarrow{\varepsilon, X \to Y/w} q$.

Unary transitions in a WPDA can be problematic for dynamic programming algorithms as they can form unary cycles that can be traversed an unbounded number of times. But what is a cycle in a WPDA? Recall that in the WFSA case, a cycle is a path that starts and ends in the same state. A cycle in a WPDA can be defined similarly, except that we need to take the stack into account.

Definition 1.6.9. *A* ***cycle*** *in a WPDA is a run* $\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \tau_1, (q_1, \boldsymbol{\gamma}_1), \cdots, \tau_k, (q_k, \boldsymbol{\gamma}_k)$*, where* $q_k = q_0$ *and* $\boldsymbol{\gamma}_k = \boldsymbol{\gamma}_0$*. Moreover, a* ***unary cycle*** *is a cycle whose transitions are unary.*

The construction resembles the ε-removal procedure for WFSAs. Whenever a non-unary transition pushes to the stack a symbol X, a unary transition can pop X and push another stack symbol Y, which can be popped immediately by another unary transition and so on. Each of these transitions replaces the top symbol of the stack, leaving the rest of the stack unchanged. This means that we could remove all of these transitions and push directly the last symbol pushed by such a run. However, we also need to take into account the weight of this run. In order to do this we need

Algorithm 18 The WPDA nullary removal algorithm. We assume that we have already computed the weights of the type w_{pXq} corresponding to the total weight of the push computations from p to q that push X.

def PDARemoveNullary($\mathfrak{P}$):
  $\mathfrak{P}' \leftarrow (Q, \Sigma, \Gamma', \delta', (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ ▷*Create new WPDA $\mathfrak{P}'$ over the same semiring.*
  $\Gamma' \leftarrow \{X^\nu \mid X \in \Gamma, \nu \in \{\varepsilon, \not\varepsilon\}\} \cup \{X_{pq} \mid X \in \Gamma, p, q \in Q\}$
  for $\tau = p \xrightarrow{a, X_1 \ldots X_k \to Y/w} q \in \delta$: ▷*Partitioning.*
    for $\nu_1, \ldots, \nu_k \in \{\varepsilon, \not\varepsilon\}$:
      if $\nu_i = \varepsilon$ for all $i \in 1, \ldots, k$ and $a = \varepsilon$:
        $\nu = \varepsilon$
      else
        $\nu = \not\varepsilon$
      add $p \xrightarrow{a, X_1^{\nu_1} \ldots X_k^{\nu_k} \to Y^\nu / w} q$ to δ'
  for $\tau \in \delta$: ▷*Removal.*
    if $\tau = p \xrightarrow{a, Y^\varepsilon \to X^{\not\varepsilon}/w} q$:
      for $t, s \in Q$:
        add $s \xrightarrow{a, \varepsilon \to X_{st}/w_{tYp} \otimes w} q$ to δ'
    if $\tau = p \xrightarrow{a, Y^\varepsilon Z^\varepsilon \to X^{\not\varepsilon}/w} q$:
      for $t, s \in Q$:
        add $s \xrightarrow{a, \varepsilon \to X_{st}/w_{tYZp} \otimes w} q$ to δ'
    if $\tau = p \xrightarrow{a, Y^{\not\varepsilon} Z^\varepsilon \to X^{\not\varepsilon}/w} q$:
      for $t \in Q$:
        add $t \xrightarrow{a, Y^{\not\varepsilon} \to X^{\not\varepsilon}/w_{tZp} \otimes w} q$ to δ'
    if $\tau = p \xrightarrow{a, Y^\varepsilon Z^{\not\varepsilon} \to X^{\not\varepsilon}/w} q$:
      for $t, s, r \in Q$:
        add $p \xrightarrow{a, Z_{rt} \to X_{rs}/w_{sYt} \otimes w} q$ to δ'
  for $p, q \in Q$, $X \in \Gamma$:
    add $q \xrightarrow{\varepsilon, X_{pp} \to X^{\not\varepsilon}/\mathbf{1}} q$ to δ' ▷*Remove state annotations.*
  return $\mathfrak{P}'$

to construct a WFSA, which we will call $\mathcal{A}_\mathsf{u}$, from the input WPDA and use Lehmann's algorithm to compute its closure. However, unlike in the WFSA case, $\mathcal{A}_\mathsf{u}$'s states will be configurations of the original WPDA.

Unary removal can be performed in WPDAs using the following three steps, which are also formalized in Alg. 19:

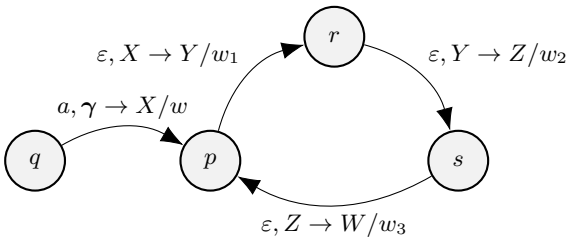
(i) Construct the automaton $\mathcal{A}_\mathsf{u}$ from $\mathfrak{P}$'s unary transitions by adding a transition $(p, X) \xrightarrow{\varepsilon/w} (q, Y)$ in $\mathcal{A}_\mathsf{u}$ if there is a transition $p \xrightarrow{\varepsilon, X \to Y/w} q$ in $\mathfrak{P}$.

(ii) Compute the closure $\kappa(\cdot, \cdot)$ of $\mathcal{A}_\mathsf{u}$ using Lehmann's algorithm.

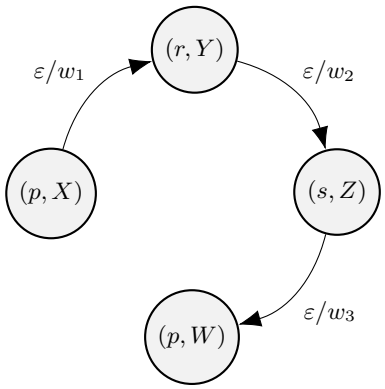
(iii) Replace every non-unary transition $p \xrightarrow{a, \boldsymbol{\gamma} \to X/w} q$ with $p \xrightarrow{a, \boldsymbol{\gamma} \to Y/w \otimes \kappa((q,X),(r,Y))} r$.

Fig. 1.23 shows an example of unary removal in a weighted pushdown automaton.

(a) Subset of the nodes and transitions in a WPDA $\mathfrak{P}$ with unary transitions.



(b) Automaton $\mathcal{A}_{\mathsf{u}}$ constructed from $\mathfrak{P}$.



(c) Modified transition in $\mathfrak{P}$ resulting from unary removal.

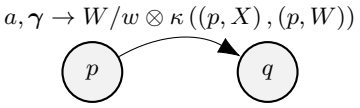


Figure 1.23: Example of unary removal in a WPDA.

Algorithm 19 The WPDA unary removal algorithm.

```
1. def PDARemoveUnary(𝔓):
2.     initialize 𝔓_ψ as a copy of 𝔓 without unary transitions
3.     initialize the automaton 𝒜_u
4.     compute the closure κ(·,·) of 𝒜_u        ▷Requires Lehmann's algorithm; Runs in O(|Q|^3 |Γ|^3)
5.     for p --a,γ→X/w--> q ∈ δ_ψ :
6.         for r ∈ Q, Y ∈ Γ :
7.             w' ← w ⊗ κ((q, X), (r, Y))
8.             replace p --a,γ→X/w--> q with p --a,γ→Y/w'--> r
9.     return 𝔓_ψ
```

1.7 WPDA Parsing

In this section we will give efficient algorithms for parsing in different types of WPDAs. We will use dynamic programming algorithms that leverage the structure of the WPDA runs. Just like in the case of WCFGs, the parsing problem traditionally refers to the *recognition problem*, i.e., checking whether a string is recognized by a pushdown automaton. In semiring-weighted PDAs, this is exactly computing the stringsum $w(\boldsymbol{y})$ of some string $\boldsymbol{y} \in \Sigma^*$ under a WPDA $\mathfrak{P}$ and it reduces to a number of other specific problems in some specific semirings.

1.7.1 Lang's Algorithm

We will start by discussing Lang (1974)'s algorithm, which works on a restricted type of WPDA, called *simple*. Just like in the case of WCFGs, the WPDA stringsum algorithms use a chart and compute the weights of items sequentially from the weights of previously-computed items.

Definition 1.7.1 (Simple WPDA)**.** *A WPDA is called* ***simple*** *if it only has k-pop, l-push transitions for $k \leq 1$ and $l \leq 1$. Moreover, the initial configuration is $(q_I, \$\$)$ and the final configuration is $(q_F, \$\$)$, for some states $q_I, q_F \in Q$ and a special symbol $\$ \in \Gamma$.*

Therefore, a simple WPDA has only four types of transitions: 0-pop, 0-push transition (leaves the stack unchanged), 0-pop, 1-push (places some symbol on top of the stack, leaving the rest of the stack unchanged), 1-pop, 0-push (removes the top symbol of the stack) and 1-pop, 1-push (replaces the top symbol with some other symbol).

The items in Lang's algorithm are of the form $[i, p, X, Y, j, q]$, which correspond to all runs such that:

- At the end of the run, the WPDA reaches state q with the stack $\boldsymbol{\gamma}Y$, i.e., the symbol Y is at the top of the stack, and the last symbol scanned was $\boldsymbol{y}_j$;
- The next stack symbol is X, i.e., $\boldsymbol{\gamma} = \boldsymbol{\gamma}'X$, and it was last at the top of the stack when the WPDA was in state p and the last symbol scanned was $\boldsymbol{y}_i$.

If the algorithm computes the weight $\boldsymbol{\beta}[i, p, X, Y, j, q] = w$ of an item $[i, p, X, Y, j, q]$, then the total weight of all runs of type $[i, p, X, Y, j, q]$ is w. It does so recursively from items corresponding to shorter runs in order of increasing span length of the substring scanned, using inference rules for each type of transition. The goal item is $[0, q_I, \$, \$, n, q_F]$, whose weight corresponds to the total weight of the accepting runs scanning $\boldsymbol{y}$, i.e., the stringsum of $\boldsymbol{y}$ under $\mathfrak{P}$.

Complexity. The algorithm derives a single weight for each item of the type $[i, p, X, Y, j, q]$, thus it has a space complexity of $\mathcal{O}(n^2|Q|^2|\Gamma|^2)$. In the worst case (using the inference rule for the 1-pop, 0-push), computing the weight of each new item requires a runtime of $\mathcal{O}(n^3|Q|^4|\Gamma|^3)$.

1.7.2 Butoi et al. (2022)

We saw in the previous subsection that the bottleneck of Lang's algorithm is the inference rule for 1-pop, 0-push transitions. Can we derive a stringsum algorithm on a WPDA that does not have such rules? Yes! In fact, the bottom-up and top-down subclasses of WPDAs were defined in order to avoid having such transitions. Moreover, the structure of the push/pop computations makes it straightforward to derive dynamic programming algorithms. In the rest of this subsection we will assume that the WPDAs are bottom-up and in normal form. An algorithm for top-down WPDAs

Algorithm 20 Lang's algorithm. Computes stringsums of simple WPDAs.

```
1.  def Lang(𝔓, y):
2.    β ← 0                                                          ▷Initialize chart.
3.    n ← |y|
4.    β[0, q_I, $, $, 0, q_I] ← 1
5.    for (p --a,ε→Y--> q) ∈ δ :                                     ▷0-pop, 1-push
6.      for i = 0, ··· , n − |a| :
7.        for X ∈ Γ :
8.          if y(i:i − |a|] = a :
9.            β[i, p, X, Y, i + |a|, q] ← δ (p --a,ε→Y--> q)
10.   for span = 1, ··· , n :
11.     for i = 0, ··· , n − span + 1 :
12.       j ← i + span
13.       for p, q, r ∈ Q, X, Y ∈ Γ :
14.         for (r --a,ε→ε--> q) ∈ δ :                               ▷0-pop, 0-push
15.           if y(j:j − |a|] = a :
16.             β[i, p, X, Y, j, q] ← β[i, p, X, Y, j − |a|, r] ⊗ δ (r --a,ε→ε--> q)
17.       for p, q, r ∈ Q, X, Y, Z ∈ Γ :
18.         for (r --a,Z→Y--> q) ∈ δ :                               ▷1-pop, 1-push
19.           if y(j:j − |a|] = a :
20.             β[i, p, X, Y, j, q] ← β[i, p, X, Z, j − |a|, r] ⊗ δ (r --a,Z→Y--> q)
21.       for p, q, r, s ∈ Q, X, Y, Z ∈ Γ :
22.         for (s --a,Z→ε--> q) ∈ δ :                               ▷1-pop, 0-push
23.           for k ∈ i, ··· , j − |a| :
24.             if y(j:j − |a|] = a :
25.               β[i, p, X, Y, j, q] ← β[i, p, X, Y, k, r] ⊗ [k, r, Y, Z, j − |a|, s] ⊗ δ (r --a,Z→ε--> q)
26.   return β[0, q_I, $, $, n, q_F]
```

can be easily defined as its mirror image. Just like the CNF requirement for the CKY algorithm, the WPDA normal form is essential for ensuring that the weight of a push computation depends on the weights of at most 2 other push computations.

Before we dive into the details of the algorithm, we need to introduce a new type of push computation, which closely resembles the one we defined previously but it additionally indexes substrings of the input.

Definition 1.7.2. *Let $\mathfrak{P}$ be a bottom-up WPDA and $\boldsymbol{y} \in \Sigma^*$ an input string. A push computation of type $[i, p, X, j, q]$ is a push computation of X from state p to state q that scans $\boldsymbol{y}(i : j]$.*

Therefore, the algorithm has items corresponding exactly to push computations of this type. If an item $\boldsymbol{\beta}[i, p, X, j, q]$ is derived with some weight w, then the total weight of the push computations of type $[i, p, X, j, q]$ is w. When the automaton is in normal form, there are only 3 types of transitions. Therefore, there are three categories of push computations based on their final transition and the algorithm includes an inference rule for each. First are those consisting of a single 0-pop, 1-push transition. The other two are the push computations ending with a 1-pop, 1-push or a 2-pop, 1-push transition, both of which can be recursively built from shorter push computations. The algorithm returns the weight $\boldsymbol{\beta}[0, q_I, S, n, q_F]$, which corresponds to the total weight of all accepting runs (all accepting runs are push computations of S in a bottom-up WPDA) that scan $\boldsymbol{y}$, i.e., the stringsum of $\boldsymbol{y}$ under $\mathfrak{P}$.

Theorem 1.7.1. *Let $\mathfrak{P}$ be a WPDA and $\boldsymbol{y} \in \Sigma^*$ an input string. The weight $\boldsymbol{\beta}[i, p, X, j, q]$ computed*

Algorithm 21 Bottom-up WPDA stringsum algorithm.

```
1.  def WPDAStringsum(𝔓, y):
2.    β ← 0                                                          ▷Initialize chart.
3.    n ← |y|
4.    for i = 0, ··· , n − 1 :
5.      for (p --a,ε→X--> q) ∈ δ :                                   ▷0-pop, 1-push
6.        if y_{i+1} = a :
7.          β[i, p, X, i + 1, q] ← δ(p --a,ε→X--> q)
8.    for span = 2, ··· , n :
9.      for i = 0, ··· , n − span + 1 :
10.       j ← i + span
11.       for p ∈ Q :
12.         for (r --a,Y→X--> q) ∈ δ :                               ▷1-pop, 1-push
13.           if y_j = a :
14.             β[i, p, X, j, q] ← β[i, p, X, j, q] ⊕ β[i, p, Y, j − 1, r] ⊗ δ(r --y_j,Y→X--> q)
15.       for p, r ∈ Q :
16.         for (s --a,YZ→X--> q) ∈ δ :                              ▷2-pop, 1-push
17.           if y(j − |a|:j] = a :
18.             for k = i + 1, ··· , j − |a| − 1 :
19.               β[i, p, X, j, q] ← β[i, p, X, j, q] ⊕ β[i, p, Y, k, r] ⊗ β[k, r, Z, j − |a|, s] ⊗ δ(s --a,YZ→X--> q)
20.   return β[0, q_I, S, n, q_F]
```

by Alg. 21 is the total weight of all push computations of $\mathfrak{P}$ *of type* $[i, p, X, j, q]$.

Proof. We prove the theorem by induction on the span length, $\mathsf{span} = j - i$.

Base Case. Assume that $j - i = 1$. The only push computations from state p to q that push X and scan $\boldsymbol{y}(i{:}j]$ are ones that have the single transition $\tau = p \xrightarrow{\boldsymbol{y}_j,\varepsilon\to X/w} q$. There cannot exist others, because normal form requires that any additional non-scanning transitions would decrease the stack height. So the total weight of all such push computations is w, and the algorithm correctly sets $\boldsymbol{\beta}[i, p, X, j, q] \leftarrow w$ at line 7.

Inductive Step. Assume that the statement holds for any spans of length at most $(\mathsf{span} - 1)$ and consider a span of length span. For such spans, the algorithm computes the total weight of all push computations $\boldsymbol{\pi}$ of type $[i, p, X, j, q]$, for all $X \in \Gamma$, $p, q \in Q$, and $j - i = \mathsf{span}$.

This weight must be the sum of weights of three types of push computations: those that end with 0-pop transitions, with 1-pop transitions, and with 2-pop transitions. But ending with a 0-pop transition is impossible, because such push computations must have only one transition and therefore $j - i \leq 1$. The 1-pop and 2-pop parts of the sum are computed at lines 11–14 and 15–19 of the algorithm, respectively.

Ending with a 1-pop transition. The algorithm sums over all possible ending transitions $\tau_{\text{end}} = r \xrightarrow{\boldsymbol{y}_j,Y\to X/w} q$. (Normal form requires that this transition is scanning.) Let Π be the set of all push computations of type $[i, p, X, j, q]$ ending in τ_{end}, and let Π' be the set of all push computations of type $[i, p, Y, j - 1, r]$. Every push computation in Π must be of the form $\boldsymbol{\pi} = \boldsymbol{\pi}' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}' \in \Pi'$, and conversely, for every $\boldsymbol{\pi}' \in \Pi'$, we have $\boldsymbol{\pi}' \circ \tau_{\text{end}} \in \Pi$. By the induction hypothesis, the

total weight of Π' was computed in a previous iteration. Then, by distributivity, we have:

$$\bigoplus_{\boldsymbol{\pi}\in\Pi}\bigotimes_{\tau\in\boldsymbol{\pi}}\delta(\tau) = \bigoplus_{\boldsymbol{\pi}'\in\Pi'}\bigotimes_{\tau\in\boldsymbol{\pi}'}\delta(\tau)\otimes\delta(\tau_{\text{end}}) \tag{1.104}$$

$$= \left(\bigoplus_{\boldsymbol{\pi}'\in\Pi'}\bigotimes_{\tau\in\boldsymbol{\pi}'}\delta(\tau)\right)\otimes\delta(\tau_{\text{end}}) \tag{1.105}$$

$$= \boldsymbol{\beta}[i,p,Y,j-1,r]\otimes\delta(\tau_{\text{end}}). \tag{1.106}$$

Ending with a 2-pop transition. The algorithm sums over all possible ending transitions $\tau_{\text{end}} = s \xrightarrow{a,YZ\to X/w} q, a \in \{\boldsymbol{y}_j, \varepsilon\}$. Every push computation $\boldsymbol{\pi}$ that ends with τ_{end} decomposes uniquely into $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}'$ and $\boldsymbol{\pi}''$ are push computations of type $[i,p,Y,k,r]$ and $[k,r,Z,j-|a|,s]$, respectively, for some $k \in [i+1{:}j-|a|-1]$ and $r \in Q$. We call (k,r) the **split point** of $\boldsymbol{\pi}$.

The algorithm sums over all split points (k,r). Let Π be the set of all push computations of type $[i,p,X,j,q]$ ending in τ_{end} with split point (k,r), and let Π' and Π'' be the sets of all push computations of type $[i,p,Y,k,r]$ and $[k,r,Z,j-|a|,s]$, respectively. Every $\boldsymbol{\pi} \in \Pi$ must be of the form $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}' \in \Pi'$ and $\boldsymbol{\pi}'' \in \Pi''$, and conversely, for every $\boldsymbol{\pi}' \in \Pi'$ and $\boldsymbol{\pi}'' \in \Pi''$, $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}} \in \Pi$. Because $i < k$, we must have $j - |a| - k \le j - k < j - i$, and because $k < j - |a|$, we must have $k - i < j - |a| - i \le j - i$. By the induction hypothesis, the total weight of Π' and Π'' were fully computed in a previous iteration. As in the previous case, by distributivity we have

$$\bigoplus_{\boldsymbol{\pi}\in\Pi}\bigotimes_{\tau\in\boldsymbol{\pi}}\delta(\tau) = \boldsymbol{\beta}[i,p,Y,k,r]\otimes\boldsymbol{\beta}[k,r,Z,j-|a|,s]\otimes\delta(\tau_{\text{end}}) \tag{1.107}$$

□

Complexity. The algorithm stores a weight for each item $[i,p,X,j,q]$, giving a space complexity of $\mathcal{O}\left(n^2|Q|^2|\Gamma|.\right)$. Computing the weight of each new item requires, in the worst case (the inference rule for 2-pop transitions), iterating over stack symbols $Y, Z \in \Gamma$, indices $j \in [0{:}n]$ and states $q, r \in Q$, resulting in a runtime of $\mathcal{O}(n|Q|^2|\Gamma|^2)$ per item. So the algorithm has a runtime of $\mathcal{O}(n^3|Q|^4|\Gamma|^3)$, the same as Lang's algorithm. This runtime can be improved by splitting the inference rule for 2-pop transitions into two rules:

$$\frac{[k,r,Z,j-|a|,s]}{[k,r,XY,j,q]} \qquad s \xrightarrow{a,YZ\to X/w} q, \quad \boldsymbol{y}(j-|a|{:}j] = a \tag{1.108}$$

$$\frac{[i,p,Y,k,r] \quad [k,r,XY,j,q]}{[i,p,X,j,q]} \tag{1.109}$$

The first rule has $\mathcal{O}(n^2|Q|^3|\Gamma|^3)$ instantiations and the second rule has $\mathcal{O}(n^3|Q|^3|\Gamma|^2)$. So, although we have lost the space-efficiency gain, the total time complexity is now in $\mathcal{O}((n^3|\Gamma|^2 + n^2|\Gamma|^3)|Q|^3)$, a speedup of a factor of more than $|Q|$. Furthermore, Lang's algorithm only works on simple PDAs. To make the algorithms directly comparable, we can assume in the 2-pop, 1-push case that $X = Y$. This reduces the space complexity by a factor of $|\Gamma|$ again. Moreover, it reduces the number of instantiations of the inference rules above to $\mathcal{O}(n^2|Q|^3|\Gamma|^2)$ and $\mathcal{O}(n^3|Q|^3|\Gamma|)$, respectively. So the total time complexity is in $\mathcal{O}(n^3|Q|^3|\Gamma|^2)$, which is a speedup over Lang's algorithm by a factor of $|Q| \cdot |\Gamma|$.

Stringsums of top-down WPDAs. A very similar algorithm can be defined by leveraging the structure of pop computations in top-down WPDAs. The items of this algorithm have the same form but now correspond to pop computations. When the WPDA is in normal form, we have again 3 types of pop computations: one formed of a single 1-pop, 0-push transition, and two ending in either a 1-pop, 1-push or a 1-pop, 2-push transition, whose weights can be derived recursively from the weights of their initial transition and 1 or 2 other shorter pop computations. It is important to notice that a pop computation always consists of a transition and a series of other shorter computations. Since the algorithm computes the items weights recursively, starting from the shortest pop computations, the string will be processed in reversed order.

Bibliography

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. *Context-Free Languages and Pushdown Automata*, pages 111–174. Springer Berlin Heidelberg, Berlin, Heidelberg.

Yehoshua Bar-Hillel, M. Perles, and E. Shamir. 1961. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172. Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley 1964, 116–150.

Alexandra Butoi, Brian DuSell, Tim Vieira, Ryan Cotterell, and David Chiang. 2022. Algorithms for weighted pushdown automata. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, USA.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian program analysis. *J. ACM*, 57(6).

Sheila A. Greibach. 1963. The undecidability of the ambiguity problem for minimal linear grammars. *Information and Control*, 6(2):119–125.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. Automata theory, languages, and computation. *International Edition*, 24(2).

John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

Mark Johnson. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics*.

Mark Johnson and Brian Roark. 2000. Compact non-left-recursive grammars using the selective left-corner transform and factoring. In *COLING 2000 Volume 1: The 18th International Conference on Computational Linguistics*.

Tadao Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages.

Donald E. Knuth. 1977. A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Hans-Jörg Kreowski. 1979. A pumping lemma for context-free graph languages. In *Graph-Grammars and Their Application to Computer Science and Biology*, pages 270–283, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *ICALP 1974: Automata, Languages and Programming*, pages 255–269.

Robert C. Moore. 2000. Removing left recursion from context-free grammars. In *1st Meeting of the North American Chapter of the Association for Computational Linguistics*.

Mark-Jan Nederhof and Giorgio Satta. 2008. *Probabilistic Parsing*, pages 229–258. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mark-Jan Nederhof and Giorgio Satta. 2009. Computing partition functions of pcfgs. *Research on Language and Computation*, 7:233.

Clemente Pasti, Andreas Opedal, Tiago Pimentel, Tim Vieira, Jason Eisner, and Ryan Cotterell. 2022. On the intersection of context-free and regular languages.

Philip Resnik. 1992. Left-corner parsing and psychological plausibility. In *The 14th International Conference on Computational Linguistics*.

G.E. Révész. 2015. *Introduction to Formal Languages*. Dover Books on Mathematics. Dover Publications.

Brian Roark and Mark Johnson. 1999. Efficient probabilistic top-down and left-corner parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 421–428, College Park, Maryland, USA. Association for Computational Linguistics.

D. J. Rosenkrantz and P. M. Lewis. 1970. Deterministic left corner parsing. In *11th Annual Symposium on Switching and Automata Theory (swat 1970)*, pages 139–152.

Michael Sipser. 1996. *Introduction to the Theory of Computation*, 1st edition. International Thomson Publishing.

J. W. Thatcher. 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *J. Comput. Syst. Sci.*, 1(4):317–322.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Inf. Control.*, $10 : 189 - -208$.