CONTEXT-FREE RECOGNITION WITH TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Transformers excel on tasks that process well-formed inputs according to some grammar, such as natural language and code. However, it remains unclear how they can process grammatical syntax internally. In fact, under standard complexity conjectures, standard transformers cannot recognize context-free languages (CFLs), a canonical formalism to describe syntax, or even regular languages, a subclass of CFLs. Merrill & Sabharwal (2025a) show that $\mathcal{O}(\log(n))$ *looping layers* (w.r.t. input length n) allows transformers to recognize regular languages, but the question of context-free recognition remained open. In this work, we show that *looped transformers* with $\mathcal{O}(\log(n))$ looping layers and $\mathcal{O}(n^6)$ padding tokens can recognize all CFLs. However, training and inference with $\mathcal{O}(n^6)$ padding tokens is potentially impractical. Fortunately, we show that, for natural subclasses such as unambiguous CFLs, the recognition problem on transformers becomes more tractable, requiring $\mathcal{O}(n^3)$ padding. We empirically validate our results and show that looping helps on languages that require logarithmic depth. Overall, our results shed light on the intricacy of CFL recognition by transformers: while general recognition may require an intractable amount of padding, natural constraints such as unambiguity yield efficient recognition algorithms.

1 INTRODUCTION

Transformers are proficient at many natural language (Qin et al., 2024) and coding (Jiang et al., 2024) tasks, both of which involve processing hierarchical structures. Analysis of internal representations has shown that transformers learn to encode syntactic features related to parsing, the task of extracting the syntactic structure of a sentence, during pre-training (Hewitt & Manning, 2019; Arps et al., 2022; Zhao et al., 2023). Classically, the ability to process hierarchically nested structures is closely connected to the ability to model context-free languages (CFLs). However, it is unclear what classes of syntax transformers can *provably* represent, and how CFL recognition can be implemented internally. To this end, we study whether transformers can correctly determine the grammaticality of a sentence according to a context-free grammar.

The problem of determining whether an input is grammatical can be stated as the *recognition problem for context-free grammars* (CFGs): Given a CFG $\mathcal{G}$, can a string $\boldsymbol{w}$ be generated by $\mathcal{G}$? Several foundational *serial* parsing algorithms (Earley, 1970; Cocke, 1969; Kasami, 1965; Younger, 1967) solve this problem. However, such serial procedures cannot be naturally implemented by transformers, due to their highly parallel, fixed-depth structure. Even regular languages, a strict subset of CFLs, cannot be recognized by fixed-depth transformers under the standard complexity conjecture $\mathrm{TC}^0 \subsetneq \mathrm{NC}^1$: Regular language recognition is complete for NC^1 (Barrington & Thérien, 1988) while fixed-depth transformers fall in TC^0 (Merrill et al., 2022; Strobl, 2023; Chiang, 2025). *Looping* layers helps: $\log(n)$ looping layers (where n is the input length) allow transformers to recognize regular languages Merrill & Sabharwal (2025a). However, the question of whether logarithmic looping enables CFL recognition remains.

In this work, we address this question by analyzing the difficulty of the recognition problem for various classes of CFLs on transformers. We conceptualize the difficulty in terms of extra resources needed: looping layers and dynamically appending blank *padding* tokens (Merrill & Sabharwal, 2025b). Precisely, while general CFL recognition cannot be implemented by fixed-depth transformers under standard complexity conjectures, we show in §3 via a direct construction that it can be expressed by looping layers $\mathcal{O}(\log(n))$ times and with $\mathcal{O}(n^6)$ padding tokens.

To the best of our knowledge, this constitutes the first proof of CFL recognition by transformers. Moreover, we ask whether there are simpler classes of CFLs that can be recognized by transformers with reduced padding requirements. Indeed, we show in §4 that natural subclasses of CFLs can be recognized by simpler transformers. We identify unambiguity and linearity as two key properties that make CFL recognition more tractable. Unambiguous CFLs, characterized by strings having at most one possible parse, allow for recognition with reduced padding but more looping. This aligns with transformers' struggles to parse ambiguous grammars in practice (Khalighinejad et al., 2023). Furthermore, imposing linearity (where each grammar rule has at most one non-terminal on its right-hand side) reduces the amount of looping and padding required for recognizing unambiguous CFL. We empirically test in §5 when looping helps generalization and find it to increase the performance on a $\log$-depth complete CFL, namely the language of variable-free Boolean formulas (Buss, 1987).

In summary, we leverage theory on parallel recognition of CFLs to show that transformers can recognize CFLs with logarithmic depth, characterizing the padding requirements for different relevant subclasses. These results imply that, in order to recognize CFLs, transformers require significantly less depth than that which would be needed to implement a serial parsing algorithm like CKY. While this comes with increased space (padding) requirements in the general case, the space can be reduced for natural CFL subclasses. These results are summarized in Tab. 1.

Language class	Padding tokens required	Looping layers required
General CFLs	n^6	$\log(n)$
Unambiguous CFLs	n^3	$\log^2(n)$
Unambiguous linear CFLs	n^2	$\log(n)$

Table 1: Main results: The amount of computational resources required by transformers to recognize different classes of context-free languages (CFLs).

2 PRELIMINARIES

An **alphabet** Σ is a finite, non-empty set of **symbols**. A **string** is a finite sequence of symbols from Σ. The **Kleene closure** Σ^* of Σ is the set of all strings over Σ, and ε denotes the empty string. A **formal language** $\mathbb{L}$ over Σ is a subset of Σ^*, and a **language class** is a set of formal languages. We treat a **language recognizer** as a function $\mathtt{R}\colon \Sigma^* \to \{0,1\}$ whose language is $\mathbb{L}(\mathtt{R}) \stackrel{\text{def}}{=} \{\boldsymbol{w} \in \Sigma^* \mid \mathtt{R}(\boldsymbol{w}) = 1\}$.

2.1 CONTEXT-FREE GRAMMARS

Definition 2.1. *A **context-free grammar** (CFG) $\mathcal{G}$ is a tuple $(\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ where: (1) Σ is an alphabet of **terminal** symbols (2) $\mathcal{N}$ is a finite non-empty set of **nonterminal** symbols with $\mathcal{N} \cap \Sigma = \emptyset$ (3) $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$ is a set of **production** rules of the form $\mathrm{A} \to \alpha$ for $\mathrm{A} \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \alpha)^*$ (4) $\mathrm{S} \in \mathcal{N}$ is a designated start non-terminal symbol As standard, we denote terminal and nonterminal symbols by lowercase and uppercase symbols, respectively.*

We call a sequence of non-terminals and terminals $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$ a **sentential form**. A context-free grammar generates strings by repeatedly applying rules to sentential forms derived from the start symbol until it produces a sequence of terminal symbols, i.e a string. We call this procedure a **derivation**, and the resulting string its **yield**. We define the relation $\mathrm{A} \to \beta$ if $\exists p \in \mathcal{P}$ such that $p = (\mathrm{A} \to \alpha\beta\gamma)$ where α, β, γ are sentential forms. We denote by $\stackrel{*}{\Rightarrow}$ the reflexive, transitive closure of $\to$.

Definition 2.2. *The **language of a grammar** $\mathcal{G}$ is the set $\mathbb{L}(\mathcal{G}) \stackrel{\text{def}}{=} \{\boldsymbol{w} \in \Sigma^* \mid \mathrm{S} \stackrel{*}{\Rightarrow} \boldsymbol{w}\}$.*

Definition 2.3. *A language $\mathbb{L}$ is **context-free** if there exists a CFG $\mathcal{G}$ such that $\mathbb{L}(\mathcal{G}) = \mathbb{L}$.*

It is common practice to consider CFGs in a normal form, namely:

Definition 2.4. *A CFG $\mathcal{G}$ is in **Chomsky Normal Form (CNF)** if any $p \in \mathcal{P}$ is either of the form $\mathrm{A} \to \mathrm{BC}$, $\mathrm{A} \to a$ or $\mathrm{S} \to \varepsilon$.*

Every CFG can be transformed into an equivalent one in CNF.

2.2 TRANSFORMERS

We consider transformers as defined in Merrill & Sabharwal (2025a;b). We assume **average-hard attention** (AHATs), where attention returns a uniform average of the values of tokens that maximize the attention score. We further assume masked pre-norm, where the layer normalization is applied before the residual connection. We assume log-precision, where symbol representations contain values that can be represented with $\mathcal{O}(\log(n))$ bits for an input of size n. Coupling AHATs and log-precision unlocks useful features such as storing string indices and performing counts across a string (Merrill & Sabharwal, 2023). We assume input strings to the transformer are augmented with both a beginning-of-sequence and end-of-sequence token. Denote by $\boldsymbol{x}^L_{\text{EOS}}$ the contextual representation of EOS at end of the forward pass of the transformer. We apply a linear classifier to $\boldsymbol{x}^L_{\text{EOS}}$ to determine string acceptance.

Looped transformers scale the number of layers with input length (Merrill & Sabharwal, 2025a).

Definition 2.5. *Let T be a transformer. We denote by $\langle A, B, C\rangle$ a partition of layers such that A is the **initial block** of layers, B is the **looped** block of layers and C is the **final block** of layers. T is $d(n)$**-looped** if upon a forward pass with an input of length n, B is repeated $\mathcal{O}(d(n))$ times.*

The amount of computation performed by self attention is definitionally quadratic in the string length. One can dynamically increase this by adding *padding space* (Merrill & Sabharwal, 2025b).

Definition 2.6. *Let T be a transformer. T is $w(n)$**-padded** if $\mathcal{O}(w(n))$ padding tokens are appended to the end of the string when computing the contextual representations of a length-n input.*

Dynamically scaling number of layers and padding tokens in transformers is analogous to scaling time and space in classical models of computation such as Boolean circuits (Merrill & Sabharwal, 2025b). Allowing for different looping and padding budgets results in different classes of transformers. We adopt naming conventions of these models from Merrill & Sabharwal (2025b). We denote by AHAT^d_k the class of languages recognized by averaging hard-attention transformers with $\mathcal{O}(\log^d(n))$-looping, $\mathcal{O}(n^k)$-padding and strict causal masking. We further use the notation uAHAT to refer to refer to average hard-attention transformers with strict causal masking with no masking, and mAHAT^d_k when induction heads may or may not employ strict causal masking.

We have the following convenient result on simulating non-masked attention layers with causally masked attention layers by adding a linear amount of padding.

Lemma 2.1 (Merrill & Sabharwal 2025b)**.** $\mathsf{uAHAT}^d_k \subseteq \mathsf{mAHAT}^d_k \subseteq \mathsf{AHAT}^d_{1+\max(k,1)}$ *for* $d \geq 1$

We refer to App. A for more details on the looped and padded transformer model.

3 RECOGNIZING GENERAL CFLS WITH TRANSFORMERS

We now introduce our parallel algorithm for general CFL recognition, which synthesizes ideas from previous work on algorithms for parallel CFL recognition (Ruzzo, 1980; Rossmanith & Rytter, 1992; Lange & Rossmanith, 1990). We then show how to implement this algorithm on AHATs.

Our goal is to recognize a CFL represented by a grammar in CNF (Def. 2.4). For a string $\boldsymbol{w}$ of length n, we will determine whether the grammar can generate the string. To do this, we will work with tuples of the form $[\mathrm{A}, i, j]$ which we call **items**, where $\mathrm{A} \in \mathcal{N}$ and $i, j \in [n]$. Given a string $\boldsymbol{w}$, the item $[\mathrm{A}, i, j]$ is **realizable** if and only if $\mathrm{A} \overset{*}{\Rightarrow} \boldsymbol{w}_i \boldsymbol{w}_{i+1} \dots \boldsymbol{w}_j$. In other words, there is a sequence of rules that can be applied to the non-terminal A that yields $\boldsymbol{w}_i \boldsymbol{w}_{i+1} \dots \boldsymbol{w}_j$.

We further define **slashed** items of the form $[\mathrm{A}, i, j]/[\mathrm{B}, k, l]$. Intuitively, solving $[\mathrm{A}, i, j]/[\mathrm{B}, k, l]$ equates to determining whether A can derive $\boldsymbol{w}_i \dots \mathrm{B} \dots \boldsymbol{w}_j$ *assuming* that the non-terminal B already derives the substring $\boldsymbol{w}_k \dots \boldsymbol{w}_l$. More formally, $[\mathrm{A}, i, j]/[\mathrm{B}, k, l]$ is **realizable** if and only if $\mathrm{A} \overset{*}{\Rightarrow} \boldsymbol{w}_i \boldsymbol{w}_{i+1} \dots \boldsymbol{w}_{k-1} \mathrm{B} \boldsymbol{w}_{l+1 \dots j}$.

Naturally, $\boldsymbol{w} \in \mathbb{L}(\mathcal{G})$ if and only if the item $[\mathrm{S}, 1, n]$ is realizable, and determining realizability can be broken down recursively as follows:

Lemma 3.1. $[\mathrm{X}, i, j]$ *is realizable if and only if one of the following conditions is met:*

- ***Base case:*** *$j = i$ and $\mathrm{X} \to \boldsymbol{w}_i$ is a rule in the grammar.*

- ***Recursive case 1:*** *There exist a rule* $\mathrm{X} \to \mathrm{YZ}$ *and an index* k *such that* $[\mathrm{Y}, i, k-1]$ *and* $[\mathrm{X}, k, j]$ *are realizable items. There are* $O(|\mathcal{P}|N)$ *such guesses for* $O(|\mathcal{N}|N^2)$ *possible given items.*
- ***Recursive case 2:*** *There exists a* $[\mathrm{Y}, k, l]$ *such that* $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ *and* $[\mathrm{Y}, k, l]$ *are both realizable. There are* $O(|\mathcal{N}|N^2)$ *such guesses for* $O(|\mathcal{N}|N^2)$ *possible given items.*

Proof. The proof follows from our definitions. In the base case where $j = i$, X needs to derive exactly the symbol $\boldsymbol{w}_i$ in one-step without producing non-terminals (assuming a trim CFG with no useless non-terminals). In the recursive case, there exists otherwise a binary rule $\mathrm{X} \to \mathrm{YZ}$ such that Y and Z derive disjoint, consecutive substrings of $\boldsymbol{w}$. This statement is equivalent to stating there there exists a non-terminal Y such that Y derives some substring $\boldsymbol{w}_k \dots \boldsymbol{w}_l$, and X derives $\boldsymbol{w}$ where $\boldsymbol{w}_k \dots \boldsymbol{w}_l$ has been replaced by Y. ■

Lemma 3.2. $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ *is realizable if and only if one of the following conditions is met:*

- ***Base case:*** $k = i$, $l = j - 1$ *and there is a rule* $\mathrm{X} \to \mathrm{YZ}$ *in the grammar such that* $\mathrm{Z} \to \boldsymbol{w}_j$. *(and symmetric case)*
- ***Recursive case 1:*** *There exist a rule* $\mathrm{X} \to \mathrm{AB}$ *and an index* p *such that* $[\mathrm{A}, i, p-1]/[\mathrm{Y}, k, l]$ *and* $[\mathrm{B}, p, j]$ *are realizable items (and symmetric case). There are* $O(|\mathcal{P}|N)$ *such guesses for* $O(|\mathcal{N}|^2N^4)$ *possible given items.*
- ***Recursive case 2:*** *There exists a* $[\mathrm{Z}, p, q]$ *such that* $[\mathrm{X}, i, j]/[\mathrm{Z}, p, q]$ *and* $[\mathrm{Z}, p, q]/[\mathrm{Y}, k, l]$ *are both realizable. There are* $O(|\mathcal{N}|N^2)$ *such guesses for* $O(|\mathcal{N}|^2N^4)$ *possible given items.*

Proof. The proof follows the same structure as the proof of Lem. 3.1. ■

Lem. 3.1 and Lem. 3.2 state that an item is realizable if *there exists* some decomposition into realizable subproblems. Assuming some parallel model of computation, we can *guess* in parallel which of these decompositions is the correct one and then recursively verify the subproblems in parallel. This suggests natural parallel algorithms for checking realizability of items and slashed items, which we present in Alg. 1 and Alg. 2.

Algorithm 1 Solving an item $[\mathrm{X}, i, j]$

```
def f([X, i, k]):
  if i = j :
    return X → w_i ∈ P
  Guess an integer in x = {1, 2}
  if x = 1 :
    Guess a rule X → YZ ∈ P and k ∈ ℕ
    return f([Y, i, k − 1]) ∧ f([Z, k, j])
  else
    Guess an item [Y, k, l]
    return f([X, i, j]/[Y, k, l]) ∧ f([Y, k, l])
```

Algorithm 2 Solving an item $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$

```
def f([X, i, j]/[Y, k, l]):
  if k = i ∧ l = j − 1 :
    return ∃ X → YZ ∈ P such that Z → w_j ∈ P
  Guess an integer in x = {1, 2}
  if x = 1 :
    Guess a rule X → AB ∈ P and p ∈ ℕ
    return f([A, i, p − 1]/[Y, k, l]) ∧ f([B, p, j])
  else
    Guess an item [Z, p, q]
    return f([X, i, j]/[Z, p, q]) ∧ f([Z, p, q]/[Y, k, l])
```

Intuitively, the recursive function f defined in Alg. 1, Alg. 2 computes the realizability of items.

Theorem 3.1 (Correctness)**.** *Given a CFG $\mathcal{G}$ in CNF and $\boldsymbol{w} \in \Sigma^*$ of length n, $f([\mathrm{S}, 1, n]) = 1$ if and only if $\boldsymbol{w} \in \mathbb{L}(\mathcal{G})$.*

Proof. By definition, $\boldsymbol{w} \in \mathbb{L}(\mathcal{G})$ if and only if $[\mathrm{S}, 1, n]$ is realizable. By Lem. 3.1 and Lem. 3.2, the item $[\mathrm{S}, 1, n]$ is realizable if and only if there exists a decomposition of $[\mathrm{S}, 1, n]$ that respects the Lemmata 3.1 and 3.2. Our algorithm recursively guesses such decompositions, guaranteeing that we will compute a valid decomposition if it exists. ■

We now analyze the resources required to compute the recursive function f induced by Alg. 1 and Alg. 2. Our algorithm is based on a balanced decomposition of problems into subproblems of roughly equal size, which intuitively leads to a $\log(n)$-time procedure. Formally, we have the following well-known theorem for decomposing trees:

Theorem 3.2 (Jordan 1869)**.** *Given a tree with n vertices, there exists a vertex whose removal partitions the tree into two trees with each at most $n/2$ vertices.*

We rely on Thm. 3.2 to prove that Alg. 1 runs in a logarithmic number of recursive steps:

Corollary 3.1. *We can compute $f([\mathrm{S}, 1, n])$ in $\log(n) + \mathcal{O}(1)$ recursive steps $\forall \boldsymbol{w} \in \Sigma^*$ with $|\boldsymbol{w}| = n$.*

Proof. By Thm. 3.2, for any item there exists a balanced decomposition of the corresponding parse tree into two trees of roughly equal size, which can be represented by two items (the split is at the root) or a slashed item and an item (the split is not at the root). Assuming we can in parallel attend to all possible decompositions, we will necessarily guess the balanced one where subtrees have at most $n/2 + 1$ vertices. After i recursive steps, the current subtrees have at most $\frac{n}{2^i} + \mathcal{O}(1)$ nodes. Therefore, we will solve all base cases after at most $\log(n) + \mathcal{O}(1)$ steps. ■

In terms of space complexity, the bottleneck resides in solving an item $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ which occupies space $\mathcal{O}(n^4)$, and guessing a possible $[\mathrm{Z}, p, q]$ which could decompose this problem, which itself occupies space $\mathcal{O}(n^2)$, leading to a total space complexity of $\mathcal{O}(n^6)$. Combining both insights on time- and space-complexity, we can prove the following theorem:

Theorem 3.3. *Given a* CFL $\mathbb{L}$*, there exists a transformer with both causally-masked and non-masked attention layers, $\mathcal{O}(\log(n))$ looping layers and $\mathcal{O}(n^6)$ padding tokens that recognizes $\mathbb{L}$. That is,* $\mathrm{CFL} \subseteq \mathsf{mAHAT}^1_6 \subseteq \mathsf{AHAT}^1_7$.

Proof intuition. The construction will implement Alg. 1 and Alg. 2 on a transformer. Intuitively, we store padding tokens for all possible item / decomposition combinations, of which there are $\mathcal{O}(n^6)$ possibilities. We assume some ternary logic where padding tokens allocate space for an integer in $\{0, 1, \bot\}$, denoting that the item is non-realizable, realizable or not known yet to be realizable, respectively. All padding tokens initially store $\bot$. In the initial block of layers, padding tokens associated with a base case item of the form $[\mathrm{A}, i, i]$ can attend symbol representations to verify whether the base case is valid. In the inductive step, i.e for each looped block, padding tokens attend to the padding tokens associated with the decomposition and add 1 to the residual stream if they are both realizable, 0 if any of them is non-realizable, or $\bot$ if we have not computed yet the realizability of these items. It takes $\log(n)$ looping layers to populate the values of all items in their respective padding tokens due to Thm. 3.2. Finally, we can check whether there exists a padding token associated with $[\mathrm{S}, 1, n]$ that holds the value 1. Applying Lem. 2.1 yields inclusion in AHAT^1_7. The detailed proof is in App. B.1. ■

While we firstly prove that mAHATs with $\log(n)$-depth can recognize all CFLs, we still require $\mathcal{O}(n^6)$ padding tokens to do so which is intractable in practice. To this extent, we will see in the next section that for more restricted classes of CFLs, algorithms for recognition can leverage some grammar constraints such that a tractable number of padding tokens suffices. We unify insights from previous work to contrast the resources required to process different classes of CFLs.

4 UNAMBIGUITY REDUCES PADDING REQUIREMENTS FOR RECOGNITION

Intuitively, a general algorithm for recognizing an arbitrary CFL requires a large amount of padding because an arbitrary grammar can be highly ambiguous. Guessing how to decompose an arbitrary item requires a substantial amount of space. In this section, we show that *unambiguous* CFLs enable recognition with less padding.

A CFL is **unambiguous** if there is at most one possible derivation (i.e, parse tree) for any string. Unambiguity is a natural CFL feature of general interest. It has been shown that transformers struggle to parse ambiguous grammars in contrast to unambiguous grammars (Khalighinejad et al., 2023), while modern LMs struggle to process syntactically ambiguous natural language sentences (Liu et al., 2023). Moreover, modern parsers for programming languages such as LR parsers rely on deterministic (therefore unambiguous) CFLs to process inputs in linear time.

We will now present an algorithm for recognizing unambiguous grammars with a tractable space complexity but in $\log^2(n)$-time. Crucially, we are also able to implement this algorithm on AHATs with a tractable number of padding tokens.

4.1 A PATH SYSTEM FRAMEWORK FOR UNAMBIGUOUS CFL RECOGNITION

We now formulate recognition of unambiguous CFLs as a path system problem. Informally, a path system consists of initial nodes that are associated with either the value 1 or 0, and a relation $\mathcal{R}$ that characterizes how to connect nodes with edges. By associating base items of the form $[\mathrm{A}, i, i]$ to base nodes, general items of the form $[\mathrm{A}, i, j]$ to arbitrary nodes, and connecting nodes depending on the rules of the given grammar, computing the realizability of an item reduces to finding a path between its associated nodes and a base node. To this extent, we now present Chytil et al. (1991)'s path system framework for recognizing unambiguous grammars and transpose it to AHATs. This subsection assumes the grammars are in CNF.

We denote by $\mathcal{V}$ a set of nodes, each associated with a tuple $[\mathrm{A}, i, j]$. We denote by $\mathcal{T} \subseteq \mathcal{V}$ the **initial** set of vertices of the form $[\mathrm{A}, i, i]$ such that $\mathrm{A} \to \boldsymbol{w}_i \in \mathcal{P}$. $\mathcal{R}(x, y, z)$ is a relation on $\mathcal{V}$ where $\mathcal{R}(x, y, z) = 1$ if and only if z is associated with some tuple $[\mathrm{A}, i, j]$, x is associated with some tuple $[\mathrm{B}, i, k]$ and y is associated with some tuple $[\mathrm{C}, k, j]$ such that $\mathrm{A} \to \mathrm{BC} \in \mathcal{P}$. We denote by $\mathcal{C}(\boldsymbol{w}) \subseteq \mathcal{V}$ the smallest set containing $\mathcal{T}$ such that if $x, y \in \mathcal{C}(\boldsymbol{w})$ and $\mathcal{R}(x, y, z) = 1$ then $z \in \mathcal{C}(\boldsymbol{w})$, i.e $\mathcal{C}(\boldsymbol{w})$ is the closure of $\mathcal{T}$ with respect to the relation $\mathcal{R}$. Naturally, an intuitive way to think about $\mathcal{C}(\boldsymbol{w})$ is that it contains exactly the set of realizable elements, and the recognition problem is thus equivalent to determining whether the node associated with $[\mathrm{S}, 1, n]$is in the set $\mathcal{C}(\boldsymbol{w})$.

To compute $\mathcal{C}(\boldsymbol{w})$, we will refer to some graph-specific structures introduced in Chytil et al. (1991). We first define **dependency graphs**. Let $\mathcal{X} \subseteq \mathcal{V}$ a set of vertices that are **marked**, we denote by $\mathrm{DG}(\mathcal{X})$ the directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with respect to $\mathcal{X}$ where:

$$\mathcal{E} = \{(z, x) \mid z \notin \mathcal{X}, \mathcal{R}(x, y, z) = 1 \text{ or } \mathcal{R}(y, x, z) = 1 \text{ for some } y \in \mathcal{X}\} \tag{1}$$

Intuitively, assuming $\mathcal{X} \subseteq \mathcal{C}(\boldsymbol{w})$, the edge (z, x) can be interpreted as follows: $x \in \mathcal{C}(\boldsymbol{w})$ implies that $z \in \mathcal{C}(\boldsymbol{w})$. In fact, (z, x) being an edge signals that there is some vertex y associated with a realizable item such that $\mathcal{R}(x, y, z) = 1$. Thus, if x also is associated with a realizable item (i.e, is in the closure $\mathcal{C}(\boldsymbol{w})$), then z is a realizable item. The algorithm thus iteratively expands the known set of vertices to be associated with realizable items by computing the set of vertices that have a directed path to a marked node. We denote by $\textsc{Reach}(\mathcal{G})$ the nodes of the dependency graph $\mathcal{G}$ that have a directed path to a marked node in $\mathcal{G}$.

Chytil et al. (1991)'s procedure to compute $\mathcal{C}(\boldsymbol{w})$ is describe in Alg. 3.

The bottleneck in Alg. 3 is computing $\textsc{Reach}(\mathcal{D})$, i.e reachability queries on a directed, acyclic graph (DAG). Assuming unambiguity, we have the following powerful insight: There is at most one path between any pair of nodes in our DAG. If there are multiple paths from a node $[\mathrm{A}, i, j]$ to another node $[\mathrm{B}, k, l]$ there are then different derivations that can reduce $[\mathrm{A}, i, j]$ to $[\mathrm{B}, k, l]$, which contradicts the unambiguity condition. Therefore, for each node v the subgraph induced by nodes reachable from v becomes a *tree* rooted at v. Reachability queries on a tree reduce to evaluating the corresponding *Boolean formula*, where leaf nodes are assigned 1 if they correspond to realizable items and non-leaf nodes are assigned the $\vee$ operator. We rely on the following lemma to perform this procedure:

Algorithm 3 Algorithm for computing the closure $\mathcal{C}(\boldsymbol{w})$

1. We are given $\boldsymbol{w}$ and a grammar $\mathcal{G}$
2. Initialize $\mathcal{V}$ and $\mathcal{T}$ as defined previously
3. **def** $\mathcal{C}(\boldsymbol{w})$:
4. $\quad \mathcal{X} = \mathcal{T}$
5. $\quad$ **for** i in range $\log(n)$:
6. $\quad\quad \mathcal{D} = \mathrm{DG}(\mathcal{X})$
7. $\quad\quad \mathcal{X} = \textsc{Reach}(\mathcal{D})$
8. $\quad$ **return** $\mathcal{X}$

Lemma 4.1. *Let ϕ be a Boolean formula. Assume ϕ is represented in a transformer's residual stream as follows. For each leaf, there is a token that encodes its true/false value. For each function node, there is a token that encodes its type and one or two input arguments. Then, we can compute the value of each subformula (at its corresponding node) in $\mathcal{O}(\log(n))$ time.*

Proof intuition. Given the appropriate pointers, We implement (Rytter, 1985)'s parallel pebble game algorithm for evaluating Boolean formulas with $\mathcal{O}(\log(n))$ steps on transformers. The procedure operates in parallel at each node by iterating three steps $\mathcal{O}(\log(n))$ times: activate, square, and pebble. At each node where one child already has a value, activate sets `ptr` to the other child and sets a "conditional function" `condf`: $\{0,1\} \to \{0,1\}$ based on the node type and child's value. Square then operates at activated nodes, setting `condf` $=$ `ptr.condf`. Pebble operates at activated nodes, checking whether `ptr.value` is defined, and, if so, setting `value` $=$ `condf(ptr.value)`. Rytter (1985) show that this algorithm correctly evaluates each subformula within $\mathcal{O}(\log(n))$ steps. ■

We can now show how to simulate Alg. 3's procedure on transformers for unambiguous CFLs with $\mathcal{O}(\log(n)^2)$ looping layers and $\mathcal{O}(n^3)$ padding tokens.

Theorem 4.1. *Let* UCFL *be the classes of unambiguous* CFL*s. Then* UCFL $\subseteq \mathtt{mAHAT}^2_3 \subseteq \mathtt{AHAT}^2_4$.

Proof intuition. We will implement Alg. 3 on `mAHAT`s. Each item $[\mathrm{A}, i, j]$ (of which there are $\mathcal{O}(n^2)$) is assigned a padding token. For each item $[\mathrm{A}, i, j]$, there are $\mathcal{O}(n)$ ways to decompose it using a split index $k \in [n]$. For every potential edge between nodes associated with $[\mathrm{A}, i, j]$ and some $[\mathrm{B}, i, k]$ (or $[\mathrm{B}, k, j]$), we assign a padding token. We refer to the proof of Thm. 3.3 on how a padding token can compute and store their associated objects (non-terminals, string indices). As in Thm. 3.3, we use a ternary logic where padding tokens for nodes are at any step assigned an element in $\{0, 1, \perp\}$, denoting realizability, non-realizability or undefined (we do not know yet whether the item is realizable or not). Initially, all padding tokens store $\perp$.

Initially, padding tokens for nodes can check whether they are associated with base case items of the form $[\mathrm{A}, i, i]$. These padding tokens can then store 1 (item is realizable) or 0 (item is non-realizable) depending on whether $\mathrm{A} \to \boldsymbol{w}_i \in \mathcal{P}$.

In the iterative case, each padding token for an edge associated with items $[\mathrm{A}, i, j], [\mathrm{B}, i, k]$ can first check whether there exists a rule $\mathrm{A} \to \mathrm{BC}$ and if so, add to the residual stream $[\mathrm{C}, k+1, j]$. Crucially, there a finitely such many items (proportional to $|\mathcal{N}|$ as the splitting index k is fixed). Padding token for edges can attend to padding tokens associated with $[\mathrm{C}, k+1, j]$ and check whether any of them stores 1 denoting realizability. In that case, the padding token for that edge signals that the edge is now in the graph (following how we define edges in Eq. (1)). Padding tokens for nodes associated with items $[\mathrm{A}, i, j]$ can therefore attend to padding tokens for edges associated with $[\mathrm{A}, i, j], [\mathrm{B}, i, k]$, which yields the dependency graph.

Crucially, due to unambiguity, for each node v the subgraph induced by nodes reachable from v becomes a tree rooted at v. We then show how to binarize this tree. Reachability queries on a binary tree can be reduced to efficient evaluating Boolean formulas (Chytil et al., 1991). We invoke Lem. 4.1 to evaluate Boolean formulas in $\log(n)$ steps. The detailed proof is in App. B.2. ■

A notable example of an unambiguous CFL is the **Boolean formula value problem** (BFVP), the set of variable-free Boolean formulas that evaluate to 1. It is a canonical NC^1-complete language, i.e is

known to require $\log$-depth for recognition (Buss, 1987). Assuming formulas in postfix notation (e.g., $01\neg \wedge 0\vee$), Lem. 4.1 enables us to solve BFVP without any padding on $\log$-depth AHATs:

Lemma 4.2. *Let ϕ be a variable-free Boolean formula written in postfix notation. Then, taking ϕ as input, a causally-masked* AHAT *can evaluate ϕ after $\mathcal{O}(\log(n))$ steps.*

Proof. Each symbol representation can compute its position in the string by uniformly attending to the strict left context. For each symbol representation $\wedge$ or $\vee$ at some position i, we store the indices of the two previous symbol representations $i-1$ and $i-2$ via a feedforward network. For symbol representations $\neg$ for a position i, we add to the residual stream $i-1$. Symbol representations denoting 1 or 0 (representing leaf nodes in the binary tree) can add to the residual stream 0 or 1. We can therefore invoke Lem. 4.1 to solve this Boolean formula in $\log(n)$ steps. Strict causal-masking suffices due to postfix notation, as symbol representations associated with a binary (unary) operator need to attend to the symbol representations in the left context. ■

Dyck, the language of all correctly balanced strings of parentheses, is another canonical unambiguous CFL. This language is often used to model hierarchy and unbounded nesting in natural language. As shown in prior work, Dyck can in fact be recognized by constant-depth circuits and transformers (Hayakawa & Sato, 2024; Mix Barrington & Corbett, 1989; Hu et al., 2025; Weiss et al., 2021).

4.2 Unambiguous linear CFLs require less time and space

Finally, we show how another constraint we can impose on CFLs, namely linearity, further reduces the resources needed by transformers to recognize unambiguous CFLs. A **linear** CFL is characterized by a CFG where each rule in the grammar is the form $\mathrm{A} \to a\mathrm{B}$, $\mathrm{A} \to \mathrm{B}a$ or $\mathrm{A} \to a$. We assume grammars take this form in this subsection. While restricted, linear CFLs can already capture a wide range of features of context-freeness: Balanced counting can be modeled by the following linear CFL $\mathbb{L} = \{a^n b^n \mid n \geq 0\}$, and symmetry can be modeled by the linear CFL $\mathbb{L} = \{\boldsymbol{w}\boldsymbol{w}^R \mid \boldsymbol{w} \in \Sigma^*\}$.

We consider unambiguous linear [1] CFLs (ULCFLs), and use linearity to prove the following.

Theorem 4.2. ULCFL $\subseteq$ $\mathsf{mAHAT}^1_2 \subseteq \mathsf{AHAT}^1_3$.

5 Experiments

In contrast to prior empirical work that corroborates claims on the expressive power of transformers on sequence-to-sequence tasks (Delétang et al., 2023; Bhattamishra et al., 2020), we train transformers to recognize context-free languages of varying degrees of complexity:

- **Boolean formula value problem** (BFVP): The set of variable-free Boolean formulas that evaluate to 1. This CFL is known to be complete for NC^1 (Buss, 1987), i.e requires logarithmic time w.r.t. input length. We also consider the *postfix* notation version of the problem, where for example $1 \vee 0$ rewrites as $1\ 0\ \vee$. Parallel algorithms for BFVP typically rely on postfix notation (Buss, 1987; Buss et al., 1992).
- **Palindrome**: The language defined as $\mathbb{L} = \{\boldsymbol{w}\boldsymbol{w}^R | \boldsymbol{w} \in \Sigma^*\}$ for some alphabet Σ. In our case, we arbitrarily select an alphabet of size 2. This language is linear unambiguous and non-deterministic. Prior work has shown that fixed-depth transformers with hard attention can recognize this language (Hao et al., 2022).
- **Marked Palindrome**: This language simplifies Palindrome by extending strings with a marker between $\boldsymbol{w}$ and $\boldsymbol{w}^R$, which delimits at which index we reverse the string. In other words, $\mathbb{L} = \{\boldsymbol{w}\#\boldsymbol{w}^R | \boldsymbol{w} \in \Sigma^*\}$ where $\# \notin \Sigma$. This language is linear deterministic.
- **Dyck**: The language of all correctly balanced strings of parentheses of k types, which we denote by $\mathrm{D}(k)$. In our case, we consider $\mathrm{D}(1)$ and $\mathrm{D}(2)$. This language is non-linear and deterministic. Fixed-depth transformers can recognize $\mathrm{D}(k)$ for any k (Hayakawa & Sato, 2024; Weiss et al., 2021).

[1] There is a subtlety here: A CFL can be induced by both a non-linear unambiguous grammar and by a different linear, ambiguous grammar. Here we consider grammars that are *simultaneously* linear and unambiguous

These languages vary in complexity, allowing us test transformers' ability to learn context-free recognition constructions for languages of different difficulties. In particular, while Palindrome and $\mathrm{D}(k)$ languages can in principle be represented with a constant-depth transformer solution, BFVP requires more than constant depth (i.e., $\log$-depth), assuming $\mathrm{TC}^0 \neq \mathrm{NC}^1$. Thus, the performance of log-depth vs. constant-depth transformers on BFVP in particular is a good measure of whether transformers can learn to utilize the extra expressivity of log-depth when it is required.

Data. We use Anonymous (2025)'s length-constrained sampling algorithm for CFLs to generate strings for our datasets. For $\mathrm{D}(1)$, $\mathrm{D}(2)$, Palindrome and Marked Palindrome, negative samples are either random or adversarial. For BFVP, we sample negative strings as well-formed Boolean formulas that evaluate to 0. For BFVP we prefer to focus on a transformer's ability to correctly evaluate a Boolean formula rather than determining if an input string is a well-formed formula, which is already emcompassed by $\mathrm{D}(k)$. Our test set contains strings over longer lengths than strings in our training set to evaluate the transformer's ability to *generalize* on out-of-distribution strings.

Models and Training Procedure. We train causally masked looped transformers with no positional embeddings. Our transformers has 1.2 million parameter budget. We use the AdamW optimizer (Loshchilov & Hutter, 2019) and binary cross-entropy loss, considering runs across 5 different seeds.

Table 2: Mean accuracy ($\pm$ standard deviation) by language and transformer type across seeds.

	Test accuracy on in-distribution strings		Test accuracy on out-of-distribution strings	
Language	Fixed-depth	$\log(n)$ looping	Fixed-depth	$\log(n)$ looping
BFVP	0.97 ± 0.01	0.98 ± 0.00	0.88 ± 0.01	0.91 ± 0.01
BFVP (postfix)	0.95 ± 0.01	0.98 ± 0.00	0.87 ± 0.01	0.91 ± 0.01
Palindrome	0.94 ± 0.01	0.93 ± 0.01	0.79 ± 0.03	0.72 ± 0.03
Marked palindrome	0.97 ± 0.01	0.98 ± 0.01	0.59 ± 0.19	0.66 ± 0.18
$\mathrm{D}(1)$	0.98 ± 0.00	0.98 ± 0.00	0.94 ± 0.02	0.93 ± 0.01
$\mathrm{D}(2)$	0.98 ± 0.02	0.99 ± 0.00	0.83 ± 0.08	0.90 ± 0.08

Results. By a small margin, looping improves accuracy on both the infix and postfix version of BFVP. The failure of fixed-depth transformers is consistent with the fact that fixed-depth transformers cannot solve BFVP assuming $\mathrm{TC}^0 \neq \mathrm{NC}^1$, and the benefit of log-depth is generally in-line with our results that $\log$-depth enables recognition of CFLs. In contrast, for Palindrome and $\mathrm{D}(1)$, looping does not improve accuracy, which is supported by the fact that these languages already have fixed-size solutions (Hao et al., 2022; Hayakawa & Sato, 2024) and thus recognition does not need $\log$-depth. For $\mathrm{D}(2)$ and Marked Palindrome, looping seems to improve generalization even though these languages also have log-depth transformer constructions. Interestingly, looping seems to help generalization on $\mathrm{D}(2)$. Moreover, it is perhaps surprising that fixed-depth transformers already generalize so well on BFVP.

6 Discussion and Conclusion

We show that transformers with $\log$-depth can recognize general CFLs if they can use padding tokens (Merrill & Sabharwal, 2025b). In addition, we characterize unambiguity and linearity as CFL features that can reduce the amount of padding needed by transformers for recognition. These results reveal one way that transformers with limited depth can recognize CFLs and predict ambiguity in language could be a hurdle for transformers to process, as suggested in previous empirical work (Khalighinejad et al., 2023; Liu et al., 2023). While it is not possible to improve our $\log$-depth recognition algorithm to fixed depth unless $\mathrm{TC}^0 = \mathrm{NC}^1$, our padding bounds are not known to be tight. Therefore, future work could find more padding-efficient transformer constructions for recognizing general CFLs, or subclasses thereof. Additionally, it would be interesting to consider the psycholinguistic implications of our results for comparing how humans and LMs process language and syntax. It is believed that CFLs are too weak to model natural language (Shieber, 1988), and that mildly context-sensitive formalisms such as tree-adjoining grammars (TAGs) are a better prospect to model natural language (Joshi, 1985; Bordihn, 2004). Future work could therefore focus on analyzing transformers' ability to recognize languages induced by TAGs (TALs).

REFERENCES

Anonymous. Empirically testing expressivity bounds for neural network architectures. 2025.

David Arps, Younes Samih, Laura Kallmeyer, and Hassan Sajjad. Probing for constituency structure in neural language models, 2022. URL https://arxiv.org/abs/2204.06201.

David A. Mix Barrington and Denis Thérien. Finite monoids and the fine structure of nc1. *J. ACM*, 35(4):941–952, October 1988. ISSN 0004-5411. doi: 10.1145/48014.63138. URL https://doi.org/10.1145/48014.63138.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages, 2020. URL https://arxiv.org/abs/2009.11264.

Henning Bordihn. *Mildly Context-Sensitive Grammars*, pp. 163–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-39886-8. doi: 10.1007/978-3-540-39886-8_8. URL https://doi.org/10.1007/978-3-540-39886-8_8.

S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992. doi: 10.1137/0221046. URL https://doi.org/10.1137/0221046.

S. R. Buss. The boolean formula value problem is in alogtime. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pp. 123–131, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912217. doi: 10.1145/28395.28409. URL https://doi.org/10.1145/28395.28409.

David Chiang. Transformers in uniform tc^0, 2025. URL https://arxiv.org/abs/2409.13629.

Michal Chytil, Maxime Crochemore, Burkhard Monien, and Wojciech Rytter. On the parallel recognition of unambiguous context-free languages. *Theoretical Computer Science*, 81(2):311–316, 1991. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(91)90199-C. URL https://www.sciencedirect.com/science/article/pii/030439759190199C.

John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, USA, 1969. ISBN B0007F4UOA.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy, 2023. URL https://arxiv.org/abs/2207.02098.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL https://doi.org/10.1145/362007.362035.

Yiding Hao, Dana Angluin, and Robert Frank. Formal language recognition by hard attention transformers: Perspectives from circuit complexity, 2022. URL https://arxiv.org/abs/2204.06618.

Daichi Hayakawa and Issei Sato. Theoretical analysis of hierarchical language recognition and generation by transformers without positional encoding, 2024. URL https://arxiv.org/abs/2410.12413.

John Hewitt and Christopher D. Manning. A structural probe for finding syntax in word representations. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4129–4138, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1419. URL https://aclanthology.org/N19-1419/.

Michael Y. Hu, Jackson Petty, Chuan Shi, William Merrill, and Tal Linzen. Between circuits and chomsky: Pre-pretraining on formal languages imparts linguistic biases, 2025. URL https://arxiv.org/abs/2502.19249.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL https://arxiv.org/abs/2406.00515.

Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869. URL http://eudml.org/doc/148084.

Aravind K. Joshi. *Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?*, pp. 206–250. Studies in Natural Language Processing. Cambridge University Press, 1985.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. 1965. URL https://api.semanticscholar.org/CorpusID:61491815.

Ghazal Khalighinejad, Ollie Liu, and Sam Wiseman. Approximating CKY with transformers. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14016–14030, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.934. URL https://aclanthology.org/2023.findings-emnlp.934/.

Klaus-Jörn Lange and Peter Rossmanith. Characterizing unambiguous augmented pushdown automata by circuits. In Branislav Rovan (ed.), *Mathematical Foundations of Computer Science 1990*, pp. 399–406, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-47185-1.

Alisa Liu, Zhaofeng Wu, Julian Michael, Alane Suhr, Peter West, Alexander Koller, Swabha Swayamdipta, Noah A. Smith, and Yejin Choi. We're afraid language models aren't modeling ambiguity, 2023. URL https://arxiv.org/abs/2304.14399.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought, 2024. URL https://arxiv.org/abs/2310.07923.

William Merrill and Ashish Sabharwal. A little depth goes a long way: The expressive power of log-depth transformers, 2025a. URL https://arxiv.org/abs/2503.03961.

William Merrill and Ashish Sabharwal. Exact expressive power of transformers with padding, 2025b. URL https://arxiv.org/abs/2505.18948.

William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022. doi: 10.1162/tacl_a_00493. URL https://aclanthology.org/2022.tacl-1.49/.

David A. Mix Barrington and James Corbett. On the relative complexity of some languages in nc1. *Information Processing Letters*, 32(5):251–256, 1989. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(89)90052-5. URL https://www.sciencedirect.com/science/article/pii/0020019089900525.

Libo Qin, Qiguang Chen, Xiachong Feng, Yang Wu, Yongheng Zhang, Yinghui Li, Min Li, Wanxiang Che, and Philip S. Yu. Large language models meet nlp: A survey, 2024. URL https://arxiv.org/abs/2405.12819.

Peter Rossmanith and Wojciech Rytter. Observations on log (n) time parallel recognition of unambiguous cfl's. *Information Processing Letters*, 44(5):267–272, 1992. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(92)90212-E. URL https://www.sciencedirect.com/science/article/pii/002001909290212E.

Walter L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21(2):218–235, 1980. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(80)90036-7. URL https://www.sciencedirect.com/science/article/pii/0022000080900367.

Wojciech Rytter. The complexity of two-way pushdown automata and recursive programs. In Alberto Apostolico and Zvi Galil (eds.), *Combinatorial Algorithms on Words*, pp. 341–356, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. URL https://link.springer.com/chapter/10.1007/978-3-642-82456-2_24.

Stuart M. Shieber. *Evidence Against the Context-Freeness of Natural Language*, pp. 79–89. Springer Netherlands, Dordrecht, 1988. ISBN 978-94-009-2727-8. doi: 10.1007/978-94-009-2727-8_4. URL https://doi.org/10.1007/978-94-009-2727-8_4.

Lena Strobl. Average-hard attention transformers are constant-depth uniform threshold circuits, 2023. URL https://arxiv.org/abs/2308.03212.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers, 2021. URL https://arxiv.org/abs/2106.06981.

Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189–208, 1967. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(67)80007-X. URL https://www.sciencedirect.com/science/article/pii/S001999586780007X.

Haoyu Zhao, Abhishek Panigrahi, Rong Ge, and Sanjeev Arora. Do transformers parse while predicting the masked word? In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 16513–16542, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.1029. URL https://aclanthology.org/2023.emnlp-main.1029/.

A Extended background

A.1 Transformer models

We introduce in this section our idealization of the transformer architecture and the underlying assumptions we make.

A.1.1 Fixed-size transformers

An L-layer **transformer** of width D is a mapping $\mathsf{T}\colon \Sigma^* \to \left(\mathbb{R}^D\right)^+$:

$$\mathsf{T} \stackrel{\text{def}}{=} \mathcal{L}^{(L)} \circ \cdots \circ \mathcal{L}^{(1)} \circ \mathsf{embed} \tag{2}$$

The **input encoding function** $\mathsf{embed}\colon \Sigma^* \to \left(\mathbb{R}^D\right)^+$ applies an injective position-wise embedding function to each symbol in the input string $\boldsymbol{w}$. We assume the existence of BOS and EOS symbols, distinct symbols that are placed at the beginning and end of every input string, respectively.

$\mathcal{L}^{(\ell)}$ for $\ell \in [L]$ denotes a **transformer layer**—a mapping $\mathcal{L}^{(\ell)}\colon \left(\mathbb{R}^D\right)^+ \to \left(\mathbb{R}^D\right)^+$ that updates the hidden states of the symbols. The components of a transformer layer are the layer normalization LN, the attention layer ${\boldsymbol{f}_{\mathsf{att}}}^{(\ell)}$ and feedforward network $\mathbf{F}^{(\ell)}$. Concretely:

$$\mathcal{L}^{(\ell)} \stackrel{\text{def}}{=} \mathbf{F}^{(\ell)} \circ {\boldsymbol{f}_{\mathsf{att}}}^{(\ell)} \circ \mathsf{LN}^{(\ell)} \tag{3}$$

We recall layer-normalization maps a vector $x \in \mathbb{R}^n$ of some dimension n to $\frac{x'}{\|x'\|}$ where $x' = x - \frac{\sum_{x_i \in x} x_i}{n}$. We assume **projected pre-norm** as in Merrill & Sabharwal (2024). In standard pre-norm, we apply a layer-normalization to the entire hidden state of each symbol, while projected pre-norm applies layer-normalization to different parts of the vector individually. This will allow us to store bounded mappings of specific elements in the hidden state.

$\mathbf{F}^{(\ell)}\colon \left(\mathbb{R}^D\right)^+ \to \left(\mathbb{R}^D\right)^+$ is a position-wise function that applies the same feedforward network to every symbol of the sequence. It is parametrized by weight matrices of the form $\boldsymbol{W} \in \mathbb{R}^{D \times m}$ and $\boldsymbol{U} \in \mathbb{R}^{m \times \mathbb{R}^D}$. A feedforward network $\mathbf{F}^{(\ell)}$ can nest functions of the form $\boldsymbol{U}\mathrm{ReLU}(\boldsymbol{W} z_i)$ where z_i is an intermediate value.

The **attention mechanism** is defined by the function $\boldsymbol{f}_{\texttt{att}}{}^{(\ell)}\colon \left(\mathbb{R}^D\right)^+ \to \left(\mathbb{R}^D\right)^+$. We denote by $\boldsymbol{k}_i^{(\ell)}$, $\boldsymbol{q}_i^{(\ell)}$, $\boldsymbol{v}_i^{(\ell)}$ the key, query and value vectors respectively for symbol i at layer ℓ. $\boldsymbol{f}_{\texttt{att}}{}^{(\ell)}$ is defined as follows:

$$\boldsymbol{f}_{\texttt{att}}{}^{(\ell)}((x_1, \cdots, x_T)) \stackrel{\text{def}}{=} (y_1, \cdots, y_T) \tag{4a}$$

$$y_i \stackrel{\text{def}}{=} x_i + \sum_{i' \in m(i)} s_{i'} \boldsymbol{v}_{i'}^{(\ell)} \tag{4b}$$

$$s = \texttt{proj}(\{\texttt{score}(\boldsymbol{k}_{i'}^{(\ell)}, \boldsymbol{q}_i^{(\ell)})\}) \tag{4c}$$

$m(i)$ is a set that defines the **masking** used by the transformer. For instance, $m(i) = \{i' \mid i' < i\}$ refers to strict causal masking and $m(i) = [|\boldsymbol{w}|]$ refers to no masking. $\texttt{score}$ is a scoring function that maps two vectors of the same size to a scalar. Typically, the dot-product score is used with $\texttt{score}(x_1, x_2) = \langle x_1, x_2 \rangle$.

$\texttt{proj}$ is a projection function that normalizes the scores into weights for the symbol values.

Throughout layers, the hidden state y_i of a symbol at position i continuously evolves as it cumulatively adds up the outputs of the attention mechanism. We call this cumulative sum y_i over layers the **residual stream** at i.

Following previous work, we assume an **averaging hard attention** transformer ($\texttt{AHAT}$) which causes the probability mass to be concentrated on the symbols that maximize the attention score. It is a slightly idealized version of the standard soft-attention that has enabled theoretical analyses on the expressive power of $\texttt{AHAT}$s (Merrill et al., 2022; Strobl, 2023).

Formally, we assume $\texttt{proj} = \mathrm{hardmax}$, where:

Definition A.1. ***Averaging hard attention*** *is computed with the* $\mathrm{hardmax}$ *projection function:*

$$\mathrm{hardmax}\,(\boldsymbol{x})_d \stackrel{\text{def}}{=} \begin{cases} \frac{1}{m} & \textbf{\textit{if}}\ d \in \mathrm{argmax}\,(\boldsymbol{x}) \\ 0 & \textbf{\textit{otherwise}} \end{cases} \tag{5}$$

for $d \in [D]$*, where* $\boldsymbol{x} \in \mathbb{R}^D$ *and* $m \stackrel{\text{def}}{=} |\,\mathrm{argmax}\,(\boldsymbol{x})\,|$ *is the cardinality of the argmax set.*

Recognition. A transformer is a $\left(\mathbb{R}^D\right)^+$-valued function. To link this to language recognition, we use the representations computed by a transformer for binary classification of strings. We denote by $\boldsymbol{x}_{\text{EOS}}^L$ the hidden state of EOS at the end of the forward pass of T. Typically, string recognition is based on the final EOS representation $\boldsymbol{x}_{\text{EOS}}^L$ as EOS is the only symbol that is able to access information about every single symbol throughout all (assuming causal masking). This allows us to define a transformer's language. This is usually defined based on a linear classifier:

$$\mathbb{L}(\mathsf{T}) \stackrel{\text{def}}{=} \{\boldsymbol{w} \in \Sigma^* \mid \boldsymbol{\theta}^\top \boldsymbol{x}_{\text{EOS}}^L > 0\} \tag{6}$$

Precision. Following previous work (Merrill & Sabharwal, 2025b; 2024; 2023), we assume $\log$-precision transformers, i.e we allow the transformer to manipulate values that can be represented with $\mathcal{O}(\log(n))$ bits for an input of length n. It is a minimally extended idealization that enables the transformer to store indices (and thus access) and perform sums over an unbounded number of symbol: two crucial capabilities for our constructions.

A.1.2 Looping and Padding

Looped transformers dynamically scale the number of layers with respect to the length of the input. We formalize this notion based on Merrill & Sabharwal (2025a).

Definition A.2. *Let* T *be a transformer. We denote by* $\langle A, B, C \rangle$ *a partition of layers such that* A *is the* ***initial block*** *of layers,* B *is the* ***looped*** *block of layers and* C *is the* ***final block*** *of layers.* T *is* $d(n)$***-looped*** *if upon a forward pass with an input of length* n*,* B *is repeated* $\mathcal{O}(d(n))$ *times.*

The number of computations performed by self attention is inherently quadratic in the length of the string. One can dynamically increase this scaling the *padding space* (Merrill & Sabharwal, 2025b).

Definition A.3. *Let* T *be a transformer.* T *is* $w(n)$***-padded*** *if* $\mathcal{O}(w(n))$ *padding tokens are appended to the end of the string when computing the contextual representations of a length-*n *input.*

We assume EOS is still placed at the end of the padding tokens in padded transformers.

We call **universal transformers** the set of standard transformers that may be extended with looping and/or padding. Crucially, dynamically scaling number of layers and padding tokens in transformers is analogous to scaling time and space in classical models of computation such as Boolean circuits (Merrill & Sabharwal, 2025b).

Allowing for different looping and padding budgets results in different classes of transformers. We adopt naming conventions of these models from Merrill & Sabharwal (2025b). We denote by AHAT^d_k the class of languages recognized by averaging hard-attention transformers with $\mathcal{O}(\log^d(n))$-looping, $\mathcal{O}(n^k)$-padding and strict causal masking. We further use the notation uAHAT to refer to refer to average hard-attention transformers with strict causal masking with no masking, and mAHAT^d_k when induction heads may or may not employ strict causal masking.

We have the following convenient result on simulating non-masked attention layers with causally masked attention layers by adding a linear amount of padding.

Lemma A.1 (Merrill & Sabharwal (2025b)). $\mathsf{uAHAT}^d_k \subseteq \mathsf{mAHAT}^d_k \subseteq \mathsf{AHAT}^d_{\max(k+1,1)}$

A.1.3 Memory management

A desirable feature we will require for many of the proofs in this paper is the ability to correctly *add*, *retrieve*, *reset* or *remove* specific values from the residual stream. However, growing the number of layers with the input length raises the risk of earlier outputs interfering with later computations (Merrill & Sabharwal, 2025a). This introduces the need for some notion of *memory management*.

Recall the residual stream is a vector of D values with $\log$-precision. We define as **cell** as an index $i \in$ of the residual stream, i.e a placeholder for some value. In our proofs, as the hidden dimension D of a residual stream is fixed, we may freely allocate new cells in the initial and final block of layers as they have a fixed number of layers. However, we may not allocate new cells in layers of the looping block as we assume a fixed-size for the residual streams.

To this extent, we can as per Merrill & Sabharwal (2025a) easily clear out stored values in cells and replace them with new ones when needed.

A.1.4 Layer-norm hash

We will often use **layer-norm hash** building block (Merrill & Sabharwal, 2024). It is particularly useful for equality checks between values across different symbols, especially with a potentially unbounded number of queries and keys.

Definition A.4 (Merrill & Sabharwal, 2024). *Given a scalar $z \in \mathbb{R}$, its **layer-norm hash** is $\phi(z) \stackrel{\text{def}}{=} \langle z, 1, -z, -1\rangle / \sqrt{z^2+1}$.*

Crucially, layer-norm hash is scale invariant, and $\phi(q)\dot{\phi}(k) = 1$ if and only if $q = k$. In other words, the inner product of scalars q and k, even if computed at different positions i and j, respectively, allows us to check for the equality of q and k. Layer-norm hash thus allows us to perform equality checks over elements of residual streams at different positions.

With some abuse of notation, we allow for layer-norm hash to operate on vectors by applying the hash to every element of that vector. We thus for instance may write $\phi([\![\mathrm{A}]\!])$.

Using the layer-norm hash, we will rely in our proofs on several building blocks known to be implementable by AHATs such as performing equality-checks, division and modulo, counting over several positions in the left context (Merrill & Sabharwal, 2024; 2025a;b).

B Transformer constructions proofs

B.1 General CFL recognition on transformers

Theorem 3.3. *Given a* CFL $\mathbb{L}$*, there exists a transformer with both causally-masked and non-masked attention layers, $\mathcal{O}(\log(n))$ looping layers and $\mathcal{O}(n^6)$ padding tokens that recognizes $\mathbb{L}$. That is,* CFL $\subseteq \mathsf{mAHAT}^1_6 \subseteq \mathsf{AHAT}^1_7$.

Proof. We store padding tokens for each possible item (of the form $[\mathrm{X}, i, j]$ or $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$) and each possible way to decompose that item. There are $\mathcal{O}(n^6)$ such tokens: In the worst case, we are solving an item $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ and are guessing an item $[\mathrm{Z}, p, q]$ that decomposes that problem. Intuitively, if a padding token aims to solve the item $[\mathrm{X}, i, j]$ and holds as decomposition $[\mathrm{Y}, k, l]$, we attend to the padding tokens which solve $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ and $[\mathrm{Y}, k, l]$. Due to Thm. 3.2, if $[\mathrm{S}, 1, n]$ is realizable then there exists a padding token with associated item $[\mathrm{S}, 1, n]$ such that its value will be computed after $\mathcal{O}(\log(n))$ steps.

Storing items and their decomposition in padding tokens: We first detail how, without positional embeddings, each padding token can store a unique item / decomposition combination.

We have $\mathcal{O}(n^6)$ padding tokens, each associated with 1) an item to solve (for instance, $[\mathrm{A}, i, j]$) and 2) a tentative way to decompose this item (for instance, $[\mathrm{B}, i, k]$ and $[\mathrm{C}, k, j]$). Each padding token is distinguished by their unique position, which can be computed in one causally-masked attention layer by uniformly attending over the strict left context (Merrill & Sabharwal, 2024). .

We now sketch the intuition behind how each padding token can unpack from this position the corresponding item and decomposition.

AHATs can compute Euclidean divisions and modulo at some position i for integers smaller than i (Merrill & Sabharwal, 2025a). Furthermore, each padding token can add to their residual stream $\phi(n)$ by uniformly attending only to symbol representations on the strict left and setting

To this extent, for all padding tokens at some position i, we will *unpack* the $\mathcal{O}(\log(n^6)) = \mathcal{O}(6\log(n))$ bits in $\phi(i)$ into 1) the corresponding item to solve 2) its potential decomposition. A string index $j \in [n]$ requires $\log(n)$ bits to store, while an element of a finite set (for instance, a non-terminal $\in \mathcal{N}$) requires a constant number of bits (for instance, $\log(|\mathcal{N}|)$ bits). We can first establish arbitrarily which bits of the integer correspond to which elements of the item and decomposition (non-terminals, string indices). We can then iteratively 1) mask the first bits from the least significant bit (LSB), and then extract an index or a non-terminal (by performing modulo on i) 2) shift the binary representation of i towards the LSB to (by performing division on i) to then extract the following number. Shifting (masking) $\log(n)$ bits is done by dividing (taking modulo) the integer representation by n, which is stored in the residual stream.

Thus, we have a way for padding tokens to store the encoding of the item they are solving (for instance, $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$) and the encoding of objects that decompose that item (for instance, $[\mathrm{Z}, p, q]$).

We will now detail how to compute the realizability of items associated with these padding tokens. We consider items of the form $[\mathrm{X}, i, j]$, solving items of the form $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ is the exact same idea.

Padding tokens allocate space for an element of $\{0, 1, \bot\}$, which describes whether the associated item is realizable, not realizable, or not known yet to be realizable. Padding tokens initially all store $\bot$.

Base case: Items that form a base case are of type $[\mathrm{X}, i, i]$. A feedforward network can for each padding token first check that both indices are the same (i.e $i = j$). With an attention layer, we can then retrieve and add to the residual stream the symbol $\boldsymbol{w}_i$ for a given base case item $[\mathrm{X}, i, i]$ as follows. Every symbol representation can add to its residual stream its position in the string by uniformly attending with a causally-masked attention layer to all symbol representations in the past and counting them. An equality check via dot product between a padding token's stored index and the position of the symbol's representation enables such padding tokens to attend to relevant symbols of the string. Setting as value the one-hot encoding of the symbol $[\![\boldsymbol{w}_i]\!]$, we can add to the residual stream the relevant symbol $\boldsymbol{w}_i$. Finally, a feedforward network can add to the residual stream 1 if $\mathrm{X} \to \boldsymbol{w}_i$ is a valid rule and otherwise 0: A mapping between two finite sets $\mathcal{N} \times \Sigma \to \{0, 1\}$ can be computed by a feedforward network.

Induction step: Recall a padding token stores 1) an item to solve (for instance, $[\mathrm{X}, i, j]$) and 2) a set of objects that enable us to decompose that item (for instance, $[\mathrm{Y}, k, l]$). Given $[\mathrm{X}, i, j]$, $[\mathrm{Y}, k, l]$, a feedforward network adds the encodings of $[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$ and $[\mathrm{Y}, k, l]$ to the residual stream via a feedforward network. Otherwise, if a padding token is associated with $[\mathrm{X}, i, j]$, $\mathrm{X} \to \mathrm{YZ}$ and k, we add $[\mathrm{Y}, i, k-1]$ and $[\mathrm{Z}, k, j]$ to the residual stream via a feedforward network. In the latter case, a

feedforward network can also ensure the rule $\mathrm{X} \to \mathrm{YZ}$ is in the grammar, and store 2 in the residual stream (denoting non-realizability) if the rule is not in the grammar.

Finally, with one attention layer and a feedforward network, we can attend to all padding tokens that aim to solve the first subproblem ($[\mathrm{X}, i, j]/[\mathrm{Y}, k, l]$) and copy the integer in the allocated cell for realizability. We also perform the same procedure for the second subproblem to solve.

We compute the realizability of the current item via an extension of standard Boolean logic to handle the case where padding tokens have not yet computed the realizability of their associated item. We do not elicit the standard rules of propositional logic for brevity.

P	Q	$P \wedge Q$
1	$\perp$	$\perp$
$\perp$	1	$\perp$
0	$\perp$	0
$\perp$	0	0
$\perp$	$\perp$	$\perp$

Table 3: Truth table for ternary logic
that handles items that have not been solved yet

Crucially, a feedforward network can compute this mapping as it is between two finite sets.

After at most $\log(n)$ steps (Jordan, 1869), some padding token aiming to solve an item $[\mathrm{A}, i, j]$ will necessarily store 1 if and only if $[\mathrm{A}, i, j]$ is realizable: There exists some balanced decomposition represented by two padding tokens that we can attend to and store the realizability of their associated items.

Recognition step: The EOS token can uniformly attend to all padding tokens that encode the item $[\mathrm{S}, 1, n]$ (we can add $\mathrm{S}, 1$ and n to the residual stream beforehand) item and ensure one of them holds 1, denoting realizability. ■

B.2 Unambiguous CFL recognition on transformers

Lemma 4.1. *Let ϕ be a Boolean formula. Assume ϕ is represented in a transformer's residual stream as follows. For each leaf, there is a token that encodes its true/false value. For each function node, there is a token that encodes its type and one or two input arguments. Then, we can compute the value of each subformula (at its corresponding node) in $\mathcal{O}(\log(n))$ time.*

Proof. We are given a Boolean formula represented in padding tokens as follows: Padding tokens associated with non-leaf nodes have pointers to two children nodes (by storing the encoding of their associated padding tokens), padding tokens associated with leaf nodes are already assigned either 1 or 0. Each padding token for a non-leaf node also internally stores an integer representation of either $\wedge$ or $\vee$. We can then implement Rytter (1985)'s parallel pebble game algorithm for evaluating Boolean formulas in $\mathcal{O}(\log(n))$ steps (assuming there are n leaves).

The parallel pebbling game consists of three steps which are repeated $\mathcal{O}(\log(n))$ times: ACTIVATE, SQUARE and PEBBLE. We introduce each operation and detail how to perform them on AHATs.

ACTIVATE takes a non-leaf node and adds a pointer to its left son if and only if its right son is associated with a realizable item (and vice-versa). On AHATs, if a padding token p_1 stores the encodings of padding tokens p_2 and p_3 according to the dependency graph, p_1 can attend to p_2 and p_3 with two different separate layers via equality check can store their realizability. If p_2 is realizable, p_1 stores a pointer to p_3 and vice-versa.

SQUARE computes the one-step closure of ACTIVATE: if the node v_1 points to the node v_2, and v_2 points to the node v_3, SQUARE assigns v_3 to v_1 (instead of v_2). In an attention layer, if a padding token p_1 stores the encoding of a padding token p_2 and p_2 stores the encoding of a padding token p_3, the p_2 can set as value the encoding of p_3, and p_1 can attend to p_2 via equality check, and then copy the encoding of p_3.

PEBBLE sets the current node to realizable if the node it points to is itself realizable. A padding token p_1 storing a pointer to p_2 can thus attend to it via equality-check and copy its realizability.

The binary dependency graph has $\mathcal{O}(n+m)$ nodes (because we added m extra nodes to make the graph binary). Equivalently, we require $\mathcal{O}(n+m)$ padding tokens. As we populate nodes of the tree from the leaves to the root in parallel, it takes $\mathcal{O}(\log(n+m))$ looping layers to populate the tree (Chytil et al., 1991; Rytter, 1985). ■

Theorem 4.1. *Let* UCFL *be the classes of unambiguous* CFL*s. Then* UCFL $\subseteq \mathtt{mAHAT}^2_3 \subseteq \mathtt{AHAT}^2_4$.

Proof. Each item $[\mathrm{A}, i, j]$ is associated with a padding token. Each potential edge between nodes representing items $[\mathrm{A}, i, j], [\mathrm{B}, i, k]$ is associated with a padding token. There are $\mathcal{O}(n^3)$ such padding tokens. The procedure for padding tokens to compute and store these objects is the same as in App. B.1. We unpack the bits of the position of the padding token. We can also then apply a feedforward network to obtain the associated objects.

Each padding token for nodes allocates space to store an element in $\{0, 1, \perp\}$ that describes whether 1) the item is realizable 2) the item is not realizable 3) we have not computed realizability yet. Alg. 3's algorithm computes whether items are part of the closure $\mathcal{C}(\boldsymbol{w})$ (i.e, are realizable) or not.

Initial items: A padding token for node can check whether its associated item is of the form $[\mathrm{A}, i, i]$ via a feedforward network that checks the indices are the same. For all such padding tokens, another feedforward network adds 1 to the residual stream if and only if $\mathrm{A} \to \boldsymbol{w}_i \in \mathcal{P}$ to signal the realizability of that item. It otherwise signals the item is not realizable by adding 0 to the residual stream. We can do exactly as in the base case of App. B.1.

Creating the dependency graph: Padding tokens for edges store items of the form $[\mathrm{A}, i, j], [\mathrm{B}, i, k]$. There are finitely many $[\mathrm{C}, k+1, j]$ such that $\mathrm{A} \to \mathrm{BC} \in \mathcal{P}$ (proportionally many in $|\mathcal{N}|$), which can be added to the residual stream via a feedforward network. According to Eq. (1), there is an edge between nodes associated with $[\mathrm{A}, i, j]$ and $[\mathrm{B}, i, k]$ if and only if there is a $[\mathrm{C}, k+1, j]$ such that $[\mathrm{C}, k+1, j]$ is realizable (i.e, corresponding padding token stores 1 in residual stream) and $\mathrm{A} \to \mathrm{BC} \in \mathcal{P}$. The padding token for edge associated with $[\mathrm{A}, i, j], [\mathrm{B}, i, k]$ can attend to such padding tokens and check whether any of them stores 1. If so, the padding token signals that the edge between nodes for $[\mathrm{A}, i, j]$, [B and $i, k]$ exists.

Binarization: Due to unambiguity, there is at most one path between any pair of nodes in the dependency graph. If there are multiple paths from a node $[\mathrm{A}, i, j]$ to another node $[\mathrm{B}, k, l]$ there are then different derivations that can reduce $[\mathrm{A}, i, j]$ to $[\mathrm{B}, k, l]$, which contradicts the unambiguity condition. Evaluating reachability queries on a tree reduces to solving the Boolean formula induced by this tree where leaf nodes are assigned 1 or 0 depending on if they are associated with realizable items and non-leaf nodes are assigned the $\vee$ operator.

However, to efficiently evaluate this Boolean expression, we require a *binary* tree where each node has at most two children. For every block of looping layers, we consider a binarization of the dependency graph as follows.

The binarization assumes we have $\mathcal{O}(n^3)$ additional padding tokens appended to the input, i.e $\mathcal{O}(n^2)$ extra padding tokens for every node.). Note that asymptotically, appending $\mathcal{O}(n^3)$ padding tokens does not impact our claim on the resource bounds required. Effectively, we are given a n-arity tree and use $\mathcal{O}(n)$ extra nodes to create a binary tree by replacing edges with these intermediate nodes. Denote by r the root nodes, by $v_1, v_2, \ldots v_n$ the leaf nodes, and by $h_1, h_2, \ldots h_n$ the extra intermediary nodes. We will create a right-branching binary tree as follows. The root node has edges to v_1 and h_1, and now h_1 recursively needs to span $v_2, v_3 \ldots v_n$. h_1 then has edges to v_2 and h_2, so on and so forth. The resulting tree is a binary, right-branching tree.

We can implement this procedure with padding tokens as follows. For a padding token associated with an item $[\mathrm{A}, i, j]$, there are $\mathcal{O}(n)$ additional padding tokens. As described in App. B.1, we unpack the bits of the position of the padding token and can also apply a feedforward network to obtain padding tokens that store $[\mathrm{A}, i, j]$ and an index in $[n]$ (denoting the ordering of the additional padding tokens). We assume some ordering on the current n children of $[\mathrm{A}, i, j]$, where each associated padding token also stores an index in $[n]$. The padding token for the node associated with $[\mathrm{A}, i, j]$ attends to its first

child and the first additional padding token. The first additional padding token attends to the second child and the second additional padding token. We can implement this procedure with an attention layer.

Solving reachability queries: Reachability queries over binary trees now reduce to evaluating the Boolean formula associated with the binary tree. Leaf nodes associated with realizable items are assigned 1. A non-leaf node has a path to such a leaf if evaluating the induced Boolean expression where non-leaf compute $\vee$ over their children yields 1. We can therefore invoke Lem. 4.1 to evaluate this Boolean formula.

Recognition step: The EOS token can attend to to the padding token for node associated with $[\mathrm{S}, 1, n]$ and check whether it is realizable, i.e store 1 in its residual stream.

■

Theorem 4.1. *Let* UCFL *be the classes of unambiguous* CFL*s. Then* UCFL $\subseteq \mathtt{mAHAT}^2_3 \subseteq \mathtt{AHAT}^2_4$.

Proof. We define $\mathcal{V}$ as in §4.1. Now assuming linearity, there is an edge from v_1 to v_2 if and only if v_1 takes the form $[\mathrm{A}, i, j]$, v_2 takes the form $[\mathrm{B}, i+1, j]$ such that $\mathrm{A} \to \boldsymbol{w}_i\mathrm{B} \in \mathcal{P}$ (or symmetric case where $\mathrm{A} \to \mathrm{B}\boldsymbol{w}_i \in \mathcal{P}$). We first remark that we now have a *constant* number of outgoing edges for each node. Due to linearity, non-terminal rules are of the form $\mathrm{A} \to \mathtt{w}\mathrm{B}$ or $\mathrm{A} \to \mathrm{B}\mathtt{w}$, and solving an item $[\mathrm{A}, i, j]$ therefore reduces to solving items that aim to derive either $\boldsymbol{w}_{i+1} \dots \boldsymbol{w}_j$ or $\boldsymbol{w}_i \dots \boldsymbol{w}_{j-1}$. There are finitely many items.

Moreover, in contrast to Alg. 3, we may solely consider the dependency graph $\mathrm{DG}(\mathcal{T})$ and invoke a single time Lem. 4.1 on it. The intuition comes from the fact that any production rule used in a derivation always spawns a terminal in the string. If $\mathrm{A} \to \mathtt{w}\mathrm{B}$ is a production rule used in the derivation of a string, then $[\mathtt{w}, i, i] \in \mathcal{T}$ for for some i, and $\mathcal{R}([\mathtt{w}, i, i], [\mathrm{B}, i+1, j], [\mathrm{A}, i, j]) = 1$. Crucially, any production rule applied in the derivation of a string that reduces some item $[\mathrm{A}, i, j]$ to another item $[\mathrm{B}, i+1, j]$ leads to an edge between their associated items in the *initial* dependency graph $\mathrm{DG}(\mathcal{T})$. ■