BENCHMARKING NEURAL NETWORKS ON FORMAL LANGUAGE CLASSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Understanding the computational abilities of neural networks is key to predicting their performance on real-world tasks, and formal language theory offers a mature, rigorous framework for characterizing what kinds of problems they can solve. A large body of theoretical work has sought to pin down which formal language classes they can express, but this work relies on various simplifications (e.g., hard attention) that differ from practice. Empirical studies that use unsimplified architectures therefore provide an important supplement to this work, but they typically evaluate models on small sets of hand-picked languages, leaving the rest of a class drastically underexplored. To remedy this, we evaluate transformers, simple RNNs, and LSTMs on formal languages sampled randomly from a hierarchy of pertinent classes: $\mathscr{R}$-trivial, star-free, regular, and context-free languages. We sample from distributions with full support over each class, so expanding the coverage of a class is simply a matter of scale. We also develop a novel algorithm for efficiently sampling a string of a target length from a probabilistic context-free grammar. Transformers appear to agree with predictions based on unique hard attention: they recognize some non-$\mathscr{R}$-trivial languages but no non-star-free languages. All models learn to approximate context-free languages well but struggle to recognize them perfectly. We have publicly released our code.[1]

1 INTRODUCTION

Transformer-based language models have infiltrated almost every aspect of daily life. They increasingly serve not only as instruments for processing natural language, but as general-purpose solvers of computational problems. However, we still lack a precise understanding of their computational abilities. Formal language theory provides a mature, precise framework for elucidating this, and a growing body of recent work has sought to characterize the abilities of transformers (and other neural network architectures) within this framework (see Strobl et al. (2024) for a survey).

The theory of formal languages and neural networks is of great practical interest. Suppose, for instance, that one uses a transformer language model as a code completion tool for C (without chain-of-thought). In order to merely determine whether the prompt ends inside or outside of a string literal, the language model must determine whether the number of unescaped `"` characters is even or odd. However, this closely resembles the well-known PARITY problem: determining whether a binary string contains an odd number of 1's. Hahn & Rofin (2024) and Kövér et al. (2026) proved that it is nearly impossible for transformers to learn this function, and Yang et al. (2024) and Li & Cotterell (2025) proved that transformers cannot even express it. Thus, the theoretical literature predicts a failure mode in this case.

However, these theoretical results often hinge upon simplifications of the transformer architecture that differ significantly from practice, and different sets of simplifications lead to different results. For example, Yang et al. (2024) assume transformers with rightmost unique hard attention, strict future masking, and no positional encodings and show that they express a set of problems known as the star-free languages. On the other hand, Li & Cotterell (2025) assume transformers with fixed precision, softmax attention, strict future masking, and no positional encodings and show that they express a smaller subset of problems known as the $\mathscr{R}$-trivial languages. To date, no one has been able to characterize the expressivity of unsimplified transformers.

[1] See the anonymous supplementary material.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

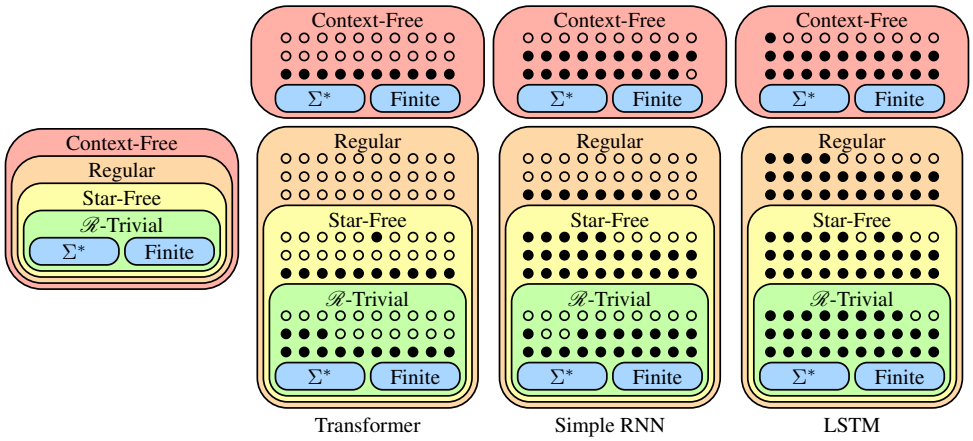

Figure 1: Left: Venn diagram showing the hierarchy of language classes we study, where Σ^* denotes the class $\{\Sigma^* \mid \Sigma \text{ is an alphabet}\}$, and Finite denotes the class of finite languages. Right: Venn diagrams summarizing our results. Circles represent languages, and a filled circle means that the neural network architecture listed beneath learned to recognize the language perfectly. The placement of languages accurately reflects inclusion and non-inclusion in the corresponding classes, and the ordering of languages is consistent across architectures. The transformer can recognize some non-$\mathscr{R}$-trivial languages, but not any non-star-free languages. The RNN and LSTM recognize almost a proper superset of the languages the transformer recognizes.

Empirical studies, therefore, play an invaluable role in understanding the computational abilities of neural networks (Delétang et al., 2023; van der Poel et al., 2024; Butoi et al., 2025), as they allow us to work with unsimplified architectures. Unfortunately, much of this work attempts to characterize the performance of an architecture on an entire, infinite class of problems while in reality only evaluating models on a finite set of hand-picked problems, leaving the class severely underexplored. We remedy this by proposing algorithms that randomly sample problems with full support over the entire class. This way, expanding coverage of a class is simply a matter of scale; we can always sample new languages with non-zero probability. Although some prior work has tried randomly sampling problems (Valvoda et al., 2022; Finlayson et al., 2022; Khalighinejad et al., 2023; Borenstein et al., 2024; Akyürek et al., 2024; Petty et al., 2025), it has not sampled with full support over the class.

We run experiments with transformers, simple RNNs, and LSTMs on a hierarchy of well-known classes: $\mathscr{R}$-trivial, star-free, regular, and context-free languages (see Figure 1). We use the dataset generation and training methodology of Butoi et al. (2025) and propose a novel algorithm for efficiently sampling strings from a desired length range from a context-free language. We show that transformers learn some non-$\mathscr{R}$-trivial languages but no non-star-free languages, in agreement with Yang et al.'s (2024) prediction. We also see that simple RNNs and LSTMs outperform transformers. All architectures learn strong approximations of context-free languages but struggle to learn them perfectly. We also experimentally affirm Hahn & Rofin's (2024) finding that transformers fail to learn high-sensitivity languages. Our methodology provides valuable insight into existing theories of neural network expressivity and an important benchmark for new neural architectures.

2 Background

We first review some essential definitions. An **alphabet** Σ is a finite, non-empty set of elements called **symbols**. A **string** $\boldsymbol{w}$ over Σ is a finite sequence of symbols in Σ. We denote the length of $\boldsymbol{w}$ as $|\boldsymbol{w}|$, the concatenation of strings $\boldsymbol{u}$ and $\boldsymbol{v}$ as $\boldsymbol{uv}$ or $\boldsymbol{u} \circ \boldsymbol{v}$, and the empty string as ε. A **language** (or **formal language**) L over Σ is a set of strings over Σ. We can use any of the familiar set operations on languages, such as union ($\cup$) and complementation ($\overline{L} = \Sigma^* \setminus L$). The concatenation of languages L_1 and L_2, denoted $L_1 \circ L_2$, is $\{\boldsymbol{uv} \mid \boldsymbol{u} \in L_1, \boldsymbol{v} \in L_2\}$. For any language (or alphabet) L, the Kleene closure L^* is the set of strings that can be formed by concatenating zero or more elements in L. A **language class** is a set of formal languages.

Definition 1. *A **deterministic finite automaton (DFA)**[2] is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where (1) Q is a finite set of **states**, (2) Σ is an alphabet, (3) $\delta\colon Q \times \Sigma \to Q \cup \{\emptyset\}$ is the **transition function**, where $\emptyset$ indicates the absence of a transition, (4) $q_0 \in Q$ is the **start state**, and (5) $F \subseteq Q$ is the set of **accept states**. If $\delta(q, a) = r \neq \emptyset$, we say that $\mathcal{A}$ has a transition from q to r that scans a, and we write $q \xrightarrow{a} r \in \delta$.*

A **path** of a DFA $\mathcal{A}$ is a sequence of connecting states and transitions. A path that starts and ends in the same state is a **cycle**. If a path's transitions scan the symbols $w_1, \ldots, w_n$, we say that it **scans** the string $w_1 \cdots w_n$. We say that $\mathcal{A}$ **accepts** the string $\boldsymbol{w}$ if it has a path from q_0 to an accept state that scans $\boldsymbol{w}$ and that it **rejects** $\boldsymbol{w}$ otherwise (see Algorithm 1). A DFA $\mathcal{A}$ **recognizes** the language $\{\boldsymbol{w} \in \Sigma^* \mid \mathcal{A} \text{ accepts } \boldsymbol{w}\}$. A language is **regular** if some DFA recognizes it; such languages comprise the class of **regular languages**. Two DFAs are equivalent if they recognize the same language. A DFA is **minimal** if it has the minimum number of states among its equivalent DFAs. Every regular language has a unique minimal DFA (modulo the renaming of states) (Hopcroft et al., 2006). A DFA is **trim** if every state occurs in some accepting path.

A **partially-ordered DFA (PODFA)** is a DFA whose only cycles are self-loops, i.e., the reflexive transitive closure of δ is a partial ordering of the states. PODFAs recognize the class of **$\mathscr{R}$-trivial languages**[3] (Brzozowski & Fich, 1980), a proper subset of the regular languages. Li & Cotterell (2025) proved that finite-precision, future-masked, soft-attention transformers without positional encodings express exactly the $\mathscr{R}$-trivial languages.

Definition 2. *A **regular expression** over alphabet Σ is one of (1) ε; (2) $\emptyset$; (3) a for some $a \in \Sigma$; (4) $(R_1 \circ R_2)$ for some regular expressions R_1, R_2; (5) $(R_1 \cup R_2)$ for some regular expressions R_1, R_2; or (6) $(R)^*$ for some regular expression R. Items (1), (2), (3) represent the languages $\{\varepsilon\}$, $\emptyset$, $\{a\}$, respectively. Items (4)–(6) represent the languages obtained by applying their respective operations.*

Regular expressions describe exactly the regular languages (Kleene, 1956). A **star-free regular expression** is defined as above, except that it may include the complement operation $(\bar{\cdot})$ but *not* the Kleene star $(\cdot^*)$. Star-free regular expressions describe exactly the class of **star-free languages** (Schützenberger, 1965). Star-free languages are a proper superset of $\mathscr{R}$-trivial languages and a proper subset of regular languages. Yang et al. (2024) proved that transformers with rightmost hard attention and without positional encodings express exactly the star-free languages.

Definition 3. *A **context-free grammar (CFG)** is a tuple $\mathcal{G} = (V, \Sigma, R, S)$ where (1) V is a finite set called the **variables**, (2) Σ is a finite set, disjoint from V, called the **terminals**, (3) R is a finite set of **rules** of the form $X \to \boldsymbol{\alpha}$, where $X \in V$ and $\boldsymbol{\alpha} \in (V \cup \Sigma)^*$, and (4) $S \in V$ is the **start variable**.*

A string of terminals and variables is called a **sentential form**. If X is the leftmost variable in a sentential form and $X \to \boldsymbol{\alpha} \in R$, we can replace that X with $\boldsymbol{\alpha}$ to produce a new sentential form.[4] If a sentential form $\boldsymbol{\beta}$ can yield $\boldsymbol{\beta}'$ through zero or more leftmost replacements, we write $\boldsymbol{\beta} \overset{*}{\Rightarrow} \boldsymbol{\beta}'$. We call such a sequence of steps a **derivation**. We denote the set of derivations in CFG $\mathcal{G}$ from $\boldsymbol{\beta}$ to $\boldsymbol{\beta}'$ as $\Omega_{\boldsymbol{\beta},\boldsymbol{\beta}'}(\mathcal{G})$ and the set of derivations from $\boldsymbol{\beta}$ to any string of terminals as $\Omega_{\boldsymbol{\beta}}(\mathcal{G})$. A CFG $\mathcal{G}$ **generates** the string $\boldsymbol{w}$ if $S \overset{*}{\Rightarrow} \boldsymbol{w}$ (possibly by more than one derivation), and it **generates** the language $\{\boldsymbol{w} \in \Sigma^* \mid \mathcal{G} \text{ generates } \boldsymbol{w}\}$. A language is **context-free** if some CFG generates it, and such languages form the class of **context-free languages**. Context-free languages are a proper superset of regular languages. A CFG is **trim** if all of its variables and rules occur in at least one derivation from S to a string of terminals.

Definition 4. *A CFG $\mathcal{G} = (V, \Sigma, R, S)$ is in **Chomsky normal form (CNF)** if R only contains rules of the form (1) $S \to \varepsilon$; (2) $X \to YZ$, where $Y, Z \in V \setminus \{S\}$, which we call **binary rules**; or (3) $X \to a$, where $a \in \Sigma$, which we call **terminal rules**.*

Every context-free language has a CFG in CNF that generates it (Hopcroft et al., 2006). CNF often admits more efficient algorithms than general CFGs. Next, we show how to assign weights drawn from a semiring to the rules of a CFG.

[2] What we call a DFA is sometimes called a *partial* DFA, as the transition function may be incomplete.

[3] The name $\mathscr{R}$-trivial comes from an algebraic characterization of PODFAs (Brzozowski & Fich, 1980). The class also has an equivalent logical characterization called **LTL[P]** (Li & Cotterell, 2025).

[4] We only allow leftmost replacements in order to eliminate spurious ambiguity due to differences in replacement order.

Definition 5. *A **monoid** is a tuple $(\mathbb{K}, \odot, \boldsymbol{I})$, where $\mathbb{K}$ is a set, $\odot$ is an associative binary operation, and $\boldsymbol{I} \in \mathbb{K}$, called the **identity** element, satisfies $\boldsymbol{I} \odot a = a \odot \boldsymbol{I} = a$ for all $a \in \mathbb{K}$. If $a \odot b = b \odot a$ for all $a, b \in \mathbb{K}$, we say that the monoid is **commutative**.*

Definition 6. *A **semiring** is a tuple $(\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ where $(\mathbb{K}, \oplus, \mathbf{0})$ is a commutative monoid and $(\mathbb{K}, \otimes, \mathbf{1})$ is a monoid. Additionally, $\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$; furthermore, $\mathbf{0} \otimes a = a \otimes \mathbf{0} = \mathbf{0}$.*

An example of a semiring is the real semiring $(\mathbb{R}, +, \times, 0, 1)$, which uses the familiar addition and multiplication operations.

Definition 7. *A **weighted context-free grammar (WCFG)** over a semiring $(\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a tuple $\mathcal{G} = (V, \Sigma, R, S, \psi)$ where (V, Σ, R, S) is a CFG and $\psi \colon R \to \mathbb{K}$ is a **rule weighting function**. If $\psi(X \to \boldsymbol{\alpha}) = w$, we write $X \xrightarrow{w} \boldsymbol{\alpha}$.*

The **weight** $\psi(\tau)$ of a derivation τ is the $\otimes$-product of the weights of the rules it contains. For a WCFG $\mathcal{G}$, the weight of a string $\boldsymbol{w}$ is $\psi(\boldsymbol{w}) \stackrel{\text{def}}{=} \bigoplus_{\tau \in \Omega_{S,\boldsymbol{w}}(\mathcal{G})} \psi(\tau)$. For each variable X, the **partition function** $Z_X(\mathcal{G}) \stackrel{\text{def}}{=} \bigoplus_{\tau \in \Omega_X(\mathcal{G})} \psi(\tau)$.

Definition 8. *A WCFG $\mathcal{G} = (V, \Sigma, R, S, \psi)$ over the real semiring is a **probabilistic CFG (PCFG)** if all rules have non-negative weights and, for all $X \in V$, $\sum_{X \xrightarrow{p} \boldsymbol{\alpha}} p = 1$.*

A PCFG $\mathcal{G}$ is **tight** if $Z_S(\mathcal{G}) = 1$. We denote the probability distribution over Σ^* that a tight PCFG $\mathcal{G}$ induces as $p_{\mathcal{G}}$. Trim PCFGs are always tight (Booth & Thompson, 1973).

Let $\mathcal{U}(A)$ denote the uniform distribution over set A, and let $\mathrm{NB}(r, p)$ denote a negative binomial distribution with r successes and a success rate of p. At several points in this paper, we wish to sample an integer with minimum value n and mean value $\mu > n$, with full support over $\{n, n+1, \ldots\}$; we do this by sampling from $\mathrm{NB}(\mu - n, 0.5) + n$. See App. A for more details and formal definitions.

3 Method

The goal of our experiments is to investigate how well sequential neural network architectures align with certain formal language classes in terms of the languages they can learn to recognize. We do this for the four classes introduced above: $\mathscr{R}$-trivial languages, star-free languages, regular languages, and context-free languages. As Figure 1 shows, these classes form a hierarchy. In order to test a wide variety of languages within each class, we propose four algorithms that randomly generate languages from each of these classes. We design our algorithms according to two important desiderata: (1) they avoid sampling languages from smaller classes, and (2) they generate every remaining language in the class with non-zero probability, i.e., the probability distribution over languages has full support over the language class, minus any smaller class that has been excluded. This way, accurately measuring how well a neural network architecture aligns with a language class is only a matter of scale; one can simply sample more languages to make the experiments more comprehensive. Our approach improves upon similar empirical studies, which either train models on *hand-picked* languages (Bhattamishra et al., 2020; Delétang et al., 2023; Butoi et al., 2025; Li & Cotterell, 2025) or on sampled ones but without full support over the target class (Valvoda et al., 2022; Borenstein et al., 2024; Khalighinejad et al., 2023; Petty et al., 2025; Gupta et al., 2025). We present our language sampling algorithms in §§ 3.1 to 3.4.

For each language, we use Butoi et al.'s (2025) method to randomly generate balanced datasets of strings paired with positive or negative labels, indicating that they are members or non-members of the language, respectively. This method generates strings restricted to a desired length range, and it generates negative strings both by randomly sampling symbols uniformly and by applying edits to positive strings, using a membership testing algorithm to reject strings that are in the language. Therefore, all datasets contain a challenging variety of positive and negative strings, including negative strings that closely resemble positive strings. Following Butoi et al. (2025), we generate a training set of 10k strings[5] with length range $[0, 40]$, a validation set of 1k strings with length range $[0, 80]$, and a test set of 5,010 strings with length range $[0, 500]$. This way, we test the ability of

[5] As we show in App. E, increasing the training data to 100k strings has little effect on accuracy, indicating that 10k is a reasonable choice.

models to generalize to strings much longer than those seen in training, demonstrating that they have learned the underlying algorithm (cf. Delétang et al., 2023; Butoi et al., 2025).

Butoi et al.'s (2025) dataset generation method requires two language-specific algorithms: (1) an algorithm for randomly sampling a positive string within a desired length range, and (2) an algorithm for checking whether an arbitrary string is a member of the language. We represent our randomly sampled $\mathscr{R}$-trivial, star-free, and regular languages as minimal DFAs, in which case we use the same DFA-specific algorithms as Butoi et al. (2025). On the other hand, we represent our randomly sampled context-free languages as trim CFGs in CNF. We develop a novel algorithm for length-constrained string sampling from CFGs (§3.6). For membership testing, we use a tensorized PyTorch (Paszke et al., 2019) implementation of the CKY algorithm (Cocke, 1969; Kasami, 1965; Younger, 1967) that is parallelized across the length of the string; implementing this efficiently proved crucial, as testing the membership of strings in the test set was the main performance bottleneck of our dataset generation pipeline. After generating the datasets, we use them to train neural networks and evaluate their recognition accuracy (§3.7).

3.1 Sampling $\mathscr{R}$-trivial languages

To sample an $\mathscr{R}$-trivial language, we randomly build a DFA that is partially-ordered by construction. First, using a negative binomial distribution as described in §2, we randomly sample a number of states $|Q| \geq 1$ with mean μ_Q and an alphabet size $|\Sigma| \geq 1$ with mean μ_Σ, where μ_Q and μ_Σ are parameters of the sampling algorithm. We let $\Sigma = \{a_0, \ldots, a_{|\Sigma|-1}\}$ and $Q = \{q_0, \ldots, q_{|Q|-1}\}$, where q_0 is the start state, and $q_0, \ldots, q_{|Q|-1}$ serves as an ordering of the states. For each $q_i \in Q$ and $a \in \Sigma$, with probability p_δ, we sample a state r from $\mathcal{U}(\{q_i, \ldots, q_{|Q|-1}\})$ and add the transition $q \xrightarrow{a} r$. For each $q \in Q$, with probability p_F, we make q an accept state. In this way, we can sample every possible DFA that conforms to a partial ordering of Q. Then we trim the DFA.

Since our datasets require both positive and negative strings, we reject PODFAs that encode $\emptyset$ or Σ^* using simple checks. We also require the PODFA to recognize an infinite language, which we can verify by checking whether the PODFA contains at least one self-loop. Since Butoi et al.'s (2025) method relies on rejection sampling to generate negative strings, we avoid languages that have small complements so that dataset generation runs reasonably fast. To this end, we define a notion of the **density** $\rho_{[n_{\min}, n_{\max}]}(L)$ of a language L over alphabet Σ and length range $[n_{\min}, n_{\max}]$.

$$\rho_{[n_{\min}, n_{\max}]}(L) = \mathop{\mathbb{E}}_{n \sim \mathcal{U}(\{n_{\min}, \ldots, n_{\max}\})} \mathop{\mathbb{E}}_{\boldsymbol{w} \sim \mathcal{U}(\Sigma^n)} \mathbb{1}[\boldsymbol{w} \in L] \tag{1}$$

We can repeatedly sample n and $\boldsymbol{w}$ and use a membership testing algorithm to obtain an estimate $\hat{\rho}_{[n_{\min}, n_{\max}]}(L)$. Using 50 samples, we reject any PODFA whose language is L where $\hat{\rho}_{[0,500]}(L) > 0.8$. Note that this decreases the probability of high-density languages but does not exclude them outright, since estimating the density is still a matter of chance. We re-sample a PODFA until it satisfies all the requirements. Finally, we minimize the PODFA using Hopcroft's algorithm (Hopcroft, 1971) in order to make subsequent steps as efficient as possible. See App. B.1 for details.

3.2 Sampling star-free languages

To sample a star-free language, we randomly construct a star-free regular expression and convert it to the equivalent minimal DFA. We first sample an alphabet size $|\Sigma| \geq 1$ with mean μ_Σ as in §3.1. We then recursively sample one of six different types of expression: (1) ε; (2) $\emptyset$; (3) a symbol $a \in \Sigma$; (4) $R_1 \circ R_2$ for some expressions R_1, R_2; (5) $R_1 \cup R_2$ for some expressions R_1, R_2; or (6) $\overline{R}$ for some expression R. We associate each expression type t with a weight parameter $w_t > 0$ that controls the probability of sampling it. When we sample an expression of type (3), we return a symbol sampled from $\mathcal{U}(\Sigma)$. For expressions of type (4)–(6), we sample the sub-expressions recursively. In order to prevent expressions from blowing up in size, we cause the probability of expression types that contain sub-expressions to decay with recursion depth. Afterwards, in order to decrease the size of the expression for subsequent steps, we recursively simplify the expression using the following identities: $R \circ \emptyset = \emptyset \circ R = \emptyset$, $R \circ \varepsilon = \varepsilon \circ R = R$, $R \cup \emptyset = \emptyset \cup R = R$, and $\overline{\overline{R}} = R$. Then, we recursively convert the expression to the equivalent minimal DFA, keeping the DFAs minimal at each step to avoid a blowup in size. We ensure that this minimal DFA's language is not $\mathscr{R}$-trivial by checking that it contains at least one cycle, ignoring self-loops (Brzozowski & Fich, 1980). We also

ensure that the language's density is not too high as in §3.1. We re-sample a DFA until it satisfies these requirements. See App. B.2 for details.

3.3 Sampling regular languages

To sample a regular language, we randomly build a DFA. In contrast to prior methods which bound the numbers of states and symbols (Valvoda et al., 2022; Borenstein et al., 2024; Akyürek et al., 2024; Gupta et al., 2025), we sample from a distribution with full support over DFAs of any size. We sample $|Q|$ and $|\Sigma|$ as in §3.1. For each $q \in Q$ and $a \in \Sigma$, with probability p_δ, we sample a state r from $\mathcal{U}(Q)$ and add the transition $q \xrightarrow{a} r$. For each $q \in Q$, with probability p_F, we make q an accept state. Then, we minimize the DFA. Van der Poel et al. (2024) tested a similar approach for sampling DFAs but advised against it, as they found that most DFAs recognized languages belonging to an extremely simple subclass called strictly local languages. We obviate this problem by rejecting DFAs that recognize star-free languages, which are a superset of strictly local languages. We check whether the DFA recognizes a star-free language using a method similar to one used by van der Poel et al. (2024). Although this problem is PSPACE-complete (Cho & Huynh, 1991), we find that it is manageable in practice. Lastly, we check that the density of the language is not too high as in §3.1. We re-sample a DFA until it satisfies all requirements. See App. B.3 for details.

3.4 Sampling context-free languages

To sample a context-free language, we randomly construct a CFG in CNF. Unlike prior work (Clark & Fijalkow, 2021; Khalighinejad et al., 2023; Petty et al., 2025), our sampling method ensures that any CFG in CNF can be drawn with positive probability. We sample a number of variables $|V| \geq 1$ with mean μ_V and an alphabet size $|\Sigma| \geq 1$ with mean μ_Σ. We designate a start variable $S \in V$ and, with probability p_ε, add the rule $S \to \varepsilon$. We control the number of binary rules per left-hand-side variable with a parameter μ_b and the number of terminal rules per left-hand-side variable with a parameter μ_t. For every $X \in V$ and $Y, Z \in V \setminus \{S\}$, with probability $\min(0.9, \mu_\mathrm{b}/(|V|-1)^2)$, we add the rule $X \to YZ$ (if $|V| \leq 1$, we use a probability of 0.9). For every $X \in V, a \in \Sigma$, with probability $\min(0.9, \mu_\mathrm{t}/|\Sigma|)$, we add the rule $X \to a$. Then we trim the CFG.

The resulting CFG might generate a regular language, but checking whether an arbitrary context-free language is regular is undecidable (Greibach, 1968). Therefore, we cannot guarantee that all of our CFGs generate non-regular languages, as Figure 1 reflects. In order to alleviate this issue, with configurable probabilities, we add chains of **self-embedding rules**, i.e., rules that cause a variable X to generate at least one terminal to the left and to the right and at least one X in the middle. For example, the rules $A \to BC, B \to 0, C \to AB$, form such a chain. The presence of self-embedding chains is a necessary (but not sufficient) condition for a CFG to generate a non-regular language (Chomsky, 1959). If the trim CFG has more than one variable, we sample a number of self-embedding rules $n_\mathrm{r} \geq 0$ with mean μ_r. Then, n_r times, we sample a chain length $n_\mathrm{c} \geq 2$ with mean μ_c. Using existing non-start variables and terminals, we add a random self-embedding chain of n_c rules. Finally, we check that the trim CFG generates an infinite language (Bar-Hillel et al., 1961) and that the density of the language is not too high as in §3.1, using CKY as the membership testing algorithm. The density check is particularly important here, as we found that some CFGs have problematically high density. It also guarantees that the CFG does not generate Σ^*. We re-sample a CFG until it satisfies these requirements. See App. B.4 for details.

3.5 Calibrating the language distributions

Each of the sampling algorithms above imposes a probability distribution over an infinite set of languages. Although they give us broader coverage over each class than hand-picked languages ever could, any distribution over an infinite language class must distribute probability unevenly, and the choice of where to concentrate probability mass will affect the expected accuracy of any model trained on languages drawn from that distribution. Therefore, we calibrate the parameters of our four sampling algorithms so that their distributions have similar expected difficulty according to a baseline model.

In this initial calibration step, we randomly sample 30 sets of parameter values for each of the four distributions. We then sample 30 languages using each parameter setting. For some regular languages, the star-free-ness check runs out of memory (we run on CPU nodes with 4 GB of RAM), in which

case we discard its parameter setting. For each language, we generate a training set of 10k examples with lengths in $[0, 40]$ and a test set of 1k examples with lengths in $[0, 500]$, which we use to train and evaluate a baseline model. Our baseline model consists of two 3-gram language models with add-1 smoothing: one trained on positive strings, and one trained on negative strings. The baseline accepts a string if the positive language model assigns a probability greater than or equal to that of the negative one. We calculate the average baseline accuracy across languages for each parameter setting. We choose one parameter setting for each language class; we choose the settings that minimize the difference between the maximum and minimum mean baseline accuracies, breaking ties by preferring lower accuracies. See App. C for details.

3.6 Length-constrained string sampling from context-free grammars

Our context-free language sampling algorithm in §3.4 produces a trim CFG $\mathcal{G}$ in CNF. Petty et al. (2025) sample a pool of strings from $\mathcal{G}$, then discard those with lengths outside of the target length range $[n_{\min}, n_{\max}]$. This becomes inefficient when $\mathcal{G}$ places little probability on strings with lengths in $[n_{\min}, n_{\max}]$. Instead, we use an approach analogous to the one Butoi et al. (2025) use for sampling strings from the language of a trim DFA. We first assign uniform probabilities to rules that share the same variable on the left-hand side, producing a tight PCFG $\mathcal{G}'$ in CNF. Let $N_{\mathcal{G}'} \stackrel{\text{def}}{=} \{n \in \{n_{\min}, \ldots, n_{\max}\} \mid \exists \boldsymbol{w} \in \Sigma^n : p_{\mathcal{G}'}(\boldsymbol{w}) > 0\}$, and let W be a $p_{\mathcal{G}'}$-distributed string-valued random variable. To sample a string from $L(\mathcal{G})$, we first sample a length n from $\mathcal{U}(N_{\mathcal{G}'})$ and then sample a string from the conditional distribution $p_{\mathcal{G}'}(\mathrm{W} \mid |\mathrm{W}| = n)$. We propose a method of pre-processing $\mathcal{G}'$ so that sampling from $\mathcal{U}(N_{\mathcal{G}'})$ and $p_{\mathcal{G}'}(\mathrm{W} \mid |\mathrm{W}| = n)$ are efficient. Like Butoi et al.'s (2025) method for length-constrained sampling from probabilistic DFAs, our approach combines two elements: (1) a **weight pushing** algorithm for converting a positive-weighted CFG to an equivalent PCFG (Smith & Johnson, 2007) and (2) the **binning semiring** of Snæbjarnarson et al. (2026).

We first review the weight pushing algorithm of Smith & Johnson (2007). Let $\mathcal{G} = (V, \Sigma, R, S, \psi)$ be a WCFG over the nonnegative real semiring. For all $X \in V$, the algorithm computes the **partition function** $Z_X(\mathcal{G}) = \sum_{\tau \in \Omega_X(\mathcal{G})} \psi(\tau)$ (assuming that this quantity converges). In other words, given rules of the form $X \xrightarrow{w} \boldsymbol{\alpha}$, where w is the weight that the rule contributes to a tree, the algorithm computes $Z_X(\mathcal{G})$, the total weight of all trees rooted at X. Now, suppose that for each rule $X \xrightarrow{p} \boldsymbol{\alpha}$ of our PCFG $\mathcal{G}'$, we replace the probability p with a vector $\boldsymbol{v} \in {\mathbb{R}_{\geq 0}}^{n_{\max}+1}$. Let j be the number of terminals in $\boldsymbol{\alpha}$. We set $(\boldsymbol{v})_i = \mathbb{1}[i = j]\, p$ for $0 \leq i \leq n_{\max}$. The vector $\boldsymbol{v}$ therefore bins the original probability of $X \xrightarrow{p} \boldsymbol{\alpha}$ according to the number of terminals that the rule contributes to a tree (of course, all probability is concentrated at j). Snæbjarnarson et al. (2026) proposed a semiring called the binning semiring that combines vectors of binned probabilities. By using a generalization of the weight pushing algorithm that uses the binning semiring instead of the usual real semiring, we can compute, for all $X \in V$, a vector $\boldsymbol{z}_X(\mathcal{G}')$ which contains the total probabilities of all trees rooted at X, binned according to the number of terminals generated. That is, $(\boldsymbol{z}_X(\mathcal{G}'))_n = p_{\mathcal{G}'}(X \stackrel{*}{\Rightarrow} \mathrm{W} \mid |\mathrm{W}| = n)$. A benefit of using the binning semiring and CNF is that the operations can be vectorized and parallelized using PyTorch. In practice, for numerical stability, we represent all probabilities as log-probabilities, which is valid because they form a semiring.

The vectors $\boldsymbol{z}_X(\mathcal{G}')$ are exactly what we need to compute $N_{\mathcal{G}'}$ and sample efficiently from $p_{\mathcal{G}'}(\mathrm{W} \mid |\mathrm{W}| = n)$. First, note that $N_{\mathcal{G}'} = \{n \in \{n_{\min}, \ldots, n_{\max}\} \mid (\boldsymbol{z}_S(\mathcal{G}'))_n > 0\}$. To sample from $p_{\mathcal{G}'}(\mathrm{W} \mid |\mathrm{W}| = n)$, we devise a way to sample from $p_{\mathcal{G}'}(\mathrm{W} \mid X \stackrel{*}{\Rightarrow} \mathrm{W}, |\mathrm{W}| = i)$ for any $X \in V$ and $0 \leq i \leq n_{\max}$, and we sample from $p_{\mathcal{G}'}(\mathrm{W} \mid S \stackrel{*}{\Rightarrow} \mathrm{W}, |\mathrm{W}| = n)$. When $i = 0$, this corresponds to an empty rule $S \to \varepsilon$, so we return ε. When $i = 1$, this corresponds to a terminal rule, so we sample a, where $X \xrightarrow{\boldsymbol{v}} a$, proportionally to $(\boldsymbol{v})_1$ (since the probabilities of $\mathcal{G}'$ are uniform, this will be a uniform distribution). When $i \geq 2$, we must sample a binary rule, so we sample a triple (Y, Z, j), where $X \xrightarrow{\boldsymbol{v}} YZ \in R$, $j \in \{1, \ldots, i-1\}$, and j indicates that we will recursively sample strings of length j and $i - j$ for Y and Z, respectively. We sample proportionally to $(\boldsymbol{v})_0\, (\boldsymbol{z}_Y(\mathcal{G}'))_j\, (\boldsymbol{z}_Z(\mathcal{G}'))_{i-j}$. We precompute these distributions once for each grammar in cumulative sum form using GPU-accelerated tensor operations in PyTorch. This gives us a simple top-down random sampling algorithm that runs in $O(n)$ time. See App. D for details.

Table 1: Test accuracy of each architecture on each language class. Results are aggregated over 30 languages per class. For each language, we train 10 models with random hyperparameters and report results for the model with the best test accuracy. We show the best scores in each column in **bold**.

(a) Mean accuracy across languages.

Architecture	$\mathscr{R}$-Trivial	Star-Free	Regular	Context-Free
Transformer	$0.998_{\pm.00}$	$0.918_{\pm.10}$	$0.775_{\pm.08}$	$0.998_{\pm.01}$
RNN	$0.999_{\pm.00}$	$0.996_{\pm.02}$	$0.872_{\pm.13}$	$0.999_{\pm.00}$
LSTM	$\mathbf{1.000}_{\pm.00}$	$\mathbf{1.000}_{\pm.00}$	$\mathbf{0.991}_{\pm.04}$	$\mathbf{0.999}_{\pm.00}$
3-Gram Classifier	$0.828_{\pm.02}$	$0.928_{\pm.12}$	$0.792_{\pm.03}$	$0.849_{\pm.03}$

(b) Proportions of languages learned perfectly. See also Figure 1.

Architecture	$\mathscr{R}$-Trivial	Star-Free	Regular	Context-Free
Transformer	0.43	0.37	0.00	0.33
RNN	0.60	0.83	0.27	0.63
LSTM	**0.93**	**0.90**	**0.80**	**0.70**
3-Gram Classifier	0.00	0.17	0.00	0.00

3.7 Training neural networks

Our neural network training and evaluation setup closely follows the expressivity experiments of Butoi et al. (2025), whose goal is to discover whether a neural network architecture admits any parameter setting that recognizes a given language. We test three neural network architectures: (1) a causally-masked **transformer** encoder (Vaswani et al., 2017) with pre-norm (Wang et al., 2019; Nguyen & Salazar, 2019) and sinusoidal positional encodings, (2) a multi-layer **simple RNN** (Elman, 1990) with a $\tanh$ activation function and learned initial hidden states, and (3) a multi-layer **LSTM** (Gers & Schmidhuber, 2001) with decoupled input and forget gates and learned initial hidden states. Each network reads the symbols of an input string and uses a binary classification head attached to the last layer of the last position to classify whether it belongs to a language; the networks do not use additional chain-of-thought steps before making a prediction. The transformer always receives a reserved BOS symbol as its first input. We train the networks using a binary cross-entropy loss function. We do not use the auxiliary loss terms of Butoi et al. (2025), as they showed that those loss terms do not consistently improve accuracy. The model's loss on the validation set serves as the criterion for learning rate decay and early stopping. Since the validation set contains strings that are longer than those in the training data, the model's ability to generalize to longer strings guides the training process.

We train 10 models per architecture on each language using randomly sampled hyperparameters and report results for the model with highest accuracy on the test set. We randomly sample the maximum number of tokens per batch and the initial learning rate. Unlike Butoi et al. (2025), we also randomly search different model shapes and sizes. For transformers, we randomly sample the number of layers from $\mathcal{U}(\{1, \ldots, 16\})$ and the number of attention heads per layer from $\mathcal{U}(\{1, \ldots, 16\})$. For simple RNNs and LSTMs, we randomly sample the number of layers from $\mathcal{U}(\{1, \ldots, 8\})$. We sample a target number of parameters n_{params} from $\mathcal{U}(\{10\text{k}, \ldots, 100\text{k}\})$. We adjust the size of the model's hidden representations to ensure that the total number of parameters in the model is as close as possible to n_{params}, using SymPy (Meurer et al., 2017) to work out the algebra automatically. For transformers, this means adjusting the model width d_{model}; for simple RNNs and LSTMs, we adjust the hidden state size. In transformers, we ensure that the size of the feedforward layers is $4d_{\text{model}}$. All other details about the architectures and training procedure we use are identical to those of Butoi et al. (2025).

4 Results

We show results aggregated over languages for each class and architecture in Table 1. We report both the mean accuracy and the proportion of languages that each architecture learns perfectly. We also report results for the baseline 3-gram classifier from §3.5 as an indicator of general difficulty. Cases in which a model recognizes a language perfectly are particularly informative, as they demonstrate the existence of an exact solution, not just an approximation. Formal language theory and neural

network expressivity literature typically deal only with exact solutions. Figure 1 shows which languages each architecture learns perfectly.

Li & Cotterell (2025) and Yang et al. (2024) proved two different expressivity results for transformers based on different simplifying assumptions. Li & Cotterell (2025) proved that finite-precision, future-masked, soft-attention transformers without positional encodings recognize exactly the $\mathscr{R}$-trivial languages, whereas Yang et al. (2024) proved that masked unique hard-attention transformers without positional encodings express exactly the star-free languages. Whichever expressivity bound proves more accurate, we expect transformers to learn only a subset of the class in our experiments, as the training algorithm is not guaranteed to find an optimum. Our results align with the result of Yang et al. (2024): transformers learn some non-$\mathscr{R}$-trivial star-free languages but no non-star-free regular languages. However, we note that Li & Cotterell's (2025) result likely applies only to extremely long strings; our test set of strings up to length 500 may simply not be long enough to see this effect. The RNN and LSTM, which have been shown to be at least as expressive as regular languages (Merrill, 2019; Merrill et al., 2020; Svete & Cotterell, 2023), learn regular languages throughout the hierarchy. On every language class, in terms of both accuracy and number of languages learned perfectly, the LSTM is the best-performing architecture, whereas the transformer is the worst. In Figure 1, we see that the RNN and LSTM recognize almost a superset of what the transformer does. All architectures attain very high accuracy on context-free languages, but they perfectly solve only a subset of them.

We next test the finding of Hahn & Rofin (2024), who proved that transformers struggle to learn Boolean functions with high **sensitivity**. For a Boolean function $f\colon \{-1,1\}^n \to \mathbb{R}$, the sensitivity of f measures how much the output of f changes when a random bit in the input is flipped. We use a generalized sensitivity metric that applies to languages over alphabets with more than two symbols and strings of variable length. For any string $\boldsymbol{w}$, position i, and symbol a, let $\boldsymbol{w}_{i \to a}$ denote the string $\boldsymbol{w}$ with the symbol at position i replaced with a. Let L be a language over alphabet Σ with $|\Sigma| \geq 2$. (None of the languages we sampled have unary alphabets; languages with unary alphabets tend to have high density, so our language sampling algorithms reject them with high probability.) The sensitivity of L over length range $[n_{\min}, n_{\max}]$ is

$$s_{[n_{\min},n_{\max}]}(L) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{n\sim\mathcal{U}(\{n_{\min},\ldots,n_{\max}\})} \mathop{\mathbb{E}}_{\boldsymbol{w}\sim\mathcal{U}(\Sigma^n)} \mathop{\mathbb{E}}_{i\sim\mathcal{U}(\{1,\ldots,n\})} \mathop{\mathbb{E}}_{a\sim\mathcal{U}(\Sigma\setminus\{\boldsymbol{w}_i\})} \mathbb{1}[\boldsymbol{w} \in L \oplus \boldsymbol{w}_{i\to a} \in L], \tag{2}$$

where $\oplus$ denotes exclusive or. We can calculate an estimate $\hat{s}_{[n_{\min},n_{\max}]}(L)$ by repeatedly sampling n, $\boldsymbol{w}$, i, and a and using the membership testing algorithms mentioned in §3. For each language L in our experiments, we calculate $\hat{s}_{[0,500]}(L)$ using 1k samples.

To understand how the language class, neural network architecture, and sensitivity influence the accuracy of a model, we analyze our results using a linear mixed effects model implemented with `statsmodels` (Seabold & Perktold, 2010). We use only the best of the 10 models for each language and architecture, and we use test accuracy as the response variable. We use a linear mixed effects model to account for the fact that multiple data points correspond to the same language. We treat language as a random effect, and we treat language class, architecture, sensitivity, and baseline accuracy as fixed effects. We model interaction effects between class and architecture and between architecture and sensitivity. We fit the model using REML and perform Wald tests. The statistically significant ($p < 0.001$) results are as follows. Non-star-free regular languages are generally more difficult than $\mathscr{R}$-trivial languages. Transformers perform poorly on non-star-free regular languages, whereas LSTMs perform strongly (relative to RNNs and $\mathscr{R}$-trivial languages). Interestingly, we see that the transformer's accuracy does correlate negatively with the sensitivity of a language (relative to the RNN), affirming the result of Hahn & Rofin (2024).

5 Conclusion

We proposed a methodology for benchmarking neural network architectures on formal languages sampled with full support over entire language classes, offering much more comprehensive results than benchmarks comprised of hand-picked languages. For transformers, our experiments affirm the expressivity result of Yang et al. (2024) and the learnability result of Hahn & Rofin (2024), but not the expressivity result of Li & Cotterell (2025). LSTMs almost entirely outperform transformers on formal languages despite transformers' empirical success on natural language. Our methodology provides an important means of testing future neural network architectures.

AI USE STATEMENT

We produced the vast majority of this paper's text, code, experiments, and underlying ideas without AI assistance. We occasionally used generative AI tools for help with writing (e.g., some parts of the appendix), code generation, research, and guidance on research methodology (namely, the star-free-ness check mentioned in §3.3 and the linear mixed effects model used in §4). We have carefully checked all AI-generated content.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we have publicly released all code we used to run our experiments and generate the figures and tables in this paper. In order to make it easy to replicate our software development, we have provided a Docker container definition which we used when developing our code and running our experiments. In our README, we have documented each of the shell commands we used for each experiment, figure, and table. We use fixed random seeds for all random sampling in our experiments, so our results are reproducible deterministically. (The only exception is the dropout masks used when training neural networks, which can result in some minor variation in results.) We have thoroughly documented our algorithms and experimental settings in §3 and the appendix.

REFERENCES

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Architectures and algorithms. In *Forty-first International Conference on Machine Learning*, volume 235, pp. 787–812, 2024. URL https://proceedings.mlr.press/v235/akyurek24a.html.

Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *Sprachtypologie und Universalienforschung*, 14:143–172, 1961. URL https://www.proquest.com/openview/fb41296047fb7453dcb1de182b4aa0b6/1?pq-origsite=gscholar&cbl=1817266.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 7096–7116, 2020. URL https://aclanthology.org/2020.emnlp-main.576/.

Taylor L. Booth and Richard A. Thompson. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, C-22:442–450, 1973. URL https://api.semanticscholar.org/CorpusID:41410699.

Nadav Borenstein, Anej Svete, Robin Chan, Josef Valvoda, Franz Nowak, Isabelle Augenstein, Eleanor Chodroff, and Ryan Cotterell. What languages are easy to language-model? A perspective from learning probabilistic regular languages. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15115–15134, 2024. URL https://aclanthology.org/2024.acl-long.807.

J. A. Brzozowski and Faith E. Fich. Languages of R-trivial monoids. *Journal of Computer and System Sciences*, 20(1):32–49, 1980. URL https://www.sciencedirect.com/science/article/pii/0022000080900033.

Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, 12:529–561, 1962. URL https://api.semanticscholar.org/CorpusID:118363215.

Alexandra Butoi, Ghazal Khalighinejad, Anej Svete, Josef Valvoda, Ryan Cotterell, and Brian DuSell. Training neural networks as recognizers of formal languages. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=aWLQTbfFgV.

Sang Cho and Dung T. Huynh. Finite-automaton aperiodicity is PSPACE-complete. *Theoretical Computer Science*, 88(1):99–116, 1991. URL https://www.sciencedirect.com/science/article/pii/030439759190075D.

Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959. URL https://www.sciencedirect.com/science/article/pii/S0019995859903626.

Alexander Clark. Testing distributional properties of context-free grammars. In *Proceedings of The 13th International Conference on Grammatical Inference*, volume 57 of *Proceedings of Machine Learning Research*, pp. 42–53, 2017. URL https://proceedings.mlr.press/v57/clark16.html.

Alexander Clark and Nathanaël Fijalkow. Consistent unsupervised estimators for anchored PCFGs. In *Proceedings of the Society for Computation in Linguistics 2021*, pp. 471–473, 2021. URL https://aclanthology.org/2021.scil-1.60/.

John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, 1969. URL https://dl.acm.org/doi/10.5555/1097042.

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. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WbxHAzkeQcn.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. URL https://www.sciencedirect.com/science/article/pii/036402139090002E.

Matthew Finlayson, Kyle Richardson, Ashish Sabharwal, and Peter Clark. What makes instruction learning hard? An investigation and a new challenge in a synthetic environment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 414–426, 2022. URL https://aclanthology.org/2022.emnlp-main.27/.

Felix A. Gers and Jürgen Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001. URL https://ieeexplore.ieee.org/document/963769.

Sheila Greibach. A note on undecidable properties of formal languages. *Mathematical Systems Theory*, 2:1–6, 1968. URL https://link.springer.com/article/10.1007/BF01691341.

Kavi Gupta, Kate Sanders, and Armando Solar-Lezama. Randomly sampled language reasoning problems reveal limits of LLMs. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2025. URL https://openreview.net/forum?id=OC0kqmA1sH.

Michael Hahn and Mark Rofin. Why are sensitive functions hard for transformers? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14973–15008, 2024. URL https://aclanthology.org/2024.acl-long.800/.

John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pp. 189–196. Academic Press, New York, 1971. URL https://www.cs.cmu.edu/~cdm/resources/Hopcroft1971.pdf.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., 3rd edition, 2006. URL https://dl.acm.org/doi/book/10.5555/1196416.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory, 1965. URL https://www.semanticscholar.org/paper/An-Efficient-Recognition-and-Syntax-Analysis-for-Kasami/af66165c454a0e94afbab36271fe3deaae0b421a.

Ghazal Khalighinejad, Ollie Liu, and Sam Wiseman. Approximating CKY with transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14016–14030, 2023. URL https://aclanthology.org/2023.findings-emnlp.934/.

Stephen C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, number 34 in Annals of Mathematics Studies, pp. 3–41. Princeton University Press, 1956. URL https://www.degruyterbrill.com/document/doi/10.1515/9781400882618-002/html?srsltid=AU7gw4WAkHtMMUGoaxZbWfMCGBX6lUSSqkDS69K-6fIrBeJfnJ-snA17.

Blanka Kövér, Alexandra Butoi, Anej Svete, Michael Hahn, and Ryan Cotterell. Understanding the parameter space geometry of transformers encoding Boolean functions. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=rUZVcnQjYW.

Jiaoda Li and Ryan Cotterell. Characterizing the expressivity of fixed-precision transformer language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=29LwAgLFpj.

William Merrill. Sequential neural networks as automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pp. 1–13. Association for Computational Linguistics, 2019. doi: 10.18653/v1/W19-3901. URL https://aclanthology.org/W19-3901/.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. A formal hierarchy of RNN architectures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 443–459. Association for Computational Linguistics, 2020. URL https://aclanthology.org/2020.acl-main.43/.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3, 2017. doi: 10.7717/peerj-cs.103.

Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. In *Proceedings of the 16th International Conference on Spoken Language Translation*. Association for Computational Linguistics, 2019. URL https://aclanthology.org/2019.iwslt-1.17.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Jackson Petty, Michael Y. Hu, Wentao Wang, Shauli Ravfogel, William Merrill, and Tal Linzen. RELIC: Evaluating compositional instruction following via language recognition. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling*, 2025. URL https://openreview.net/forum?id=lhf1MlcbKx.

M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965. URL https://www.sciencedirect.com/science/article/pii/S0019995865901087.

Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with Python. *9th Python in Science Conference*, 2010. doi: 10.25080/Majora-92bf1922-011.

Noah A. Smith and Mark Johnson. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics*, 33(4):477–491, 12 2007. URL https://direct.mit.edu/coli/article/33/4/477/1961/Weighted-and-Probabilistic-Context-Free-Grammars.

Vésteinn Snæbjarnarson, Anej Svete, Josef Valvoda, Reda Boumasmoud, Brian DuSell, and Ryan Cotterell. Causally evaluating the learnability of formal language tasks. In *Forty-third International Conference on Machine Learning*, Seoul, South Korea, July 2026. URL https://arxiv.org/abs/2606.09822.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? A survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024. URL https://aclanthology.org/2024.tacl-1.30/.

Anej Svete and Ryan Cotterell. Recurrent neural language models as probabilistic finite-state automata. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8069–8086. Association for Computational Linguistics, 2023. URL https://aclanthology.org/2023.emnlp-main.502/.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11 (6):419–422, June 1968. doi: 10.1145/363347.363387.

Josef Valvoda, Naomi Saphra, Jonathan Rawski, Adina Williams, and Ryan Cotterell. Benchmarking compositionality with formal languages. In *Proceedings of the 29th International Conference on Computational Linguistics*, pp. 6007–6018, 2022. URL https://aclanthology.org/2022.coling-1.525/.

Sam van der Poel, Dakotah Lambert, Kalina Kostyszyn, Tiantian Gao, Rahul Verma, Derek Andersen, Joanne Chau, Emily Peterson, Cody St. Clair, Paul Fodor, Chihiro Shibata, and Jeffrey Heinz. MLRegTest: A benchmark for the machine learning of regular languages. *Journal of Machine Learning Research*, 25(283):1–45, 2024. URL https://jmlr.org/papers/v25/23-0518.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1810–1822. Association for Computational Linguistics, July 2019. doi: 10.18653/v1/P19-1176. URL https://aclanthology.org/P19-1176.

Andy Yang, David Chiang, and Dana Angluin. Masked hard-attention transformers recognize exactly the star-free languages. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=FBMsBdH0yz.

Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967. URL https://www.sciencedirect.com/science/article/pii/S001999586780007X.

A Further background

Here, we provide additional details and definitions to supplement §2.

A.1 Finite automata

A DFA (Def. 1) is **complete** if δ never outputs $\emptyset$.

Definition 9. *A **non-deterministic finite automaton (NFA)** is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where (1) Q is a finite set of **states**, (2) Σ is an alphabet, (3) $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a finite set of **transitions**, (4) $I \subseteq Q$ is the set of **start states**, and (5) $F \subseteq Q$ is the set of **accept states**. If $(q, a, r) \in \delta$, we say that $\mathcal{A}$ has a transition from q to r that scans a (which can be ε), and we write $q \xrightarrow{a} r \in \delta$.*

Any NFA can be *determinized*, i.e., converted into an equivalent deterministic automaton (Hopcroft et al., 2006).

Definition 10. *A **path** π of $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is a sequence of connecting states and transitions,*

$$\pi = r_0 \xrightarrow{w_1} r_1 \xrightarrow{w_2} r_2 \cdots r_{m-1} \xrightarrow{w_n} r_m, \tag{3}$$

*where $r_i \xrightarrow{w_{i+1}} r_{i+1} \in \delta$ for all $i = 0, \ldots, m-1$. If π's transitions scan the sequence of symbols $w_1, \ldots, w_n$, we say that $\mathcal{A}$ **scans** the string $\boldsymbol{w} = w_1 \cdots w_n$.*

A DFA accepts a string $\boldsymbol{w}$ if it has a path from the initial state q_0 to a final state $r \in F$ that scans $\boldsymbol{w}$. Importantly, a DFA has at most one path scanning $\boldsymbol{w}$ for every string $\boldsymbol{w} \in \Sigma^*$. This allows a linear-time algorithm for checking whether a string is accepted by a DFA (see Algorithm 1).

Definition 11. *Let $\delta^* \colon Q \times \Sigma^* \to Q \cup \{\emptyset\}$ be the **transitive closure** of δ, defined as*

$$\delta^*(q, a) = \delta(q, a), \text{ for } a \in \Sigma \tag{4a}$$

$$\delta^*(q, w_1 \cdots w_n) = \delta(\delta^*(q, w_1 \cdots w_{n-1}), w_n) \tag{4b}$$

with $\delta^*(q, \varepsilon) = q$ and $\delta^*(\emptyset, \boldsymbol{w}) = \emptyset$ for any $\boldsymbol{w} \in \Sigma^*$.

Definition 12. *A **partially-ordered DFA (PODFA)** is a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ such that Q, Σ, q_0, and F are defined as in Def. 1, and for all $q \neq r \in Q$, if $\delta^*(q, \boldsymbol{w}) = r$ for some $\boldsymbol{w} \in \Sigma^*$, then there is no $\boldsymbol{w}' \in \Sigma^*$ with $\delta^*(r, \boldsymbol{w}') = q$.*

Algorithm 1 Check whether a DFA $\mathcal{A}$ accepts a string $\boldsymbol{w} \in \Sigma^*$. Time complexity: $O(n)$.

```
def SimulateDFA(A = (Q, Σ, δ, q_0, F), w = w_1 ··· w_n):
    q ← q_0
    for a = w_1, ..., w_n :
        q ← δ(q, a)
        if q = ∅ :
            return ⊥
    return q ∈ F
```

Definition 13. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We call a state $q \in Q$ **accessible** if $\mathcal{A}$ has a path from q_0 to q and **co-accessible** if $\mathcal{A}$ has a path from q to r, where $r \in F$.*

Definition 14. *We call a state **useless** if it is either non-accessible or non-co-accessible. We call a transition **useless** if it contains any useless states. We call $\mathcal{A}$ **trim** if none of its states and transitions are useless.*

Importantly, the correctness of the length-constrained sampling method for regular languages relies on the fact that the automaton is trim. We show how to remove useless states and transitions from an NFA in Algorithm 2.

Algorithm 2 Trim useless states and transitions from an NFA $\mathcal{A}$. The algorithm Reachable$(G = (V, E), S)$ can be any graph search algorithm that returns the set of all vertices in V that are reachable from any vertex in S in $O(|V| + |E|)$ time.

```
def TrimNFA(A = (Q, Σ, δ, q_0, F)):
    G ← (Q, {(q, r): q -a-> r ∈ δ})
    Q' ← Reachable(G, {q_0})
    G ← (Q', {(r, q): q -a-> r ∈ δ | q, r ∈ Q'})
    F' ← F ∩ Q'
    Q' ← Reachable(G, F')
    δ' ← {q -a-> r ∈ δ | q, r ∈ Q'}
    add q_0 to Q'
    return (Q', Σ, δ', q_0, F')
```

A.2 Context-free grammars

Definition 15. *A variable $X \in V$ is called **reachable** if there exists a derivation $S \overset{*}{\Rightarrow} \boldsymbol{\alpha}_1 X \boldsymbol{\alpha}_2$, where $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in (V \cup \Sigma)^*$. A variable $X \in V$ is called **generating** if there exists a derivation $X \overset{*}{\Rightarrow} \boldsymbol{w}$ where $\boldsymbol{w} \in \Sigma^*$. A variable is called **useful** if it is both reachable and generating; otherwise, it is called **useless**. A rule is called **useful** if all its variables are useful; otherwise it is called **useless**.*

Definition 16. *A CFG is called **trim** if all its variables and rules are useful.*

Algorithm 3 shows how to remove useless variables and rules from a CFG in Chomsky normal form.

Algorithm 3 Trim useless variables and rules from a CFG $\mathcal{G}$ in Chomsky normal form. The algorithm REACHABLE$(G = (V, E), X)$ can be any graph search algorithm that returns the set of all vertices in V that are reachable from any vertex in X in $O(|V| + |E|)$ time. Lines 2–6 are equivalent to running weight pushing in the Boolean semiring. Time complexity: $O(|V|^2 + |V||R| + |R|)$.

```
1. def TRIMCFG(G = (V, Σ, R, S)):
2.   V' ← NONEMPTYVARIABLES(G)
3.   E ← ∅
4.   for X → Y Z ∈ R :
5.     if Y ∈ V' and Z ∈ V' :
6.       add (X, Y) and (X, Z) to E
7.   V' ← REACHABLE((V' ∪ {S}, E), {S})
8.   R' ← {S → ε ∈ R} ∪ {X → Y Z ∈ R | X, Y, Z ∈ V'} ∪ {X → a ∈ R | X ∈ V'}
9.   return (V', Σ, R', S)
```

Algorithm 4 Given a CFG $\mathcal{G}$ in Chomsky normal form, determine the set of variables in $V \setminus \{S\}$ that generate at least one string. Time complexity: $O(|V|^2 + |V||R| + |R|)$.

```
1. def NONEMPTYVARIABLES(G = (V, Σ, R, S)):
2.   R'_bin ← {X → Y Z ∈ R | X ≠ S}
3.   T_0 ← {X : X → a ∈ R | X ≠ S}
4.   T ← T_0
5.   loop
6.     T' ← T_0
7.     for X → Y Z ∈ R'_bin :
8.       if Y ∈ T and Z ∈ T :
9.         add X to T'
10.    if |T'| = |T| :
11.      return T
12.    T ← T'
```

Definition 17. *Let $\mathcal{G} = (V, \Sigma, R, S, \psi)$ be a WCFG over the semiring $(\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. A **partial derivation** τ is a sequence of sentential forms and rules,*

$$\tau = \boldsymbol{\alpha}_0 \overset{r_1}{\Longrightarrow} \boldsymbol{\alpha}_1 \overset{r_2}{\Longrightarrow} \boldsymbol{\alpha}_2 \cdots \boldsymbol{\alpha}_{m-1} \overset{r_m}{\Longrightarrow} \boldsymbol{\alpha}_m, \tag{5}$$

*such that $\boldsymbol{\alpha}_0 = X \in V$, $\boldsymbol{\alpha}_m = \boldsymbol{w} \in \Sigma^*$, and for all $i = 1, \ldots, m$, $r_i \in \psi$ and $\boldsymbol{\alpha}_{i-1} \overset{r_i}{\Longrightarrow} \boldsymbol{\alpha}_i$. If $\boldsymbol{\alpha}_0 = S$, τ is called a **derivation**. The **weight** of τ is defined as*

$$\psi(\tau) \overset{\text{def}}{=} \bigotimes_{i=1}^{m} \psi(r_i). \tag{6}$$

Definition 18. *Let $\Omega_X(\mathcal{G})$ be the set of derivations of $\mathcal{G}$ rooted at X. The **partition function** $Z_X(\mathcal{G})$ is the total weight of all partial derivations of $\mathcal{G}$ rooted X,*

$$Z_X(\mathcal{G}) = \bigoplus_{\tau \in \Omega_X(\mathcal{G})} \psi(\tau). \tag{7}$$

*The **weight** of $\mathcal{G}$ is $Z_S(\mathcal{G})$, i.e., the total weight of its derivations.*

B Details of language sampling

B.1 Details of sampling $\mathscr{R}$-trivial languages

Here, we provide more detail about the $\mathscr{R}$-trivial language sampling algorithm described in §3.1. We give pseudocode in Algorithm 5.

Algorithm 5 Algorithm for randomly sampling a PODFA. Here, $\mu_Q > 1$ is the mean number of states, $\mu_\Sigma > 1$ is the mean alphabet size, $p_\delta \in [0, 1]$ is the probability of adding a transition for a given $(q, a) \in Q \times \Sigma$, and $p_F \in [0, 1]$ is the probability of making a state an accept state. The output is a minimal PODFA $\mathcal{A}$. MINIMIZEDFA is Hopcroft's DFA minimization algorithm. A non-trivial language is neither $\emptyset$ nor Σ^*.

def SAMPLEPODFA($\mu_Q, \mu_\Sigma, p_\delta, p_F, \rho_{\max}$):
  repeat
    sample $|Q| \sim \mathrm{NB}(\mu_Q - 1, 0.5) + 1$
    sample $|\Sigma| \sim \mathrm{NB}(\mu_\Sigma - 1, 0.5) + 1$
    let $Q = \{q_0, \ldots, q_{|Q|-1}\}$
    let $\Sigma = \{a_0, \ldots, a_{|\Sigma|-1}\}$
    let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA
    for $q_i = q_0, \ldots, q_{|Q|-1}$:
      for $a \in \Sigma$:
        if $\phi \sim \mathrm{Ber}(p_\delta)$:
          sample $r \sim \mathcal{U}(\{q_i, \ldots, q_{|Q|-1}\})$
          add $q_i \xrightarrow{a} r$ to δ
      if $\phi \sim \mathrm{Ber}(p_F)$:
        add q_i to F
    $\mathcal{A} \leftarrow$ TRIMDFA($\mathcal{A}$)
  until $\mathcal{A}$ accepts a non-trivial infinite language with density at most $\rho_{\max}$
  return MINIMIZEDFA($\mathcal{A}$)

B.2 DETAILS OF SAMPLING STAR-FREE LANGUAGES

Here, we provide more detail about the star-free language sampling algorithm described in §3.2.

First, we provide more details about how we use the weights w_t to sample regular expression types. We cause the probability of expression types that contain sub-expressions to decay with recursion depth d, where the initial depth is $d = 0$. Let k_t be the number of sub-expressions in expression type t, and let $s > 0$ be a sharpness parameter that controls the rate of decay. Then the probability of sampling t is

$$p(t) \propto \frac{w_t + sd}{k_t + 1}. \tag{8}$$

For expressions of type (4)–(6), we sample the sub-expressions recursively with depth $d + 1$.

Next, we provide more details about converting the simplified regular expression to the equivalent minimal DFA. Constructing the minimal DFA for types (1)–(3) is straightforward. For an expression $R_1 \circ R_2$, where R_1 and R_2 have minimal DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, we construct an NFA that recognizes $L(R_1 \circ R_2)$ using Thompson's construction (Thompson, 1968) and convert it to the minimal DFA using Brzozowski's algorithm (Brzozowski, 1962). Although this can result in an exponential blowup of states, we find that this is not an issue in practice. For an expression $R_1 \cup R_2$ with minimal DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, we construct a DFA that recognizes $L(R_1 \cup R_2)$ using the product construction (Hopcroft et al., 2006) and minimize it using Hopcroft's algorithm. For an expression $\overline{R}$ with minimal DFA $\mathcal{A}$, we make $\mathcal{A}$ complete, flip its accept and non-accept states, and trim any states that cannot lead to an accept state. Note that in order to keep the complement DFA minimal, we need to treat the DFA encoding $\emptyset$, which consists of a single non-accept state, as a special case.

Checking whether the minimal DFA is partially-ordered correctly identifies $\mathscr{R}$-trivial languages and only $\mathscr{R}$-trivial languages. Proof: If the minimal DFA is partially-ordered, clearly its language is $\mathscr{R}$-trivial. If the minimal DFA contains a cycle of size 2 or more, since minimization can never increase the size of a cycle, all equivalent DFAs also contain cycles of 2 or more and are not partially-ordered. Therefore, the language cannot be $\mathscr{R}$-trivial.

We provide pseudocode in Algorithm 6 and Algorithm 7.

Algorithm 6 Sample recursively a star-free regular expression. The probabilities of sampling union, concatenation, and complement operations decay as the depth of the regular expression increases.

def SAMPLESFRE($d, \Sigma, \boldsymbol{w}, s$):
  let $E \leftarrow (\emptyset, \varepsilon, a, \circ, \cup, \neg)$ ▷*Types of regular expression.*
  $(w_\emptyset, w_\varepsilon, w_a, w_\circ, w_\cup, w_\neg) \leftarrow \boldsymbol{w}$
  $\boldsymbol{w} \leftarrow (w_\varepsilon + d * s, w_\varepsilon + d * s, w_a + d * s, w_\circ + \frac{d*s}{3}, w_\cup + \frac{d*s}{3}, w_\neg + \frac{d*s}{2})$
  $p \leftarrow$ NORMALIZE($\boldsymbol{w}$) ▷*Normalize weights.*
  sample e from E according to probabilities p
  switch e :
    case $\emptyset$:
      return $\emptyset$
    case ε:
      return ε
    case a:
      sample $c \sim \mathcal{U}(\Sigma)$
      return c
    case $\circ$:
      return SAMPLESFRE$(d+1, \Sigma, \boldsymbol{w}, s) \circ$ SAMPLESFRE$(d+1, \Sigma, \boldsymbol{w}, s)$
    case $\cup$:
      return SAMPLESFRE$(d+1, \Sigma, \boldsymbol{w}, s) \cup$ SAMPLESFRE$(d+1, \Sigma, \boldsymbol{w}, s)$
    case $\neg$:
      return $\overline{\text{SAMPLESFRE}(d+1, \Sigma, \boldsymbol{w}, s)}$

Algorithm 7 Algorithm for randomly sampling a DFA encoding a star-free language. Here, $\mu_\Sigma \geq 1$ is the mean number of symbols; $\boldsymbol{w} = (w_\emptyset, w_\varepsilon, w_a, w_\circ, w_\cup, w_\neg) \in (0, \infty)^6$ is a vector of weights for sampling the six different expression types; and s is a sharpness parameter that controls how fast the weights of recursive expressions decay as a function of expression depth. The output is a minimal DFA $\mathcal{A}$. SAMPLESFRE recursively samples a star-free regular expression (Algorithm 6), SIMPLIFY simplifies it using heuristics, and SFRETOMINDFA converts it into a minimal automaton.

def SAMPLESFDFA($\mu_\Sigma, \boldsymbol{w}, s, \rho_{\max}$):
  repeat
    sample $|\Sigma| \sim \text{NB}(\mu_\Sigma - 1, 0.5) + 1$ ▷*Sample alphabet size.*
    let $\Sigma = \{a_0, \ldots, a_{|\Sigma|-1}\}$
    $R \leftarrow$ SIMPLIFY(SAMPLESFRE$(0, \Sigma, \boldsymbol{w}, s)$) ▷*Sample a star-free regular expression.*
    $\mathcal{A} \leftarrow$ SFRETOMINDFA(R) ▷*Convert into minimal DFA.*
  until $\mathcal{A}$ is not a PODFA, accepts a non-trivial infinite language, and has density at most $\rho_{\max}$
  return $\mathcal{A}$

B.3 DETAILS OF SAMPLING REGULAR LANGUAGES

Here, we provide more detail about the regular language sampling algorithm described in §3.3.

First, we describe how we check whether the language of a DFA is star-free. We first minimize the DFA using Hopcroft's algorithm. For a DFA with states Q, a **transformation** is a function $f\colon Q \to Q$. For a minimal DFA $\mathcal{A}$ with states Q and transition function δ, the **syntactic monoid** of $L(\mathcal{A})$ is the set $\{f \mid \exists \boldsymbol{w} \in \Sigma^* : \forall q \in Q : f(q) = \delta(q, \boldsymbol{w})\}$. This set is always finite, albeit its size may be as large as $|Q|^{|Q|}$. Representing transformations as tuples in $Q^{|Q|}$, we compute the syntactic monoid by starting with the identity transformation and iteratively composing transformations with $\delta(\cdot, a)$ for all $a \in \Sigma$ using depth-first search.

The DFA's language is star-free iff the syntactic monoid is aperiodic (Schützenberger, 1965); that is, for every f in the syntactic monoid, there is no i such that $f^i = f^{i+1}$, where f^i denotes iterated composition. If there are M transformations in the syntactic monoid, then we only need to check $f^i = f^{i+1}$ for some $i \geq M$, as it must converge after M steps, if ever. We compute f^i using exponentiation by squaring; the rest of the check is straightforward.

We provide pseudocode in Algorithm 8.

Algorithm 8 Algorithm for randomly sampling a DFA. Here, $\mu_Q \geq 1$ is the mean number of states, $\mu_\Sigma \geq 1$ is the mean alphabet size, $p_\delta \in [0, 1]$ is the probability of adding a transition for a given $(q, a) \in Q \times \Sigma$, and $p_F \in (0, 1)$ is the probability of making a state an accept state. The output is a minimal DFA $\mathcal{A}$. MINIMIZEDFA is any DFA minimization algorithm.

def SAMPLEDFA($\mu_Q, \mu_\Sigma, p_\delta, p_F, \rho_{\max}$):
  repeat
    sample $|Q| \sim \mathrm{NB}(\mu_Q - 1, 0.5) + 1$
    sample $|\Sigma| \sim \mathrm{NB}(\mu_\Sigma - 1, 0.5) + 1$
    let $Q = \{q_0, \ldots, q_{|Q|-1}\}$
    let $\Sigma = \{a_0 \ldots, a_{|\Sigma|-1}\}$
    let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA
    for $q \in Q$:
      for $a \in \Sigma$:
        if $\phi \sim \mathrm{Ber}(p_\delta)$:
          sample $r \sim \mathcal{U}(Q)$
          add $q \xrightarrow{a} r$ to δ
      if $\phi \sim \mathrm{Ber}(p_F)$:
        add q to F
    $\mathcal{A} \leftarrow$ MINIMIZEDFA($\mathcal{A}$)
  until $\mathcal{A}$'s language is not star-free, and has density at most $\rho_{\max}$
  return $\mathcal{A}$

B.4 DETAILS OF SAMPLING CONTEXT-FREE LANGUAGES

Here, we provide more detail about the context-free language sampling algorithm described in §3.4.

We compute the probability of sampling a binary or terminal rule by dividing the mean by the number of all possibilities. We cap this probability at 0.9 in order to ensure that it is always possible for a given rule not to be sampled.

We trim the CFG before we add self-embedding chains of rules. Because adding self-embedding chains does not introduce any new variables, the resulting CFG is also trim. This requirement is essential for the correctness of subsequent steps, including the length-constrained string sampling algorithm. To check whether a CFG is finite, we look for cycles in the reachability graph (cf. Clark, 2017).

We provide pseudocode in Algorithm 9 and Algorithm 10.

C DETAILS OF CALIBRATING THE LANGUAGE DISTRIBUTIONS

Here we provide more details about how we calibrate our distributions over languages (see §3.5). We randomly sample parameters for each of our four language sampling algorithms as follows. For all language classes, we fix $\mu_\Sigma = 5$. For $\mathscr{R}$-trivial and regular languages, we sample μ_Q from $\mathcal{U}([5, 15])$, p_δ from $\mathcal{U}([0.2, 0.8])$, and p_F from $\mathcal{U}([0.2, 0.8])$. For star-free languages, we sample each of $w_\emptyset$, w_ε, and w_a from $\mathcal{U}([1, 10])$, we sample each of $w_\circ$, $w_\cup$, and $w_{\bar{}}$ from $\mathcal{U}([50, 1000])$, and we sample s from $\mathcal{U}([1, 1000])$. For context-free languages, we sample μ_V from $\mathcal{U}([5, 15])$, we sample p_ε from $\mathcal{U}([0.2, 0.8])$, we sample μ_b from $\mathcal{U}([3, 7])$, we sample μ_t from $\mathcal{U}([0.5, 1.5])$, we sample μ_r from $\mathcal{U}([0.5, 2])$, and we sample μ_c from $\mathcal{U}([2, 5])$. We give our algorithm for selecting parameters with the smallest span of values in Algorithm 11.

D DETAILS OF LENGTH-CONSTRAINED SAMPLING FOR CONTEXT-FREE LANGUAGES

Setup. Let $\mathcal{G}$ be a *trim* CFG in Chomsky normal form and let $[n_{\min}, n_{\max}]$ be the length range of interest. We convert $\mathcal{G}$ into a PCFG $\mathcal{G}'$ by assigning uniform probabilities to all rules sharing a left-hand side. $\mathcal{G}'$ then defines a distribution $p_{\mathcal{G}'}$ over strings, and sampling a string amounts to sampling a derivation. The grammar being trim is essential: with useless variables, $\mathcal{G}'$ would lose

Algorithm 9 Algorithm for sampling a random trim context-free grammar in Chomsky normal form. Here, $\mu_V \geq 1$ is the mean number of variables, $\mu_\Sigma \geq 1$ is the mean number of terminals, $p_\varepsilon \in [0, 1]$ is the probability of adding the rule $S \to \varepsilon$, $\mu_\mathrm{b} > 0$ is the mean number of binary rules per variable, $\mu_\mathrm{t} > 0$ is the mean number of terminal rules per variable, $\mu_\mathrm{r} \geq 0$ is the mean number of self-embedding chains, $\mu_\mathrm{c} \geq 2$ is the mean self-embedding chain length, and $\rho_{\max}$ is the maximum allowed density of the language.

def SAMPLECFG($\mu_V, \mu_\Sigma, p_\varepsilon, \mu_\mathrm{b}, \mu_\mathrm{t}, \mu_\mathrm{r}, \mu_\mathrm{c}, \rho_{\max}$):
  repeat
    sample $|V| \sim \mathrm{NB}(\mu_V - 1, 0.5) + 1$
    sample $|\Sigma| \sim \mathrm{NB}(\mu_\Sigma - 1, 0.5) + 1$
    let $V_1 = \{X_1, X_2, \ldots, X_{|V|-1}\}$ ▷*Variables excluding the start symbol.*
    let $V = \{S\} \cup V_1$
    let $\Sigma = \{a_0, \ldots, a_{|\Sigma|-1}\}$
    if $|V| > 1$:
      let $p_b = \min(0.9, \frac{\mu_\mathrm{b}}{(|V|-1)^2})$ ▷*Probability of sampling a binary rule.*
    else
      let $p_b = 0.9$
    let $p_l = \min(0.9, \frac{\mu_\mathrm{t}}{|\Sigma|})$ ▷*Probability of sampling a terminal rule.*
    let $R = \emptyset$
    if $\phi \sim \mathrm{Ber}(p_\varepsilon)$:
      add $S \to \varepsilon$ to R
    for $X \in V$:
      for $Y, Z \in V_1$:
        if $\phi \sim \mathrm{Ber}(p_b)$:
          add $X \to YZ$ to R
      for $a \in \Sigma$:
        if $\phi \sim \mathrm{Ber}(p_l)$:
          add $X \to a$ to R
    let $\mathcal{G} = (V, \Sigma, R, S)$ be a CFG
    $(V, \Sigma, R, S) \leftarrow$ TRIMCFG($\mathcal{G}$)
    if $|V| > 1$:
      sample $c \sim \mathrm{NB}(\mu_\mathrm{r}, 0.5)$
      for c times :
        $\mathcal{G} \leftarrow$ ADDSELFEMBEDDINGCHAIN($\mathcal{G}, \mu_k$)
  until $\mathcal{G}$ encodes an infinite language and has density at most $\rho_{\max}$
  return $\mathcal{G}$

probability mass on invalid derivations. Writing $N_{\mathcal{G}'} \stackrel{\text{def}}{=} \{n \in [n_{\min}, n_{\max}] \mid \exists \boldsymbol{w} \in \Sigma^n, p_{\mathcal{G}'}(\boldsymbol{w}) > 0\}$ for the set of lengths the grammar *can* produce and W for a $p_{\mathcal{G}'}$-distributed random variable, our target is the distribution

$$p(\boldsymbol{w}) = \mathbb{1}[\,|\boldsymbol{w}| \in N_{\mathcal{G}'}]\,\frac{1}{|N_{\mathcal{G}'}|}\,p_{\mathcal{G}'}(\mathrm{W} = \boldsymbol{w} \mid |\mathrm{W}| = |\boldsymbol{w}|), \tag{9}$$

from which we sample by first drawing a length n uniformly from $N_{\mathcal{G}'}$ and then drawing a string from $p_{\mathcal{G}'}(\mathrm{W} = \boldsymbol{w} \mid |\mathrm{W}| = n)$. One could instead draw strings from $p_{\mathcal{G}'}$ and discard those of the wrong length (Petty et al., 2025), but this is inefficient whenever the grammar places little mass on the target length.

The allsum. Sampling from the conditional distribution means finding the derivations $S \stackrel{*}{\Rightarrow} \boldsymbol{w}$ such that $|\boldsymbol{w}| = n$ and renormalizing their weights. This requires knowing the total weight of these derivations. The allsum algorithm (Algorithm 13), based on Smith & Johnson (2007), computes a table Z indexed by V in which Z_X is the total weight of all derivations rooted at X. Run over the real semiring, this collapses every length into a single number, whereas we need the weights broken down by the length of the string derived. Computing them one length at a time would mean one run of the algorithm per length. Instead, we change the semiring.

Algorithm 10 Algorithm for adding a self-embedding chain of rules to a CFG $\mathcal{G}$ in Chomsky normal form. Here, $\mu_k \geq 2$ is the mean derivation length.

def AddSelfEmbeddingChain($\mathcal{G} = (V, \Sigma, R, S), \mu_k$):
  sample $k \sim \mathrm{NB}(\mu_k - 2, 0.5) + 2$ ▷*Long-tail distribution with a mean of μ_k*
  sample $k_L \sim \mathcal{U}(\{1, \ldots, k-1\})$ ▷*Number of rules that generate symbols to the left.*
  let $\phi_1, \ldots, \phi_k$ be a uniformly sampled random permutation of k_L $\top$'s and $(k - k_L)$ $\bot$'s
  $V' \leftarrow V \setminus \{S\}$
  sample $X_0 \sim \mathcal{U}(V')$
  $X \leftarrow X_0$
  for $i = 1, \ldots, k$:
    if $i < k$:
      sample $Y \sim \mathcal{U}(V')$
    else
      $Y \leftarrow X_0$
    sample $Z \sim \mathcal{U}(V')$
    if ϕ_i :
      add $X \to ZY$ to R ▷*Generate something to the left.*
    else
      add $X \to YZ$ to R ▷*Generate something to the right.*
    $X \leftarrow Y$
  return $\mathcal{G}$

Algorithm 11 Select parameter settings with minimal baseline accuracy spread. A_i is the list of candidate parameter settings for language class i, sorted in non-decreasing order by $\mathrm{acc}(\cdot)$, where $\mathrm{acc}(h)$ is the mean accuracy of the baseline n-gram model over the languages sampled from setting h. The algorithm returns one setting per class minimizing the difference between the largest and smallest selected acc.

def SelectParameters($A_1, \ldots, A_k$):
  $Q \leftarrow$ min-heap of triples $(\mathrm{acc}(A_i[1]),\ i,\ 1)$ for all i
  $m \leftarrow \max_i \mathrm{acc}(A_i[1])$ ▷*Current maximum.*
  $(\ell^*, u^*) \leftarrow (-\infty, \infty)$
  $c_1^*, \ldots, c_k^* \leftarrow 1, \ldots, 1$ ▷*Current selection indices.*
  while $|Q| = k$:
    $(v, i, j) \leftarrow$ PopMin(Q)
    if $m - v < u^* - \ell^*$:
      $(\ell^*, u^*) \leftarrow (v, m)$
      $c^* \leftarrow c$ ▷*New best selection.*
    if $j < |A_i|$:
      Push(Q, $(\mathrm{acc}(A_i[j+1]),\ i,\ j+1)$)
      $c_i \leftarrow j + 1$
      $m \leftarrow \max(m,\ \mathrm{acc}(A_i[j+1]))$
  return $A_1[c_1^*], \ldots, A_k[c_k^*]$

The binning semiring. The semiring we need keeps a separate weight for each possible string length, so that a single allsum computation returns all of them. We use the binning semiring of Snæbjarnarson et al. (2026), setting its counter to record the number of symbols a derivation generates. Its two operations then behave exactly as we need: addition is componentwise, since alternative derivations of the same length pool together, and multiplication is a convolution, since composing two sub-derivations adds their symbol counts. We therefore **lift** $\mathcal{G}'$ to a WCFG $\mathcal{G}_D$ over the $n_{\max}^{\text{th}}$-order binning semiring (Algorithm 12) by placing each rule's probability at the index equal to the number of symbols it emits: a rule $X \xrightarrow{p} a$ receives $(\mathbf{0}, p, \mathbf{0}, \ldots, \mathbf{0})$, while $X \xrightarrow{p} YZ$ and $X \xrightarrow{p} \varepsilon$ receive $(p, \mathbf{0}, \ldots, \mathbf{0})$. The same interpretation extends to the weight of any derivation or string. Running the allsum algorithm on $\mathcal{G}_D$ returns, for each variable X, a vector $\boldsymbol{z}_X(\mathcal{G}') = (\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{n_{\max}})$ whose entry i is the total weight of all derivations $X \overset{*}{\Rightarrow} \boldsymbol{w}$ with $|\boldsymbol{w}| = i$. One run of the algorithm covers every length in $[n_{\min}, n_{\max}]$ at once.

Renormalization and sampling. To turn these totals into the conditional distribution $p_{\mathcal{G}'}(\boldsymbol{w} \mid |\boldsymbol{w}| = n)$, we exploit the decomposable structure of derivations in Chomsky normal form. A partial derivation rooted at X and yielding n symbols takes one of three forms: (1) a binary rule $X \to YZ$ followed by derivations $Y \overset{*}{\Rightarrow} \boldsymbol{w}_1$ and $Z \overset{*}{\Rightarrow} \boldsymbol{w}_2$ with $|\boldsymbol{w}_1| = i$ and $|\boldsymbol{w}_2| = n - i$ for some split point $i \in [1, n-1]$, (2) a terminal rule $X \to a$, when $n = 1$, or (3) the empty rule $S \to \varepsilon$, available only when $X = S$ and $n = 0$. Forms (2) and (3) are valid whenever the rule itself has non-zero weight. Form (1) is valid when, in addition, both sub-derivations have non-zero weight at their respective lengths, which is exactly what the table Z records. We therefore enumerate the valid options for each variable and length (Algorithm 14) and renormalize their weights into a probability distribution (Algorithm 15), at a cost of $O(|R| \times D^2)$. Sampling a string of length n (Algorithm 17) then proceeds top-down from S: at each step we draw one of the valid options according to these weights, which for form (1) means jointly drawing the rule and the split point i, and recurse on the two sub-derivations. This takes $O(n)$ time per string.

Algorithm 12 Convert a CFG $\mathcal{G}$ in CNF to a WCFG $\mathcal{G}_D$ over the $n_{\max}{}^{\text{th}}$-order binning semiring with respect to the log semiring. This implicitly creates an intermediate PCFG $\mathcal{G}'$ with uniform probabilities over the rules for each variable on the left-hand-side. Time complexity: $O(|R| \times n_{\max})$.

def LIFTWEIGHTSCFG($\mathcal{G} = (V, \Sigma, R, S)$):
  let $\mathcal{G}_D = (V, \Sigma, \psi, S)$) be a new WCFG over the $n_{\max}{}^{\text{th}}$-order binning semiring with respect to the log semiring
  for $X \in V$:
    $k \leftarrow 0$
    for $X \to \boldsymbol{\alpha} \in R$:
      $k \leftarrow k + 1$
    $p \leftarrow -\log k$ ▷*Set the probability to $\frac{1}{k}$ (in log space)*
    for $X \to \boldsymbol{\alpha}$:
      if $\boldsymbol{\alpha} = a$:
        set $\psi(X \to \boldsymbol{\alpha}) = (-\infty, p, -\infty, \ldots, -\infty)$
      else
        set $\psi(X \to \boldsymbol{\alpha}) = (p, -\infty, \ldots, -\infty)$
  return $\mathcal{G}_D$

Algorithm 13 Fixed-point iteration algorithm for computing the allsum of a WCFG $\mathcal{G}$ over the semiring $\mathcal{W}$. Time complexity: $O(|\psi| \times n_{\text{iter}})$, where n_{iter} is the number of iterations required for convergence.

def COMPUTEALLSUM($\mathcal{G} = (V, \Sigma, R, S, \psi), \mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$):
  let $Z^{(0)}$ be a mapping from $V \cup \Sigma$ to $\mathbb{K}$
  for $X \in V$:
    $Z_X^{(0)} \leftarrow \mathbf{0}$
  for $a \in \Sigma$:
    $Z_a^{(0)} \leftarrow \mathbf{1}$
  repeat
    $n \leftarrow n + 1$
    let $Z^{(n)}$ be a mapping from $V \cup \Sigma$ to $\mathbb{K}$
    for $X \in V$:
      $Z_X^{(n)} \leftarrow \mathbf{0}$
    for $a \in \Sigma$:
      $Z_a^{(n)} \leftarrow \mathbf{1}$
    for $X \to \boldsymbol{\alpha} \in \psi$:
      $Z_X^{(n)} \leftarrow Z_X^{(n)} \oplus \psi(X \to \boldsymbol{\alpha}) \otimes \bigotimes_{\beta \in \boldsymbol{\alpha}} Z_\beta^{(n-1)}$
  until $Z^{(n)} \approx Z^{(n-1)}$
  $\mathbf{z} \leftarrow Z_S^{(n-1)}$
  return $Z^{(n)}, \mathbf{z}$

The steps detailed above can be summarized as follows. Given a *trim* CFG $\mathcal{G}$, we run the following pre-processing steps once, before sampling.

1. Convert $\mathcal{G}$ into a PCFG $\mathcal{G}'$ by assigning uniform probabilities to the rules for each variable on the left-hand-side, then convert $\mathcal{G}'$ into a WCFG $\mathcal{G}_D$ over the $n_{\max}{}^{\text{th}}$-order binning semiring with respect to the log semiring (Algorithm 12). Time complexity: $O(|R| \times n_{\max})$.
2. Compute the partition functions $Z_X(\mathcal{G}_D)$ by running the allsum algorithm on $\mathcal{G}_D$ (Algorithm 13). Time complexity: $O(|R| \times n_{\text{iter}})$.
3. Compute the set of all valid partial derivations for each variable X and length $n \in [n_{\min}, n_{\max}]$ (Algorithm 14). Time complexity: $O(|\psi| \times D^2)$.
4. Renormalize the weights of all valid partial derivations for each variable X and length $n \in [n_{\min}, n_{\max}]$ (Algorithm 15). Time complexity: $O(|\psi| \times D^2)$.
5. Compute the subset of valid lengths $N \subseteq [n_{\min}, n_{\max}]$ (Algorithm 16). Time complexity: $O(n_{\max} - n_{\min})$.

Then, we run the following steps once per string.

1. Sample a length n uniformly from the set of valid lengths N. Time complexity: $O(1)$.
2. Sample a string of length n (Algorithm 17). Time complexity: $O(n)$.

Algorithm 14 Given a WCFG $\mathcal{G}_D$ over the Dth-order binning semiring with respect to the log semiring, a binning size D and an allsum table A, compute for each partial derivation rooted at X and length $i \in [0, D]$ a set of valid partial derivations to sample recursively and their weights. Notice that these weights need to be normalized to form a probability distribution. Time complexity: $O(|\psi| \times D^2)$.

def COMPUTEVALIDDERIVATIONS($\mathcal{G}_D = (V, \Sigma, R, S, \psi), D, A$):
  let $T^{(0)}$ be a mapping from V to sets of tuples $\mathbb{K} \times \{\varepsilon\}$
  if $S \xrightarrow{\boldsymbol{v}} \varepsilon \in \psi$:
    add $(\boldsymbol{v}_0, \varepsilon)$ to $T^{(0)}[S]$
  let $T^{(1)}$ be a mapping from V to sets of tuples $\mathbb{K} \times \Sigma$
  for $X \xrightarrow{\boldsymbol{v}} a \in \psi$:
    add $(\boldsymbol{v}_1, a)$ to $T^{(1)}[X]$
  let $T^{(i)}$ for $i = 2 \dots D$ be a mapping from V to sets of tuples $\mathbb{K} \times (V \times \mathbb{Z}_{\geq 0})^2$
  for $X \xrightarrow{\boldsymbol{v}} YZ \in \psi$:
    for $i = 2 \dots D$:
      for $j = 1 \dots i - 1$:
        $w \leftarrow \boldsymbol{v}_0 + A[Y]_j + A[Z]_{i-j}$
        add $(w, ((Y, j), (Z, i - j)))$ to $T^{(i)}[X]$
  return $T^{(0)}, \dots, T^{(D)}$

Algorithm 15 Normalize the weights of the valid derivation subtrees. The input is one of the mappings produced by Algorithm 14. Time complexity: $O(|\psi| \times D)$.

def NORMALIZEDERIVATIONWEIGHTS($T^{(i)}$):
  for $X \in T^{(i)}$:
    $T^{(i)}[X] \leftarrow \text{softmax}_\alpha T^{(i)}[X][:, \alpha]$
  return $T^{(i)}$

Algorithm 16 Given an allsum weight $\boldsymbol{z} \in (\mathbb{R} \cup \{-\infty\})^{D+1}$, a minimum length $n_{\min}$, and a maximum length $n_{\max}$, where $D \geq n_{\max}$, compute the set of valid lengths N. The allsum weight $\boldsymbol{z}$ can be the second output of Algorithm 13 or Algorithm 13.

def COMPUTEVALIDLENGTHS($\boldsymbol{z}, n_{\min}, n_{\max}$):
  return $\{n \in \{n_{\min}, \ldots, n_{\max}\} \mid \boldsymbol{z}_n > -\infty\}$

Algorithm 17 Sample a string of length n from a WCFG $\mathcal{G}$ over the Dth-order binning semiring with respect to the real semiring, where $D \geq n$. Time complexity: $O(n)$

def SAMPLE($\mathcal{G} = (V, \Sigma, R, S, \psi), n$):
  $\boldsymbol{w} \leftarrow$ SAMPLEDERIVATION(S, n)
  return $\boldsymbol{w}$

Algorithm 18 Sample a derivation tree rooted at X that generates a string of length n. Time complexity: $O(n)$

def SAMPLEDERIVATION(X, n):
  if $n = 0$ or $n = 1$:
    sample $a \sim p$ where $p(a) = w$ for (a, w) in $T^{(n)}[X]$
    return a
  else
    sample $(Y, i), (Z, j) \sim p$ where $p((Y, i), (Z, j)) = w$ for $((Y, i), (Z, j), w)$ in $T^{(n)}[X]$
    $\boldsymbol{w}_1 \leftarrow$ SAMPLEDERIVATION(Y, i)
    $\boldsymbol{w}_2 \leftarrow$ SAMPLEDERIVATION(Z, j)
    return $\boldsymbol{w}_1 \boldsymbol{w}_2$

E EXPERIMENTS WITH MORE TRAINING DATA

We test whether increasing the size of the dataset results in better performance. We generate training datasets of 100k samples for 5 of the languages from each language class. Tables 2 and 3 show the average accuracies and the proportions of languages learned perfectly when training with 10k vs. 100k samples. Generally, we observe that adding more training data does not improve accuracy, and it even decreases it in certain cases, although it helps the LSTM learn all regular languages perfectly.

	$\mathscr{R}$-Trivial		Star-Free		Regular		Context-Free	
Architecture	10k	100k	10k	100k	10k	100k	10k	100k
Transformer	$1.00_{\pm.00}$	$1.00_{\pm.00}$	$0.90_{\pm.13}$	$0.89_{\pm.15}$	$0.73_{\pm.05}$	$0.75_{\pm.05}$	$1.00_{\pm.00}$	$1.00_{\pm.00}$
RNN	$1.00_{\pm.00}$	$1.00_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$0.80_{\pm.12}$	$0.80_{\pm.13}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$
LSTM	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$\mathbf{1.00}_{\pm.00}$	$1.00_{\pm.00}$	$1.00_{\pm.00}$

Table 2: Results for multiple training data sizes on a subset of languages. Mean of max test accuracy over 5 languages in each language class, with 10 models per language.

	$\mathscr{R}$-Trivial		Star-Free		Regular		Context-Free	
Architecture	10k	100k	10k	100k	10k	100k	10k	100k
Transformer	0.2	0.0	0.6	0.6	0.0	0.0	0.4	0.2
RNN	0.6	0.6	**1.0**	**1.0**	0.2	0.2	**0.8**	0.4
LSTM	**0.8**	**0.8**	**1.0**	**1.0**	**0.6**	**1.0**	**0.8**	**0.8**

Table 3: Results for multiple training data sizes on a subset of languages. Proportions of languages perfectly out of 5 languages in each language class, with 10 models per language.

F DETAILS OF THE LINEAR MIXED EFFECTS MODEL

Here, we provide more detail about the linear mixed effects model used in §4. For each architecture $a \in \{\text{RNN}, \text{LSTM}, \text{transformer}\}$, language class $c \in \{\mathscr{R}\text{-trivial}, \text{star-free}, \text{regular}, \text{context-free}\}$, and language number $i \in \{1, \ldots, 30\}$, we model the test accuracy as

$$y_{aci} = \beta_0 + \beta_a + \beta_c + \beta_{a\times c} + \beta_{\text{s}} s_{ci} + \beta_{\text{s}\times a} s_{ci} + \beta_{\text{b}} b_{ci} + u_{ci} + \epsilon_{aci}, \tag{10}$$

where β_0 is the intercept; $\beta_a, \beta_c, \beta_{a\times c}$ are coefficients for architecture, class, and their interaction; a and c are encoded using treatment coding with RNN and $\mathscr{R}$-trivial as the reference categories, so that $\beta_{\text{RNN}} = \beta_{\mathscr{R}\text{-trivial}} = \beta_{\text{RNN}\times c} = \beta_{a\times\mathscr{R}\text{-trivial}} = 0$; $\beta_{\text{s}}, \beta_{\text{s}\times a}$ are coefficients for the estimated sensitivity s_{ci} and its interaction with architecture; β_{b} is the coefficient for baseline accuracy b_{ci}; u_{ci} is the random effect for each language; and ϵ_{aci} is the residual error. See Table 4 for full results.

Table 4: Results of the linear mixed effects model defined in Eq. (10).

Parameter	Value	Std. Error	p-value
β_0	0.845	0.043	0.000
$\beta_{\text{star-free}}$	−0.020	0.016	0.220
β_{regular}	−0.119	0.014	0.000
$\beta_{\text{context-free}}$	−0.004	0.013	0.763
β_{LSTM}	0.000	0.012	0.993
$\beta_{\text{transformer}}$	0.006	0.012	0.635
$\beta_{\text{star-free}\times\text{LSTM}}$	−0.003	0.019	0.892
$\beta_{\text{regular}\times\text{LSTM}}$	0.115	0.018	0.000
$\beta_{\text{context-free}\times\text{LSTM}}$	−0.001	0.017	0.972
$\beta_{\text{star-free}\times\text{transformer}}$	−0.037	0.019	0.053
$\beta_{\text{regular}\times\text{transformer}}$	−0.072	0.018	0.000
$\beta_{\text{context-free}\times\text{transformer}}$	−0.001	0.017	0.957
β_{s}	−0.187	0.945	0.843
$\beta_{\text{s}\times\text{LSTM}}$	0.667	1.186	0.574
$\beta_{\text{s}\times\text{transformer}}$	−5.088	1.186	0.000
β_{b}	0.187	0.050	0.000
u_{ci}	0.267	0.097	0.006