Prefix Parsing is Just Parsing

Clemente Pasti[1,5] **Andreas Opedal**[1] **Timothy J. O'Donnell**[2,3,4,5]
Ryan Cotterell[1] **Tim Vieira**[1]
{clemente.pasti, andreas.opedal, ryan.cotterell}@inf.ethz.ch
timothy.odonnell@mcgill.ca tim.f.vieira@gmail.com
[1]ETH Zürich [2]McGill University [3]Canada CIFAR AI Chair [4]Mila [5]CHI-FRO

Abstract

Prefix parsing asks whether an input prefix can be extended to a complete string generated by a given grammar. In the weighted setting, it also provides prefix probabilities, which are central to context-free language modeling, psycholinguistic analysis, and syntactically constrained generation from large language models. We introduce the *prefix grammar transformation*, an efficient reduction of prefix parsing to ordinary parsing. Given a grammar, our method constructs another grammar that generates exactly the prefixes of its original strings. Prefix parsing is then solved by applying any ordinary parsing algorithm on the transformed grammar without modification. The reduction is both elegant and practical: the transformed grammar is only a small factor larger than the input, and any optimized implementation can be used directly, eliminating the need for bespoke prefix-parsing algorithms. We also present a strategy—based on algorithmic differentiation—for computing the next-token weight vector, i.e., the prefix weights of *all* one-token extensions, enabling efficient prediction of the next token. Together, these contributions yield a simple, general, and efficient framework for prefix parsing.

genlm/prefix-acl-26

1 Introduction

Parsing a string with respect to a context-free grammar (CFG) means determining whether—and with what weight—the grammar generates that string. *Prefix* parsing asks the closely related question of whether an input prefix can be extended to a complete string generated by the grammar. In the probabilistic setting,[1] these tasks correspond, respectively, to computing (i) the probability that the grammar generates a given string, and (ii) the total probability of strings beginning with a given prefix.

Existing prefix parsers are typically tied to particular parsing algorithms. For example, Jelinek and Lafferty (1991) adapted CKY (Cocke and Schwartz, 1970; Kasami, 1965; Younger, 1967), while Stolcke (1995) adapted Earley's (1970) algorithm.[2] Although these algorithms require bespoke and surprisingly involved modifications to the base parser, they also share notable similarities. This raises a natural question: is there a general-purpose reduction from prefix parsing to parsing, one that would let us adapt *other* parsers to the prefix setting, e.g., Valiant's (1975) subcubic parsing algorithm, GPU implementations,[3] and fast CPU parsers?[4] We show that such a reduction exists. Given any CFG, we produce a new CFG—the *prefix grammar*—that generates exactly the weighted prefix language of the original, and is only a small constant factor larger. Consequently, *any* parsing algorithm serves as a prefix parser off-the-shelf: its input-length dependence is preserved, with the entire overhead confined to grammar-structural factors (Theorem 2). How much those factors grow depends on the parser's preprocessing pipeline; we analyze them for `CKY` and `Earley` (App. E).[5] Moreover, the prefix grammar can be viewed as the composition of the original grammar with a two-state prefix transducer (App. D), a perspective that extends naturally to other grammar formalisms.

Finally, we develop efficient algorithms for computing the *next-token weights* of a string—the prefix weights of all one-token extensions. Surprisingly, our strategy—which leverages algorithmic differentiation (see, e.g., Griewank and Walther, 2008; Eisner, 2016) and is closely related to the outside algorithm (Baker, 1979)—allows us to compute the next-token weights of a string in the same asymptotic runtime as prefix-parsing the string itself. On the other hand, a naïve approach would incur a multiplicative factor of $|\Sigma|$, corresponding to $|\Sigma|$

[1]More generally, our method applies to grammars whose weights lie in any commutative semiring (see, e.g., Goodman, 1999). This includes the boolean semiring, which is useful for enforcing grammaticality constraints in language models (e.g., Shin et al., 2021; Poesia et al., 2022; Loula et al., 2025).

[2]Nowak and Cotterell (2023) developed a prefix parser based on CKY with the *hook trick* (Eisner and Blatz, 2007).

[3]E.g., Stanojević and Sartran (2023), Rush (2020), Yi et al. (2011), Dunlop (2014), and Canny et al. (2013).

[4]E.g., Kegler (2023) and Dunlop (2014).

[5]We found that our implementation of Earley's algorithm with the prefix grammar is comparable to Luong et al.'s (2013) implementation of Stolcke's algorithm (App. B).

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 286–309
July 2-7, 2026 ©2026 Association for Computational Linguistics

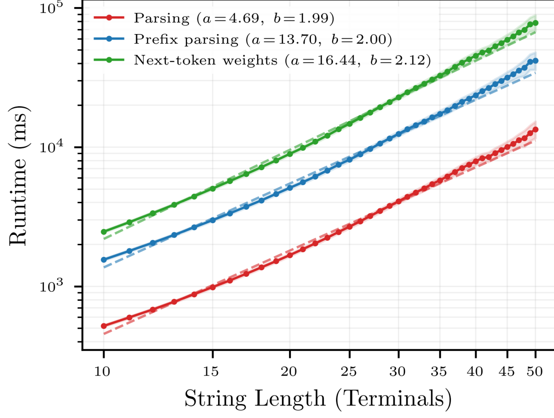


Figure 1: We compare parsing, prefix parsing, and the next-token weight vector algorithm on WSJ 5000 (Luong et al., 2013, `EarleyX`), a large grammar with 35,016 rules, 448 nonterminals, and a total size of 116,667. The underlying parser is Earley's (see Alg. 4), paired with its next-token variant (§4 and App. F.2). Results are shown for 500 strings on a log–log plot; to obtain data across all string lengths, we include the runtime for each prefix of every string. Shaded regions represent 95% confidence intervals for the mean runtime at each string length. To estimate runtime complexity, we fit a power law $r(N) = aN^b$ via least-squares regression on the log–log scale: $\log r(N) = \log a + b \log N$. Dashed lines show the fitted models. All three algorithms exhibit similar fitted complexity exponents (1.99, 2, and 2.12 respectively), confirming that the N-dependence is essentially independent of grammar size. Prefix parsing is roughly 2.9× slower than parsing (fitted coefficient $a = 30.9$ vs. 12.1), closely tracking the prefix grammar's $\approx 2.86\times$ size increase (binarize, then prefix). Both observations are predicted by Theorem 2: the N-dependence is preserved, and the overhead is absorbed into the grammar-size factor. The next-token algorithm is slower than prefix parsing by only a factor of ≈ 1.2, well within the constant-factor bound (i.e., 4×) of the next-token meta-theorem (Theorem 3).

calls to a prefix parser, where Σ is the token alphabet. Importantly, next-token weights arise in many applications, including context-free language modeling (Jelinek and Lafferty, 1991), surprisal-based psycholinguistic modeling (Hale, 2001), and syntactically constrained generation from LLMs (Shin et al., 2021; Poesia et al., 2022; Loula et al., 2025).

2 Preliminaries[6]

Basics. Let Σ be an **alphabet**, i.e., a finite set of symbols, and let Σ^* denote the **set of all strings** formed from the symbols of Σ, including the empty string ε. We write $\boldsymbol{s} \preceq \boldsymbol{t}$ if $\boldsymbol{s}$ is a prefix of $\boldsymbol{t}$, and $\boldsymbol{st}$ to denote concatenation. Let $\langle \mathbb{W}, +, \cdot, 0, 1\rangle$ be a commutative semiring, i.e., a set equipped with addition and multiplication satisfying the usual distributive and identity laws. For concreteness, the reader may take $\mathbb{W} = \mathbb{R}_{\geq 0}$. A **weighted language** is a function $\omega\colon \Sigma^* \to \mathbb{W}$. We call ω a **language model** when $\mathbb{W} = \mathbb{R}_{\geq 0}$ and $\sum_{\boldsymbol{s}\in\Sigma^*} \omega(\boldsymbol{s}) = 1$.

[6]For convenience, we provide a notation glossary in App. A.

Prefix languages. A central concept in this paper is the (weighted) **prefix language** of ω:

$$\overrightarrow{\omega}(\boldsymbol{s}) \overset{\text{def}}{=} \sum_{\boldsymbol{t}\in\Sigma^*} \mathbb{1}\{\boldsymbol{s} \preceq \boldsymbol{t}\}\, \omega(\boldsymbol{t}) \tag{1}$$

In the language model setting where $\mathbb{W} = \mathbb{R}_{\geq 0}$, the weight $\omega(\boldsymbol{s})$ of any string $\boldsymbol{s}$ factorizes as a product of conditional prefix probabilities:

$$\omega(\boldsymbol{s}) = \overrightarrow{\omega}(\text{EOS} \mid \boldsymbol{s}) \prod_{t=1}^{|\boldsymbol{s}|} \overrightarrow{\omega}(s_t \mid \boldsymbol{s}_{<t}) \tag{2}$$

where $\boldsymbol{s}_{<t} \overset{\text{def}}{=} s_1 \cdots s_{t-1}$, EOS $\notin \Sigma$ is a distinguished **end-of-string symbol**, and the **conditional next-token weights** are defined as follows:

$$\overrightarrow{\omega}(s_t \mid \boldsymbol{s}_{<t}) \overset{\text{def}}{=} \begin{cases} \frac{\omega(\boldsymbol{s}_{<t})}{\overrightarrow{\omega}(\boldsymbol{s}_{<t})} & \textbf{if } s_t = \text{EOS} \\ \frac{\overrightarrow{\omega}(\boldsymbol{s}_{<t} s_t)}{\overrightarrow{\omega}(\boldsymbol{s}_{<t})} & \textbf{otherwise} \end{cases} \tag{3}$$

provided that $\overrightarrow{\omega}(\boldsymbol{s}_{<t}) > 0$ and $s_t \in \Sigma \cup \{\text{EOS}\}$. Note that $\overrightarrow{\omega}(\cdot \mid \boldsymbol{s}_{<t})$ is a distribution over $\Sigma \cup \{\text{EOS}\}$. Conditional prefix probabilities underlie context-free language modeling and surprisal-based psycholinguistic analysis. In §4, we develop efficient algorithms for computing the prefix weights of all single-symbol extensions simultaneously.

A **weighted context-free grammar**[7] (**WCFG**, or simply **CFG**) $\mathcal{G}$ is a tuple $\langle \mathcal{N}, \Sigma, \text{S}, \mathcal{R}\rangle$, where $\mathcal{N}$ is a set of **nonterminal** symbols, Σ is an alphabet of **terminal** symbols (disjoint from $\mathcal{N}$), $\text{S} \in \mathcal{N}$ is a distinguished **start symbol**, and $\mathcal{R}$ is a finite set of weighted **rules**.[8] Each rule $r \in \mathcal{R}$ is of the form $\text{X} \xrightarrow{w} \boldsymbol{\alpha}$ where $\text{X} \in \mathcal{N}$, $\boldsymbol{\alpha} \in (\Sigma \cup \mathcal{N})^*$, and $w \in \mathbb{W}$. The **arity** of a rule r, denoted $\mathsf{ar}(r)$, is the length of its right-hand side. We say that a rule is *nullary* if its arity is zero, *unary* if its arity is one, *binary* if its arity is two, and so on. The **size** of a grammar is $|\mathcal{G}| = \sum_{r\in\mathcal{R}} 1 + \mathsf{ar}(r)$. We say that a grammar is in **canonical two-form** (**CTF**) if each of its rules has a right-hand side of length at most two, and **Chomsky normal form** (**CNF**) if each of its rules takes one of the forms $\text{X}\to\text{Y}\text{Z}$, $\text{X}\to a$, or $\text{S}\to\varepsilon$ for $\text{X}, \text{Y}, \text{Z} \in \mathcal{N}$, $a \in \Sigma$.

A **derivation tree** is a rooted, ordered tree in which each node is labeled with a symbol in $\mathcal{N} \cup \Sigma$, each nonleaf node is connected to its children by

[7]We assume the reader is familiar with WCFGs; for completeness, App. C.1 provides the necessary background.

[8]Without loss of generality, any two rules with the same left-hand side X and right-hand side $\boldsymbol{\alpha}$, $\text{X}\xrightarrow{w}\boldsymbol{\alpha}$ and $\text{X}\xrightarrow{w'}\boldsymbol{\alpha}$, can be consolidated into the single rule $\text{X}\xrightarrow{w+w'}\boldsymbol{\alpha}$.

a rule in $\mathcal{R}$, and each leaf is either a terminal or a childless nonterminal (arising from a nullary rule). The **weight** of a derivation tree is the product of the weights of all rules applied in the tree. Let $\mathcal{D}_\alpha(\boldsymbol{s})$ denote the set of derivation trees rooted at $\alpha \in \mathcal{N} \cup \Sigma$ whose **yield** is $\boldsymbol{s}$. The **weighted language** of α is $[\![\mathcal{G}_\alpha]\!](\boldsymbol{s}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{d} \in \mathcal{D}_\alpha(\boldsymbol{s})} \boldsymbol{w}(\boldsymbol{d})$, and the **weighted language** of $\mathcal{G}$ is $[\![\mathcal{G}]\!](\boldsymbol{s}) \stackrel{\text{def}}{=} [\![\mathcal{G}_{\mathrm{S}}]\!](\boldsymbol{s})$. The **total weight** of α is $\tau(\alpha) \stackrel{\text{def}}{=} \sum_{\boldsymbol{s} \in \Sigma^*} [\![\mathcal{G}_\alpha]\!](\boldsymbol{s})$. A (tight) **probabilistic context-free grammar** (**PCFG**) is one where $\tau(\alpha) = 1$ for all $\alpha \in \mathcal{N} \cup \Sigma$. If $\mathcal{G}$ is a PCFG, then $[\![\mathcal{G}]\!]$ is a language model (Booth and Thompson, 1973; Chi, 1999).

3 Prefix Parsing is Just Parsing

This section introduces the **prefix grammar transformation**, which allows us to turn any CFG parser into a *prefix* parser simply by transforming its grammar into one that generates its prefix language.[9,10]

Definition 1. *Given a grammar $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$, its **prefix grammar** is*

$$\overrightarrow{\mathcal{G}} \stackrel{\text{def}}{=} \langle \mathcal{N} \cup \{\mathrm{X}'\}_{\mathrm{X} \in \mathcal{N}} \cup \{\overrightarrow{\mathrm{S}}\}, \Sigma, \overrightarrow{\mathrm{S}}, \mathcal{R} \cup \mathcal{R}' \rangle$$

where α' denotes α itself if $\alpha \in \Sigma$ and a new prime *nonterminal if $\alpha \in \mathcal{N}$. Here $\overrightarrow{\mathrm{S}}$ is a new start symbol, and the additional rules $\mathcal{R}'$ are defined below:*

$$\overrightarrow{\mathrm{S}} \xrightarrow{1} \mathrm{S}' \tag{4a}$$

$$\overrightarrow{\mathrm{S}} \xrightarrow{\tau(\mathrm{S})} \varepsilon \tag{4b}$$

$$\mathrm{X}' \xrightarrow{w \cdot \tau(\alpha_{k+1}) \cdots \tau(\alpha_K)} \alpha_1 \cdots \alpha_{k-1}\, \alpha'_k \tag{4c}$$

$$\textit{for } \mathrm{X} \xrightarrow{w} \alpha_1 \cdots \alpha_K \in \mathcal{R},\ k \in 1, \ldots, K$$

The idea underlying the prefix grammar is that, at each level of the derivation tree, a prefix of a derived string must end inside the yield of exactly one rule application. This is captured by Eq. (4c): for each original rule $\mathrm{X} \xrightarrow{w} \alpha_1 \cdots \alpha_K$ and each border position k, the symbols $\alpha_1, \ldots, \alpha_{k-1}$ before the border are parsed normally, the symbol α_k at the border is only partially matched (handled recursively via α'_k), and the symbols $\alpha_{k+1}, \ldots, \alpha_K$ after the border are summarized by their total weights $\tau(\alpha_{k+1}) \cdots \tau(\alpha_K)$. Recall that $\tau(\alpha) = 1$ for PCFGs (§2); for general WCFGs, the total weights can be computed as described in App. C.1.

The following proposition establishes that Def. 1 correctly encodes the prefix language.

Proposition 1. *Let $\mathcal{G}$ be a CFG, and $\overrightarrow{\mathcal{G}}$ be its prefix grammar. Then, $\overrightarrow{\mathcal{G}}$ correctly encodes the prefix language of $\mathcal{G}$ (i.e., $[\![\overrightarrow{\mathcal{G}}]\!] = \overrightarrow{[\![\mathcal{G}]\!]}$).*

Proof. See App. G.1. ∎

This provides a straightforward reduction of any ordinary parsing algorithm to a prefix-parsing one. We say $\mathfrak{p}$ is a **correct parsing algorithm** if $\mathfrak{p}(\mathcal{G}, \boldsymbol{s}) = [\![\mathcal{G}]\!](\boldsymbol{s})$ for all grammars $\mathcal{G}$ and strings $\boldsymbol{s}$. Although this framing treats $\mathfrak{p}$ as a black box over arbitrary CFGs, most parsers in the literature are in fact defined only on a restricted class of grammars (e.g., CNF, nullary-free) and rely on an upstream **normal-form conversion** ϕ (e.g., CNF conversion for CKY) that is typically treated implicitly; its size cost is frequently understated and propagates through our reduction—we track it explicitly in Theorem 2. The following theorem shows that our reduction yields a correct prefix-parsing algorithm.

Theorem 1 (Prefix parsing is just parsing)**.** *Let $\mathcal{G}$ be a CFG over the alphabet Σ. Let $\overrightarrow{\mathcal{G}}$ be its prefix grammar. If $\mathfrak{p}$ is a correct parsing algorithm, then $\mathfrak{p}(\overrightarrow{\mathcal{G}}, \boldsymbol{s}) = \overrightarrow{[\![\mathcal{G}]\!]}(\boldsymbol{s})$ for all $\boldsymbol{s} \in \Sigma^*$.*

Proof. The theorem immediately follows from Prop. 1 and the premises. ∎

In App. E, we describe specific instantiations of this approach with `CKY` and `Earley`. The key to the efficiency of our reduction is that we can bound the size of the prefix grammar. For efficiency, it is advisable to binarize the grammar (i.e., convert to canonical two-form) before applying the prefix grammar transformation. Although binarization triples the grammar size, it bounds the rule arity at two, which in turn keeps the prefix grammar within a small constant factor of the original:

Proposition 2. *Let $\mathcal{G}$ be a context-free grammar in canonical two-form and let $\overrightarrow{\mathcal{G}}$ be its prefix grammar (Def. 1). Then, the size of $\overrightarrow{\mathcal{G}}$ is bounded by*

$$|\overrightarrow{\mathcal{G}}| \leq \frac{8}{3}|\mathcal{G}| + 3 \tag{5}$$

Proof. See App. G.2. ∎

[9] Prefix grammars themselves are not novel. Indeed, deriving one is an exercise on context-free grammars in Ben-David's (2022) lecture notes. However, we were unable to find a *weighted* CFG version; much to our delight, that extension proved straightforward. In this light, our contribution is to demonstrate how the prefix grammar can be used to *reduce* prefix parsing to ordinary parsing (with weights).

[10] Note that the prefix grammar can also be defined by *composition* (Bar-Hillel et al., 1961; Pasti et al., 2023) of the input grammar and a specifically designed *prefix transducer*; we discuss this view in detail in App. D.

Prop. 2 bounds $|\overrightarrow{\mathcal{G}}|$ directly. However, as noted above, most parsers' runtime bounds involve a preprocessed grammar $\phi(\mathcal{G})$ (e.g., CNF conversion for CKY) and often depend on structural parameters beyond total size, such as $|\mathcal{N}|$, which grow differently under ϕ (App. C.2). The following theorem lifts Prop. 2 to parser runtime by allowing the runtime to be any polynomial in string length and a vector $\boldsymbol{\nu}$ of grammar-structural parameters.

Theorem 2. *Let $\mathfrak{p}$ be a parsing algorithm whose runtime after grammar preprocessing by ϕ satisfies*

$$\mathrm{time}\{\mathfrak{p}(\mathcal{G}, \boldsymbol{s})\} = \mathcal{O}(\textstyle\sum_{i=1}^{m} g_i(\boldsymbol{\nu}(\phi(\mathcal{G}))) \cdot h_i(|\boldsymbol{s}|))$$

where $\boldsymbol{\nu}$ maps a grammar $\mathcal{G}' = \langle \mathcal{N}', \Sigma', \mathrm{S}', \mathcal{R}' \rangle$ to a vector of its structural parameters (e.g., $\boldsymbol{\nu}(\mathcal{G}') = [|\mathcal{G}'|, |\mathcal{N}'|, |\mathcal{R}'|, |\Sigma'|]^\top$), and each g_i and h_i is a polynomial.[11] *Then,*

$$\mathrm{time}\{\mathfrak{p}(\overrightarrow{\mathcal{G}}, \boldsymbol{s})\} = \mathcal{O}(\textstyle\sum_{i=1}^{m} g_i(\boldsymbol{\nu}(\phi(\overrightarrow{\mathcal{G}}))) \cdot h_i(|\boldsymbol{s}|))$$

In particular, the $|\boldsymbol{s}|$-dependence of each term is unchanged; the entire overhead of prefix parsing is captured by replacing $\boldsymbol{\nu}(\phi(\mathcal{G}))$ with $\boldsymbol{\nu}(\phi(\overrightarrow{\mathcal{G}}))$.

Proof. Apply $\mathfrak{p}$'s runtime guarantee with input $\overrightarrow{\mathcal{G}}$ in place of $\mathcal{G}$. ∎

Observe that Theorem 2 makes no claim about the magnitude of $\mathrm{time}\{\mathfrak{p}(\mathcal{G}, \boldsymbol{s})\}$ itself; that is a property of the parser. For example, `CKY` runs in $\mathcal{O}(|\mathcal{G}| \cdot |\boldsymbol{s}|^3)$ on CNF grammars, which fits the theorem with ϕ set to CNF conversion. Because the unary-removal step of ϕ can inflate grammar size by a factor of $|\mathcal{N}|$ (App. C.2), applying `CKY` to $\overrightarrow{\mathcal{G}}$ runs in $\mathcal{O}(|\phi(\overrightarrow{\mathcal{G}})| \cdot |\boldsymbol{s}|^3)$—a factor of $|\mathcal{N}|$ looser than the bound for $\mathcal{G}$ (App. E.1). Earley's lighter preprocessing incurs less blowup (App. E.2).

4 Next-Token Weight Vector Algorithms

Given a string $\boldsymbol{s} = s_1 \cdots s_N$ and a CFG $\mathcal{G}$, we now study how to efficiently compute the **next-token weight vector** $\boldsymbol{\pi}_a(\boldsymbol{s}) = [\![\overrightarrow{\mathcal{G}}]\!](\boldsymbol{s}a)$, whose components give the prefix weight of each one-symbol extension $\boldsymbol{s}a$ for $a \in \Sigma$. This vector is the key ingredient for the incremental processing applications discussed in §1, including constrained generation with a PCFG. Importantly, by Prop. 1, computing the next-token weight vector reduces to ordinary parsing: $\boldsymbol{\pi}_a(\boldsymbol{s}) = [\![\overrightarrow{\mathcal{G}}]\!](\boldsymbol{s}a)$. Yet a naïve implementation would still require $|\Sigma|$ parser calls, one per $a \in \Sigma$, i.e., $\mathcal{O}(|\Sigma| \cdot \mathrm{time}\{\mathfrak{p}(\overrightarrow{\mathcal{G}}, \boldsymbol{s})\})$ time per next-token vector $\boldsymbol{\pi}(\boldsymbol{s})$.

A less naïve approach is to use an **incremental parsing algorithm**—one whose evaluation on a prefix $s_1 \cdots s_{N-1}$ leaves enough cached information to extend to $s_1 \cdots s_N$ cheaply. We refer to this cached information as the **prefix state** and assume it is managed by memoization, so that the same prefix is parsed at most once. The precise representation of the prefix state is parser-specific; for incremental CKY (Alg. 2 in App. E) it is the portion of the parse chart filled in for $s_1 \cdots s_{N-1}$. Non-incremental CKY on $\overrightarrow{\mathcal{G}}$ runs in $\mathcal{O}(|\phi(\overrightarrow{\mathcal{G}})| N^3)$; incremental CKY amortizes this cost across prefixes, extending the parse by one token in only $\mathcal{O}(|\phi(\overrightarrow{\mathcal{G}})| N^2)$. Immutability lets us safely reuse the prefix state across all $|\Sigma|$ candidate one-token extensions of $s_1 \cdots s_{N-1}$, as the cached entries from the prefix do not depend on the choice of next token. This reduces the baseline to $\mathcal{O}(|\phi(\overrightarrow{\mathcal{G}})| N^2 \cdot |\Sigma|)$ per token—an improvement by a factor of N—but the $|\Sigma|$ factor remains.

In contrast, we propose an approach that eliminates the $|\Sigma|$ factor entirely, computing the full vector $\boldsymbol{\pi}(\boldsymbol{s})$ in $\mathcal{O}(\mathrm{time}\{\mathfrak{p}(\overrightarrow{\mathcal{G}}, \boldsymbol{s})\})$ time per next-token vector—the same asymptotic cost as a single parser call—by combining two key ingredients, *lattice parsing* and *algorithmic differentiation*, which we introduce next. Our approach applies to any parser that supports lattice input and whose evaluation consists of semiring operations—mild requirements satisfied by standard algorithms, such as incremental CKY and Earley (see App. E). Moreover, when the base parser is incremental, our next-token algorithm inherits its per-prefix amortization.

An algorithmic differentiation reduction. We start by defining the aggregation function:

$$Z_{\boldsymbol{s}}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_{a \in \Sigma} \boldsymbol{\pi}_a(\boldsymbol{s}) \cdot \theta_a \tag{6}$$

where $\boldsymbol{\theta} \in \mathbb{W}^\Sigma$ is a vector of formal variables, introduced solely to enable differentiation. Because this sum is linear in $\boldsymbol{\theta}$, its gradient immediately yields the next-token weight vector:[12]

$$\nabla_{\boldsymbol{\theta}} Z_{\boldsymbol{s}}(\boldsymbol{\theta}) = \boldsymbol{\pi}(\boldsymbol{s}) \qquad \forall \boldsymbol{\theta} \in \mathbb{W}^\Sigma \tag{7}$$

This reduction is productive because there is an efficient method for computing $Z_{\boldsymbol{s}}(\boldsymbol{\theta})$—namely,

[11] We assume the one-time cost of ϕ is amortized over all strings parsed with the same grammar.

[12] The gradient here is *algebraic*: $\nabla[x + y] = \nabla[x] + \nabla[y]$ and $\nabla[x \cdot y] = \nabla[x] \cdot y + x \cdot \nabla[y]$, requiring only semiring operations (see, e.g., Esparza et al., 2008).

lattice parsing, which replaces the input string with a compact automaton encoding all $|\Sigma|$ one-symbol extensions at once. Moreover, thanks to algorithmic differentiation, we have the following *meta-theorem* that ensures the computation of the gradient $\nabla_{\boldsymbol{\theta}} Z_{\boldsymbol{s}}(\boldsymbol{\theta})$ is equally efficient.

Theorem 3 (Next-token meta-theorem)**.** *For any algorithm* $\mathfrak{q}$ *that correctly computes* $Z_{\boldsymbol{s}}(\boldsymbol{\theta})$ *and whose evaluation is a sequence of semiring operations,*[13] *we can use reverse-mode algorithmic differentiation to derive the gradient algorithm* $\nabla\mathfrak{q}$ *that correctly computes* $\nabla_{\boldsymbol{\theta}} Z_{\boldsymbol{s}}(\boldsymbol{\theta}) = \boldsymbol{\pi}(\boldsymbol{s})$ *using at most* 4 *times as many arithmetic operations as* $\mathfrak{q}$.

Proof. The proof follows from Eq. (7) and the cheap gradient principle (Griewank and Walther, 2008, Eq. 3.14), which bounds the gradient computation at $4\times$ the original function's operation count. Since this bound counts arithmetic operations irrespective of their interpretation, it carries over from $\mathbb{R}$ to any commutative semiring $\mathbb{W}$. ■

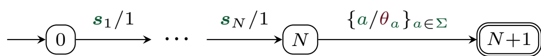
Lattice parsing. Next, we show how to efficiently compute $Z_{\boldsymbol{s}}(\boldsymbol{\theta})$ via lattice parsing. A **word lattice** (Hall, 2005) is an acyclic weighted finite-state automaton (WFSA; see App. C.3) representing a weighted set of strings. A **lattice parser** $\mathfrak{q}$ is an algorithm that, given a grammar $\mathcal{G}$ and a word lattice $\mathcal{L}$, computes the following product:[14]

$$\mathfrak{q}(\mathcal{G}, \mathcal{L}) = \sum_{\boldsymbol{s}\in\Sigma^*} [\![\mathcal{L}]\!](\boldsymbol{s})\cdot[\![\mathcal{G}]\!](\boldsymbol{s}) \tag{8}$$

That is, the lattice parser computes the inner product of the lattice's and grammar's weighted languages. Efficient lattice parsers can be derived by minimally modifying existing parsing algorithms like Earley and CKY (Hall, 2005).

Our approach to computing the *aggregation function* $Z_{\boldsymbol{s}}(\boldsymbol{\theta})$ is based on parsing the next-token lattice defined below.

Definition 2. *Let* $\boldsymbol{s} \in \Sigma^*$ *and* $\boldsymbol{\theta} \in \mathbb{W}^{\Sigma}$. *The* ***next-token lattice*** $\mathcal{L}_{\boldsymbol{s}}(\boldsymbol{\theta})$ *is the WFSA below:*



where the edge labeled $\{a/\theta_a\}_{a\in\Sigma}$ *is shorthand for a set of transitions—one for each symbol* $a \in \Sigma$.

Since the lattice accepts exactly the strings $\boldsymbol{s}a$ for $a \in \Sigma$, each weighted by θ_a, parsing it with the prefix grammar recovers the aggregation function:

$$\mathfrak{q}(\overrightarrow{\mathcal{G}}, \mathcal{L}_{\boldsymbol{s}}(\boldsymbol{\theta})) = Z_{\boldsymbol{s}}(\boldsymbol{\theta}) \tag{9}$$

Putting it together. We now combine the three ingredients—the prefix grammar (Def. 1), the next-token lattice (Def. 2), and the next-token meta-theorem (Theorem 3)—into a complete next-token weight vector algorithm. Let $\mathcal{G}$ be a CFG with prefix grammar $\overrightarrow{\mathcal{G}}$, and let $\boldsymbol{s} = s_1 \cdots s_N$ be a string. Suppose that $\mathfrak{q}$ is an algorithm whose evaluation consists of a sequence of semiring operations. Then, by the next-token meta-theorem (Theorem 3), there exists a gradient algorithm $\nabla\mathfrak{q}$ that computes the next-token weight vector $\boldsymbol{\pi}(\boldsymbol{s}) = \nabla_{\boldsymbol{\theta}} Z_{\boldsymbol{s}}(\boldsymbol{\theta})$ with the same asymptotic complexity as one evaluation of $\mathfrak{q}(\overrightarrow{\mathcal{G}}, \mathcal{L}_{\boldsymbol{s}}(\boldsymbol{\theta}))$. We present two concrete instantiations of this framework in the appendix: one based on CKY (App. F.1) and one based on Earley's algorithm (App. F.2). The empirical results displayed in Fig. 1—which use the Earley-based instantiation—confirm that computing $\boldsymbol{\pi}(\boldsymbol{s})$ is slower than prefix parsing by only a small constant factor. Finally, in many applications, we also need the total weight of a string $[\![\mathcal{G}]\!](\boldsymbol{s})$ alongside its prefix weight.[15]

Conclusion

We showed that prefix parsing is just ordinary parsing—applied to the prefix grammar, a compact CFG encoding of the prefix language. Given any CFG, we construct a new CFG whose ordinary parser computes prefix weights of the original. Unlike previous approaches that extend existing parsing algorithms with complex modifications, our reduction provides a unified recipe for prefix parsing. Additionally, using algorithmic differentiation, we provide the fastest known algorithm for computing the next-token weight vector. With our reduction in hand, there is no longer a need to develop bespoke prefix-parsing algorithms. A natural direction for future work is to develop a generalized notion of a *prefix transformation* that extends to other formalisms, such as context-sensitive grammars.

[13] Control flow (for-loops, if-statements) is permitted provided it does not depend on $\boldsymbol{\theta}$—this ensures the computed function is a polynomial in $\boldsymbol{\theta}$, so symbolic differentiation applies.

[14] Lattice parsers are routine extensions of ordinary parsers (Hall, 2005, §3.2); ordinary parsing is the special case where the lattice contains the single string $\boldsymbol{s}$ with weight one.

[15] Define $\mathcal{G}\,\textsc{eos} \overset{\text{def}}{=} \langle \mathcal{N}, \Sigma \cup \{\textsc{eos}\}, \mathrm{S}', \mathcal{R} \cup \{\mathrm{S}' \xrightarrow{1} \mathrm{S}\,\textsc{eos}\}\rangle$ where $\mathrm{S}' \notin \mathcal{N}$. Since $[\![\mathcal{G}\,\textsc{eos}]\!](\boldsymbol{s}\,\textsc{eos}) = [\![\mathcal{G}]\!](\boldsymbol{s})$, applying the next-token weight vector algorithm to the prefix grammar $\overrightarrow{\mathcal{G}\,\textsc{eos}}$ yields $[\![\mathcal{G}]\!](\boldsymbol{s})$ as the EOS component of the vector, alongside all other next-token weights. The transformation is efficient: it adds a single rule, so $|\mathcal{G}\,\textsc{eos}| = |\mathcal{G}| + 3$.

Limitations

The limitations of our approach are discussed in context throughout the main text; however, we would like to remind that one of the main limitations of our work is that, in the general case, Theorem 2 does not guarantee that a prefix parser has the same asymptotic runtime as the underlying base parser. Additionally, we would like to recall that our prefix parser works with any parser of choice, however we only ran experiments with CKY and Earley's.

Acknowledgments

We thank Jason Eisner and Franz Nowak for useful discussions. We also thank João Loula, Ben LeBrun, and Leo Du for early adoption of our method (see Loula et al., 2025). We used generative AI to improve our writing and to help debug the code. Every modification introduced by generative AI was carefully reviewed by the authors, which take full responsibility for it.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

J. K. Baker. 1979. Trainable grammars for speech recognition. In *Speech Communication Papers for the Meeting of the Acoustical Society of America*.

Yehoshua Bar-Hillel, Micha Asher Perles, and Eli Shamir. 1961. On formal properties of simple phrase-structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14.

Shalev Ben-David. 2022. Lecture 10: Closure properties for context-free languages (CS 360, Winter 2022).

Taylor L. Booth and Richard A. Thompson. 1973. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, 22(5).

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable transformations of Python+NumPy programs.

John Canny, David Hall, and Dan Klein. 2013. A multi-teraflop constituency parser using GPUs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Zhiyi Chi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1).

John Cocke and Jacob T. Schwartz. 1970. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata*. Springer.

Aaron Joseph Dunlop. 2014. *Efficient Latent-Variable Grammars: Learning and Inference*. Ph.D. thesis, Oregon Health & Science University.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2).

Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of the Workshop on Structured Prediction for NLP*.

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the Conference on Formal Grammar*.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing*.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2007. An extension of Newton's method to ω-continuous semirings. In *Developments in Language Theory*.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2008. Newton's method for ω-continuous semirings. In *Automata, Languages and Programming*.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4).

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives*. SIAM.

John Hale. 2001. A probabilistic Earley parser as a psycholinguistic model. In *Proceedings of the Meeting of the North American Chapter of the Association for Computational Linguistics on Language Technologies*.

Keith B. Hall. 2005. *Best-first Word-lattice Parsing: Techniques for Integrated Syntactic Language Modeling*. Ph.D. thesis, Brown University.

Laurent Hascoët and Valérie Pascual. 2013. The Tapenade automatic differentiation tool: Principles, model, and specification. *Association for Computing Machinery Transactions on Mathematical Software*, 39(3).

Frederick Jelinek and John D. Lafferty. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17(3).

Tadao Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory.

Jeffrey Kegler. 2023. Marpa, a practical general parser: the recognizer.

Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 8.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. 2025. Syntactic and semantic control of large language models via sequential Monte Carlo. In *Proceedings of the International Conference on Learning Representations*.

Minh-Thang Luong, Michael C. Frank, and Mark Johnson. 2013. Parsing entire discourses as very long strings: Capturing topic continuity in grounded language learning. *Transactions of the Association for Computational Linguistics*, 1.

Mark-Jan Nederhof. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics*, 25(3).

Mark-Jan Nederhof and Giorgio Satta. 2008. *Probabilistic Parsing*, chapter 7. Springer.

Franz Nowak and Ryan Cotterell. 2023. A fast algorithm for computing prefix probabilities. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Andreas Opedal, Anej Svete, Alexandra Butoi, Franz Nowak, and Ryan Cotterell. 2022. Lecture 7: Grammar transformations.

Andreas Opedal, Ran Zmigrod, Tim Vieira, Ryan Cotterell, and Jason Eisner. 2023. Efficient semiring-weighted Earley parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Clemente Pasti, Andreas Opedal, Tiago Pimentel, Tim Vieira, Jason Eisner, and Ryan Cotterell. 2023. On the intersection of context-free and regular languages. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics*.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Proceedings of the NeurIPS Autodiff Workshop*.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *Proceedings of the International Conference on Learning Representations*.

Alexander Rush. 2020. Torch-Struct: Deep structured prediction library. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Yves Schabes and Aravind K. Joshi. 1988. An Earley-type parsing algorithm for tree adjoining grammars. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Miloš Stanojević and Laurent Sartran. 2023. SynJax: Structured probability distributions for JAX. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2).

Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2).

Youngmin Yi, Chao-Yue Lai, Slav Petrov, and Kurt Keutzer. 2011. Efficient parallel CKY parsing on GPUs. In *Proceedings of the International Conference on Parsing Technologies*.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2).

Appendix Contents

A Notation Glossary **9**

B Additional Experimental Results **10**

C Additional Background **11**
- C.1 Weighted Context-Free Grammars . . . 11
- C.2 Grammar Transformations . . . 12
- C.3 Weighted Finite-State Automata . . . 13

D Deriving the Prefix Grammar by CFG–FST Composition **14**

E Instantiations of Prefix Parsing with the Prefix Grammar **16**
- E.1 Prefix Parsing with CKY . . . 16
- E.2 Prefix Parsing with Earley's Algorithm . . . 18

F Practical Instantiations of the Next-Token Weight Vector Algorithm **20**
- F.1 Next-Token Weight Vector with CKY . . . 20
- F.2 Next-Token Weight Vector with Earley's Algorithm . . . 20

G Deferred Proofs **22**
- G.1 Proof of Prop. 1 . . . 22
- G.2 Proof of Prop. 2 . . . 24

A Notation Glossary

Symbol	Description
Strings and languages (§2)	
Σ	alphabet (finite set of terminal symbols)
Σ^*	set of all strings over Σ
$\varepsilon \in \Sigma^*$	empty string
$\boldsymbol{s}, \boldsymbol{t} \in \Sigma^*$	strings
$s, a \in \Sigma$	terminal symbols
Σ^+	set of nonempty strings over Σ
$\preceq$	prefix relation ($\boldsymbol{s} \preceq \boldsymbol{t}$ iff $\boldsymbol{s}$ is a prefix of $\boldsymbol{t}$)
$\textsc{eos} \notin \Sigma$	end-of-string symbol
$\langle \mathbb{W}, +, \cdot, 0, 1 \rangle$	commutative semiring of weights
$\omega \colon \Sigma^* \to \mathbb{W}$	weighted language
$\overrightarrow{\omega} \colon \Sigma^* \to \mathbb{W}$	prefix language of ω
Grammars (§2, App. C.1)	
$\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$	weighted context-free grammar (WCFG)
$\mathcal{N}$	set of nonterminal symbols
$\mathrm{S} \in \mathcal{N}$	start symbol
$\mathcal{R}$	finite set of weighted rules
$\alpha \in \mathcal{N} \cup \Sigma$	grammar symbol (nonterminal or terminal)
$\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$	rule right-hand side
$r, (\mathrm{X} \xrightarrow{w} \boldsymbol{\alpha}) \in \mathcal{R}$	rule; X rewrites to $\boldsymbol{\alpha}$ with weight $w \in \mathbb{W}$
$\lvert\mathcal{G}\rvert \in \mathbb{N}$	grammar size
$\mathsf{ar}(r) \in \mathbb{N}$	arity of rule r
$\boldsymbol{d} \in \mathcal{D}$	derivation tree of $\mathcal{G}$
$\boldsymbol{d} \in \mathcal{D}_\alpha \subseteq \mathcal{D}$	α-rooted derivation tree
$\mathcal{D}_\alpha(\boldsymbol{s}) \subseteq \mathcal{D}_\alpha$	set of derivation trees rooted at α with yield $\boldsymbol{s}$
$\sigma(\boldsymbol{d}) \in \Sigma^*$	yield of derivation tree $\boldsymbol{d}$
$\boldsymbol{w}(\boldsymbol{d}) \in \mathbb{W}$	weight of derivation tree $\boldsymbol{d}$
$[\![\mathcal{G}]\!](\boldsymbol{s}) \in \mathbb{W}$	weighted language of $\mathcal{G}$: total weight of $\boldsymbol{s}$
$\tau(\alpha) \in \mathbb{W}$	total weight of symbol $\alpha \in \Sigma \cup \mathcal{N}$
Prefix grammar (§3)	
$\overrightarrow{\mathcal{G}}$	prefix grammar of $\mathcal{G}$
X'	prime nonterminal (partially matched $\mathrm{X} \in \mathcal{N}$, $\mathrm{X}' \notin \mathcal{N}$)
$\overrightarrow{\mathrm{S}}$	prefix start symbol ($\overrightarrow{\mathrm{S}} \neq \mathrm{S}$, $\overrightarrow{\mathrm{S}} \notin \mathcal{N}$)
Parsers and preprocessing (§3, §4)	
ϕ	grammar preprocessing function (e.g., `EnsureCNF`; see App. C.2)
$\mathfrak{p}$	abstract parsing algorithm
$\mathfrak{q}$	abstract lattice parser
Next-token algorithms (§4)	
$\boldsymbol{\pi}(\boldsymbol{s}) \in \mathbb{W}^\Sigma$	next-token weight vector
$\boldsymbol{\pi}_a(\boldsymbol{s}) \in \mathbb{W}$	next-token weight for terminal a
$\boldsymbol{\theta} \in \mathbb{W}^\Sigma$	formal parameter vector
$\theta_a \in \mathbb{W}$	parameter for terminal a
$Z_{\boldsymbol{s}}(\boldsymbol{\theta}) \in \mathbb{W}$	aggregation function
$\mathcal{L}$	word lattice (acyclic WFSA)
$\mathcal{L}_{\boldsymbol{s}}(\boldsymbol{\theta})$	next-token lattice
$z(i, \mathrm{X}) \in \mathbb{W}$	forward value
$\nabla z(i, \mathrm{X}) \in \mathbb{W}$	backward value
$\beta_k(i, \mathrm{X}) \in \mathbb{W}$	inside weight

B Additional Experimental Results

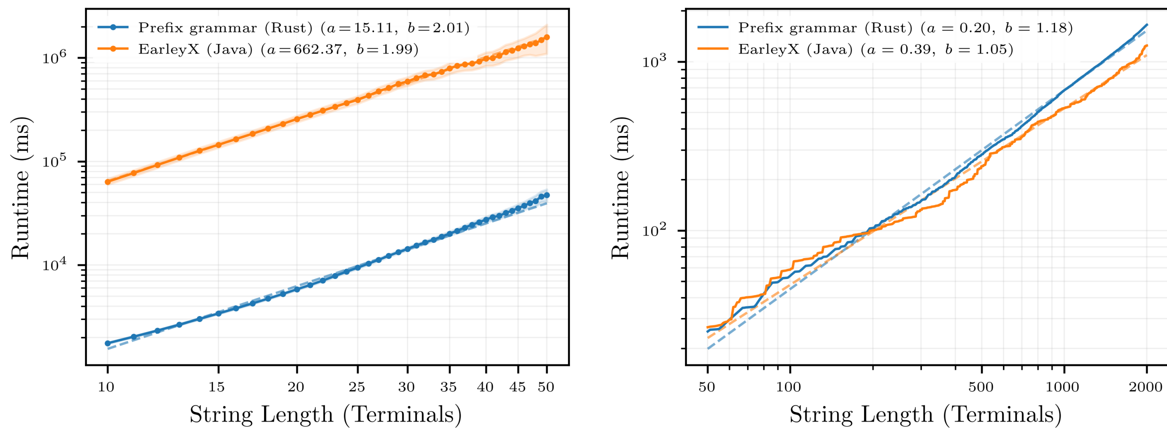


Figure 2: **Left: Our prefix parser vs. EarleyX** (Luong et al., 2013)**.** We compare the two implementations on WSJ 5000 and 500 sentences. Our algorithm uses Earley's with the prefix grammar (Alg. 3), while EarleyX uses Stolcke's (1995) algorithm to compute prefix probabilities (itself an adaptation of Earley's). As in Fig. 1, we fit a power law $r(N) = aN^b$ via least-squares regression on the log–log scale. The fitted slopes are nearly identical (2.01 vs. 1.99), while the fitted coefficient is substantially better in our implementation. This is consistent with the fact that Stolcke's algorithm pays an additional multiplicative factor of $|\mathcal{R}|$, which our Earley implementation avoids via an optimization at the completion step (Alg. 4). **Right: Sparse grammar comparison.** We repeat the experiment on the *Social Discourse* grammar (Luong et al., 2013, `EarleyX`), a sparse grammar with 72,712 rules and 233 nonterminals. The graph reports the runtime for each prefix of a sequence of concatenated sentences (`socialall.ortho.yld.concat`). Both parsers achieve near-linear time on this sparse grammar (fitted exponents 1.05 for EarleyX and 1.18 for our implementation). For this experiment, we used the specialized `EarleyParserSparse` from Luong et al. (2013), which is highly optimized for sparse grammars.

C Additional Background

C.1 Weighted Context-Free Grammars

Let $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$ be a CFG. We develop the weighted language $[\![\mathcal{G}]\!]$ (and a few other useful concepts) by first defining (weighted) derivation trees, then summing over trees with a given yield. A **derivation tree $\boldsymbol{d}$** is a rooted, ordered tree satisfying: (i) each node is labeled with a symbol in $\mathcal{N} \cup \Sigma$, (ii) each nonleaf node is connected to its children by a rule in $\mathcal{R}$, and (iii) each leaf is either a terminal or a childless nonterminal. The **set of all derivation trees** is defined recursively as the smallest set $\mathcal{D}$ satisfying:

$$\mathcal{D} \overset{\text{def}}{=} \Sigma \cup \left\{ \begin{array}{c} \mathrm{X} \\ \alpha_1 \cdots \alpha_K \\ \triangle \quad \triangle \end{array} \;\middle|\; \mathrm{X} \to \alpha_1 \cdots \alpha_K \in \mathcal{R}, \begin{array}{c} \alpha_1 \\ \triangle \end{array}, \ldots, \begin{array}{c} \alpha_K \\ \triangle \end{array} \in \mathcal{D} \right\}$$

Note that the subtrees rooted at the internal nodes of $\boldsymbol{d}$ are themselves derivation trees. Let $\rho(\boldsymbol{d})$ denote the root label of $\boldsymbol{d}$, and we write $\mathcal{D}_\alpha \overset{\text{def}}{=} \{\boldsymbol{d} \in \mathcal{D} \colon \rho(\boldsymbol{d}) = \alpha\}$ to denote the derivations with root $\alpha \in \Sigma \cup \mathcal{N}$. The **yield** $\sigma(\boldsymbol{d})$ of a tree $\boldsymbol{d}$ is recursively defined as

- **if $\boldsymbol{d} \in \Sigma$:** $\sigma(\boldsymbol{d}) \overset{\text{def}}{=} \boldsymbol{d}$ (10a)
- **otherwise:** $\sigma\left(\begin{array}{c} \mathrm{X} \\ \alpha_1 \cdots \alpha_K \\ \triangle \quad \triangle \end{array} \right) \overset{\text{def}}{=} \sigma\left(\begin{array}{c} \alpha_1 \\ \triangle \end{array} \right) \cdots \sigma\left(\begin{array}{c} \alpha_K \\ \triangle \end{array} \right)$ (10b)

Note that the yield $\sigma(\boldsymbol{d})$ of a childless node is ε. A **derivation forest** is a set of derivation trees; its weight is the sum of the weights of its trees. We write $\mathcal{D}(\boldsymbol{s}) \overset{\text{def}}{=} \{\boldsymbol{d} \in \mathcal{D} \colon \sigma(\boldsymbol{d}) = \boldsymbol{s}\}$ for the derivation forest with fixed yield $\boldsymbol{s}$, and $\mathcal{D}_\alpha(\boldsymbol{s}) \overset{\text{def}}{=} \mathcal{D}_\alpha \cap \mathcal{D}(\boldsymbol{s})$ for the forest with root α and yield $\boldsymbol{s}$. The **weight $\boldsymbol{w}(\boldsymbol{d})$** of the derivation tree $\boldsymbol{d}$ is defined as

- **if $\boldsymbol{d} \in \Sigma$:** $\boldsymbol{w}(\boldsymbol{d}) \overset{\text{def}}{=} 1$ (11a)
- **otherwise:** $\boldsymbol{w}\left(\begin{array}{c} \mathrm{X} \\ \alpha_1 \cdots \alpha_K \\ \triangle \quad \triangle \end{array} \right) \overset{\text{def}}{=} \boldsymbol{w}(\mathrm{X} \to \alpha_1 \cdots \alpha_K) \cdot \boldsymbol{w}\left(\begin{array}{c} \alpha_1 \\ \triangle \end{array} \right) \cdots \boldsymbol{w}\left(\begin{array}{c} \alpha_K \\ \triangle \end{array} \right)$ (11b)

For every $\alpha \in \Sigma \cup \mathcal{N}$, we define its **weighted language** as

$$[\![\mathcal{G}_\alpha]\!](\boldsymbol{s}) \overset{\text{def}}{=} \sum_{\boldsymbol{d} \in \mathcal{D}_\alpha(\boldsymbol{s})} \boldsymbol{w}(\boldsymbol{d}) \tag{12}$$

Every CFG defines a weighted language $[\![\mathcal{G}]\!]$ by taking the language of the root symbol:

$$[\![\mathcal{G}]\!](\boldsymbol{s}) \overset{\text{def}}{=} [\![\mathcal{G}_{\mathrm{S}}]\!](\boldsymbol{s}) \tag{13}$$

For each symbol $\alpha \in \Sigma \cup \mathcal{N}$, we define $\tau(\alpha)$ as the **total weight** of all α-rooted derivations:

$$\tau(\alpha) = \sum_{\boldsymbol{d} \in \mathcal{D}_\alpha} \boldsymbol{w}(\boldsymbol{d}) \tag{14}$$

The total weights satisfy the following system of polynomial equations:

$$\tau(\alpha) = \begin{cases} 1 & \text{for } \alpha \in \Sigma \\ \sum_{(\alpha \xrightarrow{w} \alpha_1 \cdots \alpha_M) \in \mathcal{R}} w \cdot \tau(\alpha_1) \cdots \tau(\alpha_M) & \text{for } \alpha \in \mathcal{N} \end{cases} \tag{15}$$

The total weights are well-defined in any ω-continuous semiring (Droste et al., 2009); examples include the nonnegative extended reals. They can be approximated, e.g., by fixed-point iteration or generalized Newton's method (Nederhof and Satta, 2008; Esparza et al., 2007).

C.2 Grammar Transformations

A grammar transformation is a function ϕ that maps a CFG $\mathcal{G}$ to another CFG $\phi(\mathcal{G})$. We say that ϕ is *semantics preserving* if $[\![\mathcal{G}]\!] = [\![\phi(\mathcal{G})]\!]$. Many transformations used to preprocess grammars are semantics preserving, including **dead-rule elimination**, **binarization**, **nullary removal**, **unary removal**, and **terminal separation**. The purpose of most of these transformations is self-explanatory to a reader familiar with CFGs; we refer the reader to Opedal et al. (2022) for detailed descriptions. All transformations discussed in this section are included in our code release. Converting a grammar into CNF is itself a (composed) transformation `EnsureCNF`: apply `EnsureCTF`, then `EnsureNullaryFree`, then `EnsureUnaryFree`, then `EnsureTerminalSep`.[16] In general, a grammar transformation ϕ can change both the size $|\phi(\mathcal{G})|$ of the transformed grammar and the number of nonterminals $|\mathcal{N}_\phi|$. We bound both for the transformations above:

- Dead rule elimination:

$$|\phi(\mathcal{G})| \leq |\mathcal{G}|, \qquad |\mathcal{N}_\phi| \leq |\mathcal{N}| \tag{16}$$

- `EnsureCTF` (binarization):

$$|\phi(\mathcal{G})| \leq 3|\mathcal{G}|, \qquad |\mathcal{N}_\phi| \leq |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}| \tag{17}$$

  The $|\mathcal{G}| - |\mathcal{R}|$ term accounts for intermediate nonterminals introduced when splitting rules of arity > 2.
- `EnsureNullaryFree` (nullary removal), provided $\mathcal{G}$ is in CTF:

$$|\phi(\mathcal{G})| \leq \tfrac{7}{3}|\mathcal{G}| + 3, \qquad |\mathcal{N}_\phi| \leq |\mathcal{N}| + 1 \tag{18}$$

- `EnsureUnaryFree` (unary removal):

$$|\phi(\mathcal{G})| \leq |\mathcal{N}| \cdot |\mathcal{G}|, \qquad |\mathcal{N}_\phi| \leq |\mathcal{N}| \tag{19}$$

- `EnsureUnaryCycleFree` (unary cycle removal) from Opedal et al. (2023):

$$|\phi(\mathcal{G})| \leq |\mathcal{G}| + 2|\mathcal{N}|^2, \qquad |\mathcal{N}_\phi| \leq 2|\mathcal{N}| \tag{20}$$

  The $|\mathcal{N}|^2$ term bounds the closure of the unary-rule graph; it is tight in the worst case (a single strongly connected component) and often far smaller in practice.
- `EnsureTerminalSep` (terminal separation):

$$|\phi(\mathcal{G})| \leq |\mathcal{G}| + 2|\Sigma|, \qquad |\mathcal{N}_\phi| \leq |\mathcal{N}| + |\Sigma| \tag{21}$$

- `EnsureCNF` (composing the above in the order `EnsureCTF`, `EnsureNullaryFree`, `EnsureUnaryFree`, `EnsureTerminalSep`). Tracing the pipeline:
  1. `EnsureCTF`: $|G_1| \leq 3|\mathcal{G}|,\ |N_1| \leq |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|$ (Eq. (17))
  2. `EnsureNullaryFree`: $|G_2| \leq \tfrac{7}{3}|G_1| + 3 \leq 7|\mathcal{G}| + 3,\ |N_2| \leq |N_1| + 1$ (Eq. (18))
  3. `EnsureUnaryFree`: $|G_3| \leq |N_2| \cdot |G_2| \leq (|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}| + 1)(7|\mathcal{G}| + 3),\ |N_3| \leq |N_2|$ (Eq. (19))
  4. `EnsureTerminalSep`: $|G_4| \leq |G_3| + 2|\Sigma|,\ |N_4| \leq |N_3| + |\Sigma|$ (Eq. (21))

  Summing yields:

$$\begin{aligned} |\phi(\mathcal{G})| &\leq (|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}| + 1)(7|\mathcal{G}| + 3) + 2|\Sigma| \\ |\mathcal{N}_\phi| &\leq |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}| + 1 + |\Sigma| \end{aligned} \tag{22}$$

  The dominant factor is unary removal; the $|\mathcal{G}|$ term in the first factor arises because binarization introduces up to $|\mathcal{G}| - |\mathcal{R}|$ intermediate nonterminals. See Lange and Leiß (2009) for a detailed analysis of CNF conversion orderings and their worst-case blowup.

[16] Note that the order in which these transformations are applied matters. For example, applying `EnsureNullaryFree` before `EnsureCTF` can cause exponential blowup, as a rule of arity K may spawn up to 2^K variants (Lange and Leiß, 2009). Placing terminal separation last prevents subsequent multiplicative transformations from amplifying its additive cost.

C.3 Weighted Finite-State Automata

A WFSA $\mathcal{A}$ is a tuple $\langle \Sigma, Q, \delta, \lambda, \rho \rangle$, where Σ is an alphabet, Q is a finite set of states, δ is a set of edges (each having form $q \xrightarrow{a/w} p$, with $p, q \in Q$, $w \in \mathbb{W}$ and $a \in \Sigma$), $\lambda \colon Q \to \mathbb{W}$ and $\rho \colon Q \to \mathbb{W}$ are respectively the initial and final weight functions. A **path** π is a sequence of consecutive edges

$$q_0 \xrightarrow{a_1/w_1} q_1 \xrightarrow{a_2/w_2} q_2 \cdots q_{N-1} \xrightarrow{a_N/w_N} q_N \tag{23}$$

the path's weight is defined as $\boldsymbol{w}(\pi) = \lambda(q_0) \cdot w_1 \cdot w_2 \cdots w_N \cdot \rho(q_N)$, the path's yield is $\sigma(\pi) = a_1 \cdots a_N$. We denote with $\Pi_{\mathcal{A}}(\boldsymbol{s})$ the set of paths with yield $\boldsymbol{s}$. A WFSA defines the weighted language:

$$[\![\mathcal{A}]\!](\boldsymbol{s}) = \sum_{\pi \in \Pi_{\mathcal{A}}(\boldsymbol{s})} \boldsymbol{w}(\pi) \tag{24}$$

D Deriving the Prefix Grammar by CFG–FST Composition

This section explains how we initially *derived* the prefix grammar[17] by composition with a weighted finite-state transducer. Although Def. 1 is a more succinct encoding of the prefix grammar, we believe the derivation as a composition is of independent interest. We begin by reviewing weighted relations and weighted finite-state transducers, then introduce the **prefix transducer** Δ.

Let Σ and Ξ be alphabets. A **weighted relation** between the strings Σ^* and Ξ^* is a function $t\colon \Sigma^* \times \Xi^* \to \mathbb{W}$. We define the **composition** of t with the language $\omega\colon \Sigma^* \to \mathbb{W}$, as the language $\omega \circ t\colon \Xi^* \to \mathbb{W}$, defined as follows:

$$[\omega \circ t](\boldsymbol{t}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{s} \in \Sigma^*} \omega(\boldsymbol{s})\, t(\boldsymbol{s}, \boldsymbol{t}) \tag{25}$$

Definition 3. *A* ***weighted finite-state transducer (WFST)*** *is a tuple* $\mathcal{T} = \langle \Sigma, \Xi, Q, \delta, \lambda, \rho \rangle$ *where*
- Σ *is the* ***input alphabet***
- Ξ *is the* ***output alphabet***
- Q *is a finite* ***set of states***
- δ *is a* ***set of transitions*** *where each* ***transition*** *is of the form* $(q \xrightarrow{a:\, b/w} q')$ *with* $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $b \in \Xi \cup \{\varepsilon\}$, *and* $w \in \mathbb{W}$
- $\lambda\colon Q \to \mathbb{W}$ *is the* ***initial weight*** *function*
- $\rho\colon Q \to \mathbb{W}$ *is the* ***final weight*** *function.*

A ***path*** π *is a sequence of transitions of the form:*

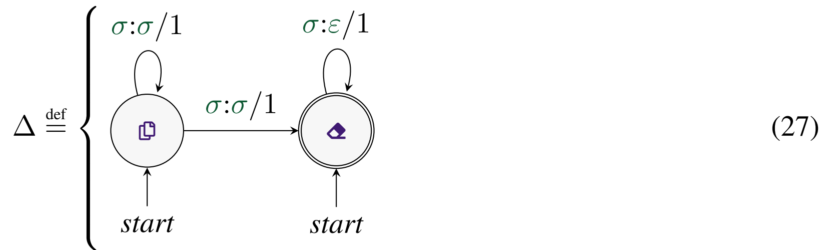
$$q_0 \xrightarrow{a_1:b_1/w_1} q_1 \xrightarrow{a_2:b_2/w_2} \cdots\, q_{N-1} \xrightarrow{a_N:b_N/w_N} q_N,$$

where $N \geq 0$ *is its length,* $a_1 \cdots a_N$ *is its* ***input yield****,* $b_1 \cdots b_N$ *is its* ***output yield****, and* $\boldsymbol{w}(\pi) \stackrel{\text{def}}{=} \lambda(q_0) w_1 \cdots w_N \rho(q_N)$ *is its* ***weight****. We denote by* $\Pi(\boldsymbol{s}, \boldsymbol{t})$ *the set of paths with input yield* $\boldsymbol{s}$ *and output yield* $\boldsymbol{t}$*. Every transducer* $\mathcal{T}$ *defines a weighted relation as follows:*

$$[\![\mathcal{T}]\!](\boldsymbol{s}, \boldsymbol{t}) \stackrel{\text{def}}{=} \sum_{\pi \in \Pi(\boldsymbol{s}, \boldsymbol{t})} \boldsymbol{w}(\pi) \tag{26}$$

The following WFST Δ implements the weighted relation $[\![\Delta]\!](\boldsymbol{s}, \boldsymbol{t}) = \mathbb{1}\{\boldsymbol{t} \preceq \boldsymbol{s}\}$.

Definition 4. *The* ***prefix transducer*** Δ *for a given alphabet* Σ *is illustrated below:*[18]



More precisely, the weighted transducer is defined by the tuple $\langle \Sigma, \Sigma, Q, \delta, \lambda, \rho \rangle$*, where*
- Σ *is both the input and output alphabet*

[17]Beyond CFGs, our observation that prefix parsing is just composition with a finite-state transducer (App. D) suggests that any formalism that efficiently supports such composition is amenable to our general approach. For example, we can, in principle, use composition to derive prefix parsers for tree-adjoining grammars (see, e.g., Nederhof, 1999). Historically, Schabes and Joshi (1988) originally presented an $\mathcal{O}(N^9)$-time algorithm to perform prefix parsing on tree-adjoining grammars. This runtime was later improved to $\mathcal{O}(N^6)$ by Nederhof (1999). Generalizing our reduction to tree-adjoining grammars would obviate the need for such bespoke algorithms.

[18]Note that any FST that encodes $[\![\Delta]\!](\boldsymbol{s}, \boldsymbol{t}) = \mathbb{1}\{\boldsymbol{t} \preceq \boldsymbol{s}\}$ would work. Of course, a smaller machine will generally lead to a smaller increase in the size of the transformed grammar. We believe this transducer is among the smallest possible, as it appears that at least two states are necessary.

- *the set of states is $Q = \{\text{⧉}, \text{⌫}\}$; we call ⧉ and ⌫ the **copy** and **erase** states, respectively*
- *the transitions δ are as shown in the picture*
- *the initial weights λ are $\lambda(\text{⧉}) = 1$ and $\lambda(\text{⌫}) = 1$*
- *the final weights are $\rho(\text{⧉}) = 0$ and $\rho(\text{⌫}) = 1$*

Note that in the picture, the start *arrow and the double circle mark states with nonzero initial and final weights, respectively. Additionally, a transition labeled σ is shorthand for a set of transitions—one for each symbol in the alphabet.*

Pasti et al. (2023) provide a general construction for the composition of a CFG $\mathcal{G}$ and a WFST $\mathcal{T}$, denoted $\mathcal{G} \circ \mathcal{T}$, which is itself a CFG, and is *correct* in the sense that it denotes the composition between the corresponding weighted relation and language, i.e., $[\![\mathcal{G} \circ \mathcal{T}]\!] = [\![\mathcal{G}]\!] \circ [\![\mathcal{T}]\!]$.

Equivalently, we can construct a grammar for the prefix language via composition with Δ: $[\![\mathcal{G} \circ \Delta]\!] = \overrightarrow{[\![\mathcal{G}]\!]}$ by Pasti et al. (2023). The composition construction is spelled out below.

Definition 5. *The **composition-based prefix grammar construction** works as follows. Given a grammar $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$, the composition $\mathcal{G} \circ \Delta = \langle \{\langle s, \alpha, s' \rangle \mid \alpha \in \mathcal{N} \cup \Sigma, s, s' \in \{\text{⧉}, \text{⌫}\}\}, \Sigma, \overrightarrow{\mathrm{S}}, \mathcal{R}' \rangle$ where $\mathcal{R}'$ is defined by the rules below.*

$$\overrightarrow{\mathrm{S}} \xrightarrow{1} \langle \text{⧉}, \mathrm{S}, \text{⌫} \rangle \tag{28a}$$

$$\overrightarrow{\mathrm{S}} \xrightarrow{1} \langle \text{⌫}, \mathrm{S}, \text{⌫} \rangle \tag{28b}$$

$$\langle \text{⧉}, \mathrm{X}, \text{⧉} \rangle \xrightarrow{w} \langle \text{⧉}, \alpha_1, \text{⧉} \rangle \cdots \langle \text{⧉}, \alpha_K, \text{⧉} \rangle \qquad \mathrm{X} \xrightarrow{w} \alpha_1 \cdots \alpha_K \in \mathcal{R} \tag{28c}$$

$$\langle \text{⧉}, a, \text{⧉} \rangle \xrightarrow{1} a \qquad a \in \Sigma \tag{28d}$$

$$\langle \text{⧉}, \mathrm{X}, \text{⌫} \rangle \xrightarrow{w} \langle \text{⧉}, \alpha_1, \text{⧉} \rangle \cdots \langle \text{⧉}, \alpha_{k-1}, \text{⧉} \rangle \, \langle \text{⧉}, \alpha_k, \text{⌫} \rangle \, \langle \text{⌫}, \alpha_{k+1}, \text{⌫} \rangle \cdots \langle \text{⌫}, \alpha_K, \text{⌫} \rangle \qquad \mathrm{X} \xrightarrow{w} \alpha_1 \cdots \alpha_K \in \mathcal{R}, \; k \in 1, \ldots, K \tag{28e}$$

$$\langle \text{⧉}, a, \text{⌫} \rangle \xrightarrow{1} a \qquad a \in \Sigma \tag{28f}$$

$$\langle \text{⌫}, \mathrm{X}, \text{⌫} \rangle \xrightarrow{w} \langle \text{⌫}, \alpha_1, \text{⌫} \rangle \cdots \langle \text{⌫}, \alpha_K, \text{⌫} \rangle \qquad \mathrm{X} \xrightarrow{w} \alpha_1 \cdots \alpha_K \in \mathcal{R} \tag{28g}$$

$$\langle \text{⌫}, a, \text{⌫} \rangle \xrightarrow{1} \varepsilon \qquad a \in \Sigma \tag{28h}$$

The construction is based directly on Pasti et al.'s (2023) composition construction for $\mathcal{G} \circ \Delta$ where Δ is given in Def. 4.[19] The proposition below establishes that the composition-based construction also correctly encodes the prefix language.

Proposition 3. *Let $\mathcal{G}$ be a CFG. Then, $\mathcal{G} \circ \Delta$ correctly encodes the prefix language of $\mathcal{G}$, i.e., $[\![\mathcal{G} \circ \Delta]\!] = \overrightarrow{[\![\mathcal{G}]\!]}$.*

Proof. Let $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$.

$$[\![\mathcal{G} \circ \Delta]\!](\boldsymbol{s}) = \sum_{\boldsymbol{z} \in \Sigma^*} [\![\mathcal{G}]\!](\boldsymbol{z}) \cdot [\![\Delta]\!](\boldsymbol{z}, \boldsymbol{s}) \qquad \triangleright \text{By Corollary 1 of Pasti et al. (2023)} \tag{29a}$$

$$= \sum_{\boldsymbol{z} \in \Sigma^*} [\![\mathcal{G}]\!](\boldsymbol{z}) \cdot \mathbb{1}\{\boldsymbol{s} \preceq \boldsymbol{z}\} \qquad \triangleright \text{By the definition of the prefix transducer (Def. 4)} \tag{29b}$$

$$= \overrightarrow{[\![\mathcal{G}]\!]}(\boldsymbol{s}) \qquad \triangleright \text{By the definition of prefix language (§2)} \tag{29c}$$

■

[19] Readers familiar with the composition algorithm may recognize that Def. 5 is a specialized application of the composition algorithm for the prefix transducer. We have hard-coded the prefix transducer's states and transitions. We have omitted nonterminals of the form $\langle \text{⌫}, _, \text{⧉} \rangle$, as they are always nongenerating and, thus, can be removed without altering the weighted language of the prefix grammar. Additionally, we have exploited knowledge of the path structure in Δ; specifically, each path with nonzero weight has the form: $\text{⧉} \xrightarrow{\sigma_1:\sigma_1/1} \text{⧉} \cdots \text{⧉} \xrightarrow{\sigma_{m-1}:\sigma_{m-1}/1} \text{⧉} \xrightarrow{\sigma_m:\sigma_m/1} \text{⌫} \xrightarrow{\sigma_{m+1}:\varepsilon/1} \text{⌫} \cdots \text{⌫} \xrightarrow{\sigma_M:\varepsilon/1} \text{⌫}$. We exploit this knowledge to reduce the number of useless rules; we can see this structure in the right-hand side of Eq. (28e).

E Instantiations of Prefix Parsing with the Prefix Grammar

In §3, we presented prefix parsing as a general transformation that allows us to turn any existing parsing algorithm into a prefix parsing one; we shall now consider two specific instantiations of this approach, one with CKY and the other with Earley as the underlying parser.

E.1 Prefix Parsing with CKY

We first instantiate the prefix-grammar approach with CKY. Our incremental formulation, CKY (Alg. 2), runs in $\mathcal{O}(|\texttt{EnsureCNF}(\mathcal{G})|N^3)$, where $\texttt{EnsureCNF}(\mathcal{G})$ is the CNF conversion of $\mathcal{G}$. PrefixCKY (Alg. 1) applies the prefix grammar transformation and then calls CKY. Note that the runtime of CKY follows the form of Theorem 2. We next analyze the blowup introduced by the prefix grammar transformation and all the preprocessing steps required in PrefixCKY.

Proposition 4. *Let $\mathcal{G}$ be a CFG. Then* $|\texttt{EnsureCNF}(\texttt{Prefix}(\texttt{EnsureCTF}(\mathcal{G})))| = \mathcal{O}(|\mathcal{G}|^2 + |\mathcal{N}||\mathcal{G}|)$.

Proof. We trace the grammar size and the nonterminal count through the preprocessing pipeline, referencing the bounds established in App. C.1. We write G_i for the grammar after step i, and N_i for its nonterminal set. Given the input grammar $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$, we have:

1. EnsureCTF: $|G'_1| \leq 3|\mathcal{G}|,\ \ |N'_1| \leq |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|$ (Eq. (17))
2. Prefix: $|G'_2| \leq \frac{8}{3}|G'_1| + 3 \leq 8|\mathcal{G}| + 3,\ \ |N'_2| = 2|N'_1| + 1$ (Prop. 2, Def. 1)
   The nonterminal count doubles because the prefix grammar introduces a prime copy X' of each nonterminal $\mathrm{X} \in N'_1$, plus one prefix start symbol $\overrightarrow{\mathrm{S}}$. Moreover, because Prefix preserves arity ≤ 2, G'_2 is already binary; invoking Eq. (22) on G'_2 would pay for an EnsureCTF step that is in fact a no-op, incurring a spurious $3\times$ factor via Eq. (17). We therefore trace the remaining EnsureCNF steps directly.
3. EnsureNullaryFree: $|G'_3| \leq \frac{7}{3}|G'_2| + 3 \leq \frac{7}{3}(8|\mathcal{G}| + 3) + 3 \leq 19|\mathcal{G}| + 10,\ \ |N'_3| \leq |N'_2| + 1$ (Eq. (18))
4. EnsureUnaryFree: $|G'_4| \leq |N'_3| \cdot |G'_3| \leq (2|N'_1| + 2)(19|\mathcal{G}| + 10),\ \ |N'_4| \leq |N'_3|$ (Eq. (19))
5. EnsureTerminalSep: $|G'_5| \leq |G'_4| + 2|\Sigma|,\ \ |N'_5| \leq |N'_4| + |\Sigma|$ (Eq. (21))

So $|\texttt{EnsureCNF}(\texttt{Prefix}(\texttt{EnsureCTF}(\mathcal{G})))| = |G'_5| \leq (2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|) + 2)(19|\mathcal{G}| + 10) + 2|\Sigma|$. Expanding and dropping lower-order terms: $|G'_5| = \mathcal{O}(|\mathcal{G}|^2 + |\mathcal{N}||\mathcal{G}|)$, matching the proposition. ■

We note that this is a worst-case bound; in practice, the blowup is much more contained. In particular, in our experiments with PCFGs, when comparing the size of the preprocessed prefix grammar $\texttt{EnsureCNF}(\overrightarrow{\mathcal{G}})$ to that of the preprocessed original grammar $\texttt{EnsureCNF}(\mathcal{G})$, the size ratio is only a small multiplicative factor (Tab. 1). Since CKY's runtime is linear in the preprocessed grammar size, this small factor translates directly to the runtime overhead for PrefixCKY.

Algorithm 1 Prefix parsing with the prefix grammar transformation and CKY (Alg. 2).

```
1. function PrefixCKY(G, s_1 ··· s_N)
2.   ▷Apply the prefix-grammar transformation (should be cached for efficiency), and parse as usual.
3.   return CKY(Prefix(EnsureCTF(G)), s_1 ··· s_N)
```

Table 1: Grammar size of different PCFGs used in our experiments. $\mathcal{G}_1$ is the grammar after CKY's preprocessing; $\mathcal{G}_2$ is the grammar after PrefixCKY's preprocessing. The ratio column reports $|\mathcal{G}_2|/|\mathcal{G}_1|$. The grammars reported in the table are WSJ 500 (obtained from the first 500 sentences of the Wall Street Journal Corpus) and "*Social Discourse*", both obtained from (Luong et al., 2013). We were not able to include the larger WSJ 5000 because our machine had insufficient memory for the unary removal operation of EnsureCNF.

Grammar	$\lvert\mathcal{G}\rvert$; $\lvert\mathcal{N}\rvert$	$\lvert\mathcal{G}_1\rvert$; $\lvert\mathcal{N}_1\rvert$	$\lvert\mathcal{G}_2\rvert$; $\lvert\mathcal{N}_2\rvert$	Ratio $\lvert\mathcal{G}_2\rvert/\lvert\mathcal{G}_1\rvert$
WSJ 500	12,573; 69	73,241; 1,766	235,459; 1,814	3.22×
Social Discourse [Sparse]	72,712; 233	211,015; 233	357,066; 306	1.69×

Algorithm 2 An incremental, left-to-right formulation of CKY that supports memoization across prefixes. CKY returns the full column of inside weights: the entry $\beta_j(i, \mathrm{X})$ gives the total weight of all derivations of X with yield $s_{i+1} \cdots s_j$. The goal weight $[\![\mathcal{G}]\!](s_1 \cdots s_N)$ is $\beta_N(0, \mathrm{S})$.

```
1.  function CKY(𝒢, s_1⋯s_N)
2.    𝒢 ← EnsureCNF(𝒢)                      ▷CNF conversion; if needed (should be cached for efficiency)
3.    β_N ← defaultdict(0)                   ▷Initialize new column
4.    if N = 0:                              ▷Base case: empty string.
5.      for (S --w--> ε) ∈ ℛ:
6.        β_N(0, S) += w
7.      return β_N(0, S), ⟨β_N⟩              ▷Below: recurse on prefix
8.    _, ⟨β_0, …, β_{N−1}⟩ ← CKY(𝒢, s_1⋯s_{N−1})
9.    for (X --w--> s_N) ∈ ℛ:                ▷Preterminal rules s_N
10.     β_N(N−1, X) += w
11.   for i in N−1 … 0:                      ▷Iterate over the start point
12.     for j in i+1 … N−1:                  ▷Iterate over the split point
13.       for (X --w--> Y Z) ∈ ℛ:            ▷Binary rules
14.         β_N(i, X) += w · β_j(i, Y) · β_N(j, Z)
15.   return β_N(0, S), ⟨β_0, …, β_{N−1}, β_N⟩
```

E.2 Prefix Parsing with Earley's Algorithm

We now instantiate the prefix-grammar approach using the weighted Earley's algorithm (Earley, 1970). Analogously to `PrefixCKY`, `PrefixEarley` (Alg. 3) applies the prefix grammar transformation and then parses with Earley's. Importantly, the experimental results (Fig. 1) show that prefix parsing is slower than ordinary parsing by a multiplicative factor of ≈ 3; this matches the prediction from the relative grammar-size blow-up (Tab. 2).

Algorithm 3 Prefix parsing with the prefix grammar transformation and `Earley` (Alg. 4).

1. **function** `PrefixEarley`$(\mathcal{G}, s_1 \cdots s_N)$
2. ▷*Apply the prefix-grammar transformation (should be cached for efficiency), and parse as usual.*
3. **return** `Earley(Prefix(EnsureCTF(`$\mathcal{G}$`)),` $s_1 \cdots s_N$`)`

An optimized version of Earley's algorithm. We present an optimized, incremental version of Earley's algorithm (Alg. 4), which we use as a base for our prefix parsing and next-token weight vector algorithms. Assuming $\mathcal{G}$ is nullary-free and unary-cycle-free, our algorithm runs in $\mathcal{O}(|\mathcal{G}|N^3)$—a factor of $|\mathcal{R}|$ faster than the traditional dotted-item Earley formulation.[20] To achieve this improved runtime, we maintain a list of *items* of the form $\langle i, \mathrm{X}/\boldsymbol{\alpha}\rangle$, where i is the starting position and $\mathrm{X}/\boldsymbol{\alpha}$ is read as "X missing $\boldsymbol{\alpha}$ to complete". The algorithm proceeds through three operations: PREDICT generates new items from grammar rules.[21] SCAN advances items over terminal symbols, and ATTACH—traditionally also known as *complete*—attaches completed items to items awaiting them. The data structure W_k indexes items at position k by the symbol they are waiting for, enabling efficient lookup during the ATTACH phase.

Grammar blowup: bounds and empirical evidence. Following Theorem 2, we bound $|\phi(\overrightarrow{\mathcal{G}})|$, where ϕ is the preprocessing pipeline required for `PrefixEarley`.

Proposition 5. *Let $\mathcal{G}$ be a CFG. Then,*
$|$`EnsureUnaryCycleFree(EnsureNullaryFree(Prefix(EnsureCTF(`$\mathcal{G}$`))))`$| = \mathcal{O}(|\mathcal{G}|^2 + |\mathcal{N}|^2)$.[22]

Proof. We analyze $\phi(\mathcal{G})$ first, then extend to $\phi(\overrightarrow{\mathcal{G}})$. We trace the grammar size and the nonterminal count through the preprocessing pipeline, referencing the bounds established in App. C.1. We write G_i for the grammar after step i, and N_i for its nonterminal set. We apply `EnsureCTF` (required before `EnsureNullaryFree` for efficiency), `EnsureNullaryFree`, and `EnsureUnaryCycleFree` in sequence. Given the input grammar $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R}\rangle$, we have

1. `EnsureCTF`: $|G_1| \le 3|\mathcal{G}|,\ |N_1| \le |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|$ (Eq. (17))
2. `EnsureNullaryFree`: $|G_2| \le 7|\mathcal{G}| + 3,\ |N_2| \le |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}| + 1$ (Eq. (18))
3. `EnsureUnaryCycleFree`: $|G_3| \le 7|\mathcal{G}| + 3 + 2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|)^2$

Thus, $|\phi(\mathcal{G})| \le 7|\mathcal{G}| + 3 + 2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|)^2$. For the prefix grammar, the preprocessing pipeline becomes:

1. `EnsureCTF`: $|G_1| \le 3|\mathcal{G}|,\ |N_1| \le |\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|$ (Eq. (17))
2. `Prefix`: $|G_2| \le \frac{8}{3}|G_1| + 3 \le 8|\mathcal{G}| + 3,\ |N_2| = 2|N_1| + 1$ (Prop. 2, Def. 1)
   Since `Prefix` preserves arity ≤ 2, G_2 is already binary and binarization is a no-op.
3. `EnsureNullaryFree`: $|G_3| \le \frac{7}{3}|G_2| + 3 \le \frac{7}{3}(8|\mathcal{G}| + 3) + 3 \le 19|\mathcal{G}| + 10,\ |N_3| \le |N_2| + 1 = 2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|) + 2$ (Eq. (18))
4. `EnsureUnaryCycleFree`: $|G_4| \le |G_3| + 2|N_3|^2 \le 19|\mathcal{G}| + 10 + 2(2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|) + 2)^2$ (Eq. (20))

Thus, $|\phi(\overrightarrow{\mathcal{G}})| \le 19|\mathcal{G}| + 10 + 2(2(|\mathcal{N}| + |\mathcal{G}| - |\mathcal{R}|) + 2)^2$. Expanding and dropping lower-order terms: $|\phi(\overrightarrow{\mathcal{G}})| = \mathcal{O}(|\mathcal{G}|^2 + |\mathcal{N}|^2)$, matching the proposition. ■

[20] Our items record only the unmatched suffix $\boldsymbol{\alpha}$, rather than the traditional $\mathrm{X} \to \boldsymbol{\gamma} \bullet \boldsymbol{\alpha}$ dotted-item form. This collapses many rules into one item, a simpler alternative to Opedal et al.'s (2023) approach, which introduces additional item types.

[21] Our implementation uses the left-corner relation to prune the PREDICT step, skipping rules whose left-hand sides are not needed by any active item (Stolcke, 1995).

[22] The $|\mathcal{G}|^2$ term arises when $\mathcal{G}$ has high-arity rules: `EnsureCTF` introduces up to $|\mathcal{G}| - |\mathcal{R}|$ new nonterminals, whose primes may participate in unary cycles. For grammars already in canonical two-form, the bound tightens to $\mathcal{O}(|\mathcal{G}| + |\mathcal{N}|^2)$. This worst case is loose in practice (see Tab. 2).

In practice, this bound is far from tight. On the grammars we experimented with, the preprocessed prefix grammar is only a small multiplicative factor larger than the ordinary grammar preprocessed for `Earley` (Tab. 2). Given our optimized `Earley` implementation's runtime bound, this small factor translates directly to the ratio of prefix parsing to parsing runtime, as confirmed empirically by our experiments (Fig. 1).

Algorithm 4 Earley's Algorithm takes the grammar $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathrm{S}, \mathcal{R} \rangle$ and a string $\boldsymbol{s} = s_1 \cdots s_k \in \Sigma^*$. It returns the *inside weight columns* $\langle \beta_0, \ldots, \beta_k \rangle$ together with the waiting-for dictionaries $\langle \mathsf{W}_0, \ldots, \mathsf{W}_k \rangle$; the total string weight is $[\![\mathcal{G}]\!](s_1 \cdots s_k) = \beta_k(0, \mathrm{S})$. The notation $\mathrm{X}/\boldsymbol{\alpha}$ is read as X missing $\boldsymbol{\alpha}$ to complete. The data structure $\mathsf{W}_k[\alpha]$ maintains a dictionary of items $\langle i, \mathrm{X}/\alpha_1 \boldsymbol{\alpha}_{[1:]} \rangle$ that are awaiting the symbol α in order to move towards completion. The queue Q is a priority queue that prioritizes items with the smallest span $k - i$, where k is the current column, and i is the item's starting position.

```
1.  def Earley(𝒢, s₁⋯s_k):                              ▷The Earley function should be memoized for efficiency.
2.    𝒢 ← EnsureUnaryCycleFree(EnsureNullaryFree(𝒢))    ▷Ensure no nullary rules, or unary rule cycles; memoize
3.    β_k ← defaultdict(0)                               ▷Initialize inside weight column
4.    W_k ← defaultdict(set)                             ▷Initialize the waiting-for dictionary
5.    Q ← priority_queue()                               ▷This queue prioritizes items with the shortest span.
6.    if k = 0:                                          ▷Base cases
7.      for (X -w-> α) ∈ ℛ:
8.        β₀(0, X/α) += w
9.      return ⟨β₀⟩, ⟨W₀⟩
10.   ⟨β₀, …, β_{k−1}⟩, ⟨W₀, …, W_{k−1}⟩ ← Earley(𝒢, s₁⋯s_{k−1})   ▷Recurse on prefix
11.   for ⟨i, X/s_k α⟩ ∈ W_{k−1}[s_k]:                   ▷SCAN
12.     β_k(i, X/α) += β_{k−1}(i, X/s_k α)
13.   while Q:                                           ▷ATTACH
14.     ⟨j, Y/ε⟩ ← Q.pop()                               ▷Pop completed item Y together with the index j where item began
15.     for ⟨i, X/Yα⟩ ∈ W_j[Y]:                          ▷Iterate through items ending at j that are waiting for a Y
16.       β_k(i, X/α) += β_j(i, X/Yα) · β_k(j, Y/ε)      ▷Attach the weights and update
17.   for (X -w-> α) ∈ ℛ:                                ▷PREDICT
18.     β_k(k, X/α) += w
19.   return ⟨β₀, …, β_{k−1}, β_k⟩, ⟨W₀, …, W_{k−1}, W_k⟩

20. ▷Helper methods
21. def [β_k(i, X/α) += v]:                              ▷This method updates the weights, Q and W_k
22.   if α = ε: Q.push(⟨i, X/ε⟩)                         ▷Newly completed items get scheduled in the priority queue
23.   else: W_k[α₁].add(⟨i, X/α⟩)                        ▷Item ⟨i, X/α⟩ is waiting for α₁ to move forward
24.   β_k(i, X/α) ← β_k(i, X/α) + v
```

Table 2: Grammar size of different PCFGs $\mathcal{G}$ used in our experiments. $\mathcal{G}_1$ and $\mathcal{G}_2$ are respectively the grammars obtained after the preprocessing step of `Earley` and `PrefixEarley`. The ratio column reports $|\mathcal{G}_2|/|\mathcal{G}_1|$. The grammars reported in the table are WSJ 500 (obtained from the first 500 sentences of the Wall Street Journal Corpus), WSJ 5000 (obtained from the first 5000 sentences of the Wall Street Journal Corpus), and "*Social Discourse*". We empirically found that applying `EnsureCNF` after the prefix grammar transformation and right before `EnsureNullaryFree` yielded a smaller grammar; the results shown in this table follow this approach.

Grammar	$\lvert\mathcal{G}\rvert$; $\lvert\mathcal{N}\rvert$	$\lvert\mathcal{G}_1\rvert$; $\lvert\mathcal{N}_1\rvert$	$\lvert\mathcal{G}_2\rvert$; $\lvert\mathcal{N}_2\rvert$	**Ratio** $\lvert\mathcal{G}_2\rvert/\lvert\mathcal{G}_1\rvert$
WSJ 500	12,573; 69	15,981; 1,775	43,701; 5,201	2.73×
WSJ 5000	116,667; 448	177,303; 30,878	494,017; 84,566	2.79×
Social Discourse [Sparse]	72,712; 233	72,712; 233	143,548; 435	1.97×

F Practical Instantiations of the Next-Token Weight Vector Algorithm

In §4, we presented a general framework for computing the next-token weight vector $\boldsymbol{\pi}(\boldsymbol{s})$ via lattice parsing and algorithmic differentiation. This section instantiates this framework with two concrete parsing algorithms: CKY (App. F.1) and Earley's (App. F.2). In both cases, we derive the gradient algorithm by manually applying the rules of reverse-mode algorithmic differentiation to the corresponding lattice parser.[23]

F.1 Next-Token Weight Vector with CKY

Algorithm 5 **Left**: the `LatticeCKY` is a specialized lattice parser that parses the next-token lattice $\mathcal{L}_{\boldsymbol{s}}(\boldsymbol{\theta})$ for some choice of $\boldsymbol{\theta} \in \mathbb{W}^{\Sigma}$. Note that `LatticeCKY` is directly derived from `CKY`, differing only in the terminal step (line 6), which scans all terminal symbols simultaneously, weighting each by θ_a. **Right**: the `NextTokenCKY` algorithm, which was derived by manually applying algorithmic differentiation to `LatticeCKY`. Given the input grammar $\mathcal{G}$, the input string $s_1 \cdots s_N$, and the inside weight columns $\langle \beta_0, \ldots, \beta_N \rangle$, it returns the next-token weight vector $\boldsymbol{\pi}(\boldsymbol{s})$. Note that both algorithms call `CKY` as a subroutine to compute the inside weight columns (which, however, can be amortized across subsequent calls when `NextTokenCKY` is called incrementally).

```
1.  function LatticeCKY(𝒢, s_1···s_N, θ)
2.    𝒢 ← EnsureCNF(𝒢)                          ▷CNF conversion; memoize
3.    ⟨β_0, ..., β_N⟩ ← CKY(𝒢, s_1···s_N)
4.    z ← defaultdict(0)                        ▷Initialize new column
5.    ▷Base case:
6.    for (X -w-> a) ∈ ℛ with a ∈ Σ:
7.      z(N, X) += w · θ_a
8.    ▷Recursive step:
9.    for i in N, ..., 0:                       ▷Iterate over the start point.
10.     for j in i+1 ... N:                     ▷Split point
11.       for (X -w-> Y Z) ∈ ℛ:                 ▷Binary rules
12.         z(i, X) += w · β_j(i, Y) · z(j, Z)
13.   return z(0, S)                            ▷equals Z_s(θ)
```

```
1.  function NextTokenCKY(𝒢, s_1···s_N)
2.    ▷Create prefix grammar in CNF; memoize
3.    𝒢⃗ ← EnsureCNF(Prefix(EnsureCTF(𝒢)))
4.    ⟨β_0, ..., β_N⟩ ← CKY(𝒢⃗, s_1···s_N)
5.    ∇z ← defaultdict(0)
6.    ∇z(0, S) += 1                             ▷Base case
7.    for i in 0 ... N−1:                       ▷Inverse iteration
8.      for j in i+1 ... N:                     ▷Loop over the split point j
9.        for (X -w-> Y Z) ∈ ℛ:
10.         ∇z(j, Z) += w · β_j(i, Y) · ∇z(i, X)
11.   π ← 0^Σ
12.   for (X -w-> a) ∈ ℛ where a ∈ Σ:           ▷Preterminal rules
13.     π_a += w · ∇z(N, X)
14.   return π
```

In `LatticeCKY` (Alg. 5, left), the *forward values* $z(i, \mathrm{X})$ for $i = 0, \ldots, N$ and $\mathrm{X} \in \mathcal{N}$ aggregate partial contributions to $Z_{\boldsymbol{s}}(\boldsymbol{\theta}) = z(0, \mathrm{S})$. In `NextTokenCKY` (right), the corresponding *backward values* $\nabla z(i, \mathrm{X}) \overset{\text{def}}{=} \frac{\partial z(0,\mathrm{S})}{\partial z(i,\mathrm{X})}$ disaggregate this sum and isolate each token's contribution. Readers familiar with the inside–outside algorithm (Baker, 1979) will notice that the backward pass is closely related to the *outside algorithm*, following Eisner's (2016) presentation of the outside algorithm as the *adjoint* of the inside algorithm. The runtime of `NextTokenCKY`, following the analysis of `PrefixCKY`, is $\mathcal{O}\big(|\texttt{EnsureCNF(Prefix(EnsureCTF}(\mathcal{G})))|N^3\big)$, including the cost of computing the *inside weight columns* $\beta_0, \ldots, \beta_N$, which can be amortized across incremental computations.

F.2 Next-Token Weight Vector with Earley's Algorithm

We now adapt the lattice parsing and algorithmic differentiation approach to Earley's algorithm (Earley, 1970) and its weighted version (Stolcke, 1995). Earley's algorithm requires less preprocessing than CKY—only `EnsureNullaryFree` and `EnsureUnaryCycleFree`—and runs in subcubic time for most grammars: linear for deterministic grammars, and quadratic for unambiguous ones.

Next-token weight vector with Earley (Alg. 6). In `IncrEarleyLattice`, the *forward values* $z(j, \mathrm{Y}/\varepsilon)$ for $j \in \{0, \ldots, N\}$ and $\mathrm{Y} \in \mathcal{N}$ aggregate partial contributions to the goal item $z(0, \mathrm{S}) = Z_{s_1 \cdots s_N}(\boldsymbol{\theta})$. In `NextTokenEarley`, the corresponding *backward values* $\nabla z(j, \mathrm{Y}/\varepsilon) \overset{\text{def}}{=} \frac{\partial z(0,\mathrm{S})}{\partial z(j,\mathrm{Y}/\varepsilon)}$ disaggregate this sum to isolate each token's contribution. The runtime of `NextTokenEarley`, following the analysis of `PrefixEarley`, is $\mathcal{O}\big(|\texttt{EnsureUnaryCycleFree(EnsureNullaryFree(Prefix(EnsureCTF}(\mathcal{G}))))|N^3\big)$, including the cost of computing the *inside weight columns* $\beta_0, \ldots, \beta_N$, which can be amortized across incremental computations.

[23] In general, the gradient program can also be obtained automatically, e.g., via program tracing, operator overloading, or source-to-source transformation (see Griewank and Walther, 2008, for an overview), using tools such as PyTorch (Paszke et al., 2017), TensorFlow (Abadi et al., 2016), JAX (Bradbury et al., 2018), Tapenade (Hascoët and Pascual, 2013), or Dyna (Eisner et al., 2005).

Algorithm 6 **Left**: `IncrEarleyLattice` is a lattice parser derived from Earley's algorithm (Alg. 4) that evaluates the next-token lattice $\mathcal{L}_s(\boldsymbol{\theta})$ for some choice of $\boldsymbol{\theta} \in \mathbb{W}^\Sigma$. It differs from `Earley` only in the SCAN step, which scans all terminal symbols simultaneously, weighting each by θ_a. Since only a single terminal is scanned at position $N+1$, the following simplifications apply: (1) SCAN processes only items from W_N that complete after scanning one symbol (line 9), and ATTACH only advances items with a single remaining symbol (line 14); (2) since no further columns are needed, PREDICT can be omitted entirely. **Right**: `NextTokenEarley` is the gradient algorithm of `IncrEarleyLattice`, obtained by applying algorithmic differentiation. Given the prefix grammar $\overrightarrow{\mathcal{G}}$, the input string $\boldsymbol{s} = s_1 \cdots s_N$, and the inside weight columns from `Earley`, it returns the next-token weight vector $\boldsymbol{\pi}(\boldsymbol{s})$. The backward values $\nabla z(j, \mathrm{Y}/\varepsilon)$ are computed by the memoized helper function (line 9).

```
1.  function IncrEarleyLattice(𝒢, s_1⋯s_N, θ)
2.    𝒢 ← EnsureNullaryFree(𝒢)
3.    𝒢 ← EnsureUnaryCycleFree(𝒢)
4.    ⟨𝒩, Σ, S, ℛ⟩ ← 𝒢
5.    ⟨β_0, …, β_N⟩,⟨W_0, …, W_N⟩ ← Earley(𝒢, s_1⋯s_N)
6.    z ← defaultdict(0)                          ▷Initialize new column
7.    Q ← priority_queue()
8.    for a ∈ W_N.keys ∩ Σ:                       ▷SCAN
9.      for ⟨i, X/a⟩ ∈ W_N[a]:
10.       z(i, X/ε) += β_N(i, X/a) · θ_a
11.       Q.push(⟨i, X/ε⟩)
12.   while Q:                                    ▷ATTACH
13.     ⟨j, Y/ε⟩ ← Q.pop()
14.     for ⟨i, X/Y⟩ ∈ W_j[Y]:
15.       z(i, X/ε) += β_j(i, X/Y) · z(j, Y/ε)
16.       Q.push(⟨i, X/ε⟩)
17.   return z(0, S/ε)                            ▷equals Z_{s_1⋯s_N}(θ)
```

```
1.  def NextTokenEarley(𝒢, s_1⋯s_N):
2.    G⃗ ← Prefix(EnsureCTF(𝒢))                    ▷Prefix transformation
3.    ⟨β_0, …, β_N⟩,⟨W_0, …, W_N⟩ ← Earley(G⃗, s_1⋯s_N)
4.    π ← 0^Σ
5.    for a ∈ W_N.keys ∩ Σ:
6.      for ⟨i, X/a⟩ ∈ W_N[a]:
7.        π_a += β_N(i, X/a) · ∇z(i, X/ε)
8.    return π
9.  def ∇z(j, Y/ε):                               ▷Memoize for efficiency
10.   if (j, Y) = (0, S): return 1                ▷Base Case
11.   v ← 0
12.   for ⟨i, X/Y⟩ ∈ W_j[Y]:                      ▷Recursive bottom-up call.
13.     v += β_j(i, X/Y) · ∇z(i, X/ε)             ▷Note: i ≤ j.
14.   return v
```

G Deferred Proofs

G.1 Proof of Prop. 1

Proposition 1. *Let $\mathcal{G}$ be a CFG, and $\overrightarrow{\mathcal{G}}$ be its prefix grammar. Then, $\overrightarrow{\mathcal{G}}$ correctly encodes the prefix language of $\mathcal{G}$ (i.e., $[\![\overrightarrow{\mathcal{G}}]\!] = [\![\overrightarrow{\mathcal{G}}]\!]$).*

Proof. We strengthen the proposition to a claim over all $\alpha \in \mathcal{N} \cup \Sigma$, prove the strengthening by induction on derivation-tree height in $\overrightarrow{\mathcal{G}}$, and recover the proposition by specializing to $\alpha = \mathrm{S}$.

Strengthening. The Prop. 1 is implied by the following statement, for all $\alpha \in \mathcal{N} \cup \Sigma$ and all $\boldsymbol{s} \in \Sigma^+$:

$$[\![\overrightarrow{\mathcal{G}}_{\alpha'}]\!](\boldsymbol{s}) = \sum_{\boldsymbol{t} \in \Sigma^*} [\![\mathcal{G}_\alpha]\!](\boldsymbol{st}) \quad \text{and} \quad [\![\overrightarrow{\mathcal{G}}_\alpha]\!](\boldsymbol{s}) = [\![\mathcal{G}_\alpha]\!](\boldsymbol{s}) \tag{30}$$

The latter holds because $\overrightarrow{\mathcal{G}}$'s additional rules have only prime nonterminals or $\overrightarrow{\mathrm{S}}$ on the LHS (Def. 1), so non-prime α-rooted derivations use only the rules of $\mathcal{R}$. Thus, we need only prove the former, which we re-express self-recursively in $\overrightarrow{\mathcal{G}}$:

$$[\![\overrightarrow{\mathcal{G}}_{\alpha'}]\!](\boldsymbol{s}) = \sum_{\boldsymbol{t} \in \Sigma^*} [\![\overrightarrow{\mathcal{G}}_\alpha]\!](\boldsymbol{st}) \tag{31}$$

$\hookrightarrow$ **Hooking up Eq. (30) to Prop. 1.** Specializing the first equation of Eq. (30) to $\alpha = \mathrm{S}$ and expanding $\overrightarrow{\mathrm{S}}$'s rules (Def. 1) yields the proposition. Recall that $\overrightarrow{\mathcal{G}}$'s start symbol $\overrightarrow{\mathrm{S}}$ has two rules: $\overrightarrow{\mathrm{S}} \xrightarrow{1} \mathrm{S}'$ and $\overrightarrow{\mathrm{S}} \xrightarrow{\tau(\mathrm{S})} \varepsilon$.

- **Case** ($\boldsymbol{s} \in \Sigma^+$): Only the first rule contributes (the second requires $\boldsymbol{s} = \varepsilon$). By Eq. (30) at $\alpha = \mathrm{S}$:

$$[\![\overrightarrow{\mathcal{G}}]\!](\boldsymbol{s}) = [\![\overrightarrow{\mathcal{G}}_{\mathrm{S}'}]\!](\boldsymbol{s}) = \sum_{\boldsymbol{t} \in \Sigma^*} [\![\mathcal{G}_{\mathrm{S}}]\!](\boldsymbol{st}) = [\![\overrightarrow{\mathcal{G}}]\!](\boldsymbol{s})$$

- **Case** ($\boldsymbol{s} = \varepsilon$): Every S'-derivation has nonempty yield (the prime chain from the root must terminate at a terminal via Eq. (4c)), so only the $\overrightarrow{\mathrm{S}} \to \varepsilon$ rule contributes. By the definition of $\tau(\mathrm{S})$:

$$[\![\overrightarrow{\mathcal{G}}]\!](\varepsilon) = \tau(\mathrm{S}) = \sum_{\boldsymbol{t} \in \Sigma^*} [\![\mathcal{G}_{\mathrm{S}}]\!](\boldsymbol{t}) = [\![\overrightarrow{\mathcal{G}}]\!](\varepsilon)$$

Notation. We introduce the following notation to keep the proof tidy.

- Let $\overrightarrow{\mathcal{D}}$ denote the set of derivation trees of $\overrightarrow{\mathcal{G}}$.
- Let $\overrightarrow{\mathcal{D}}^{(h)}$ denote the subset of trees in $\overrightarrow{\mathcal{D}}$ of height $\leq h$; $\overrightarrow{\mathcal{D}}^{(h)}_\alpha(\boldsymbol{s})$ restricts to α-rooted trees with yield $\boldsymbol{s}$.
- Let $\mathcal{G}^{(h)}$ denote $\mathcal{G}$ with its derivations restricted to height $\leq h$.
- Let $\overrightarrow{\mathcal{G}}^{(h)} \overset{\text{def}}{=} \overrightarrow{(\mathcal{G}^{(h)})}$ be the prefix grammar of $\mathcal{G}^{(h)}$; its prime rule weights use the height-bounded totals $\overrightarrow{\tau}^{(h)}_\alpha$ rather than $\tau(\alpha)$.
- Let $\overrightarrow{\mathrm{W}}^{(h)}_\alpha(\boldsymbol{s}) \overset{\text{def}}{=} \sum_{\boldsymbol{d} \in \overrightarrow{\mathcal{D}}^{(h)}_\alpha(\boldsymbol{s})} \boldsymbol{w}(\boldsymbol{d})$
- Let $\overrightarrow{\tau}^{(h)}_\alpha \overset{\text{def}}{=} \sum_{\boldsymbol{t} \in \Sigma^*} \overrightarrow{\mathrm{W}}^{(h)}_\alpha(\boldsymbol{t})$

Proof strategy. We prove Eq. (30) by induction on the height $h \geq 1$ of derivation trees in $\overrightarrow{\mathcal{G}}$. We define the height-indexed proposition:

$$\Phi(h) \iff \forall \alpha \in \mathcal{N} \cup \Sigma, \boldsymbol{s} \in \Sigma^+ \colon \overrightarrow{\mathrm{W}}^{(h)}_{\alpha'}(\boldsymbol{s}) = \sum_{\boldsymbol{t} \in \Sigma^*} \overrightarrow{\mathrm{W}}^{(h)}_\alpha(\boldsymbol{st}) \tag{IH}$$

Base case ($h = 1$). We show $\Phi(1)$ holds. There are two ways for a tree to have height one.

- **Case** ($\alpha \in \Sigma$): Since $\alpha' = \alpha$, both sides of $\Phi(1)$ reduce to $\mathbb{1}\{\boldsymbol{s} = \alpha\}$, using $\boldsymbol{s} \in \Sigma^+$ to collapse the right-hand side sum.
- **Case** ($\mathrm{X} \xrightarrow{w} \varepsilon$): X' has no height-1 derivation since the rules of Eq. (4c) have nonempty right-hand sides, so the LHS of $\Phi(1)$ is 0. The RHS is also 0 because $\overrightarrow{\mathrm{W}}^{(1)}_{\mathrm{X}}(\boldsymbol{st})$ is nonzero only when $\boldsymbol{st} = \varepsilon$, which contradicts $\boldsymbol{s} \in \Sigma^+$.

Inductive hypothesis. Suppose $\Phi(h)$ holds for some $h \geq 1$.

Inductive case. We show $\Phi(h+1)$ holds. In this case, we know that α' is of the form X' for $\mathrm{X} \in \mathcal{N}$.

$$\begin{aligned}
&\sum_{t\in\Sigma^*} \overrightarrow{\mathrm{W}}^{(h+1)}_{\mathrm{X}}(st) \\
&= \sum_{t\in\Sigma^*} \sum_{\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha_K} \sum_{st=v_1\cdots v_K} w\cdot \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k)\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K) && \text{(32a)}\\
&= \sum_{\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha_K} \sum_{k=1}^{K} \sum_{\substack{s=v_1\cdots v_k\\ t=v_{k+1}\cdots v_K}} w\cdot \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k)\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K) && \text{(32b)}\\
&= \sum_{\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha_K} \sum_{k=1}^{K} \sum_{s=v_1\cdots v_k} w\cdot \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k) \sum_{t=v_{k+1}\cdots v_K} \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K) && \text{(32c)}\\
&= \sum_{\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha_K} \sum_{k=1}^{K} \sum_{s=v_1\cdots v_k} w\cdot \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k)\left(\overrightarrow{\tau}^{(h)}_{\alpha_{k+1}}\cdots\overrightarrow{\tau}^{(h)}_{\alpha_K}\right) && \text{(32d)}\\
&= \sum_{\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha'_k} \sum_{s=v_1\cdots v_k} w\cdot \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha'_k}(v_k) && \text{(32e)}\\
&= \overrightarrow{\mathrm{W}}^{(h+1)}_{\mathrm{X}'}(s) && \text{(32f)}
\end{aligned}$$

The manipulation above proceeds in six steps:

1. **Tree-weight recursion for** X**.** Each height-$\leq h+1$ tree rooted at X applies some rule $\mathrm{X}\xrightarrow{w}\alpha_1\cdots\alpha_K$ at the root with subtrees of height $\leq h$.
2. **Border position.** Reparametrize the joint sum $\sum_t \sum_{st=v_1\cdots v_K}$ by introducing $k \in \{1,\ldots,K\}$ with $s = v_1\cdots v_k$ and $t = v_{k+1}\cdots v_K$.
3. **Factor the t-side.** The factors $\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_1}(v_1)\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k)$ are fixed by the s-split, so they move out of the inner sum over $t = v_{k+1}\cdots v_K$.
4. **Collapse to totals.** We simplify the inner summation:

$$\begin{aligned}
&\sum_{t=v_{k+1}\cdots v_K} \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K)\\
&= \sum_{v_{k+1}}\cdots\sum_{v_K} \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\cdots\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K)\\
&= \left[\sum_{v_{k+1}} \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_{k+1}}(v_{k+1})\right]\cdots\left[\sum_{v_K} \overrightarrow{\mathrm{W}}^{(h)}_{\alpha_K}(v_K)\right]\\
&= \overrightarrow{\tau}^{(h)}_{\alpha_{k+1}}\cdots\overrightarrow{\tau}^{(h)}_{\alpha_K}
\end{aligned}$$

5. **The prime rule.** The absorbed weight $w' = w\cdot\overrightarrow{\tau}^{(h)}_{\alpha_{k+1}}\cdots\overrightarrow{\tau}^{(h)}_{\alpha_K}$ matches the prime rule $\mathrm{X}'\to\alpha_1\cdots\alpha_{k-1}\alpha'_k$ (Eq. (4c)), so the joint sum over (original rule, k) becomes a pattern-match over such prime rules. Applying $\Phi(h)$ to α_k rewrites $\overrightarrow{\mathrm{W}}^{(h)}_{\alpha_k}(v_k)$ as $\overrightarrow{\mathrm{W}}^{(h)}_{\alpha'_k}(v_k)$.
6. **Tree-weight recursion for** X'**.** The result is the tree-weight recursion for $\overrightarrow{\mathrm{W}}^{(h+1)}_{\mathrm{X}'}(s)$.

This establishes $\Phi(h+1)$, completing the induction. Since every derivation tree in $\overrightarrow{\mathcal{G}}$ has finite height, $\Phi(h)$ for all $h \geq 1$ recovers Eq. (30), and the proposition follows via the hook-up step.[24]

■

[24] Our proof implicitly uses $\overrightarrow{\tau}^{(h)}$ in the prime rule weights, i.e., the totals of $\overrightarrow{\mathcal{G}}^{(h)}$ rather than $\overrightarrow{\mathcal{G}}$. As $h\to\infty$, $\overrightarrow{\tau}^{(h)}_{\alpha}\to\tau(\alpha)$, recovering $\overrightarrow{\mathcal{G}}$'s rule weights; $\Phi(h)$ for all $h\geq 1$ then gives Eq. (30). A more formal induction use pairs $(\mathbb{1}\{\alpha\in\mathcal{N}\}, h)$.

G.2 Proof of Prop. 2

Proposition 2. *Let $\mathcal{G}$ be a context-free grammar in canonical two-form and let $\overrightarrow{\mathcal{G}}$ be its prefix grammar (Def. 1). Then, the size of $\overrightarrow{\mathcal{G}}$ is bounded by*

$$|\overrightarrow{\mathcal{G}}| \leq \frac{8}{3}|\mathcal{G}| + 3 \tag{5}$$

Proof. According to Def. 1, the prefix grammar $\overrightarrow{\mathcal{G}}$ consists of the original rules $\mathcal{R}$ plus the additional prefix rules $\mathcal{R}'$. We bound each contribution:

- The original rules $\mathcal{R}$ contribute $|\mathcal{G}|$.
- Eqs. (4a) and (4b) contribute $2 + 1 = 3$.
- For Eq. (4c):
  - Each nullary rule (size 1) produces no prefix rules.
  - Each unary rule (size 2) produces one prefix rule ($k = 1$) of size 2, contributing 2 to the size.
  - Each binary rule (size 3) produces two prefix rules: one for $k = 1$ of size 2 and one for $k = 2$ of size 3, contributing $2 + 3 = 5$ to the size.

  In these cases, the ratio of the prefix contribution to the original rule size is at most $5/3$ (attained by binary rules). So the total contribution of Eq. (4c) is at most $\frac{5}{3}|\mathcal{G}|$.

Summing: $|\overrightarrow{\mathcal{G}}| \leq |\mathcal{G}| + 3 + \frac{5}{3}|\mathcal{G}| = \frac{8}{3}|\mathcal{G}| + 3$. ■