

Learning the Lexical Structure of Black-Box Systems by Parsing Systematic String Edit Mutations

Moeketsi Raselimo

Humboldt-Universität zu Berlin
Berlin, Germany
raselimm@hu-berlin.de

Lars Grunske

Humboldt-Universität zu Berlin
Berlin, Germany
grunske@informatik.hu-berlin.de

Bernd Fischer

Stellenbosch University
Stellenbosch, South Africa
bfischer@sun.ac.za

Abstract

We describe and evaluate an approach to learn the lexical structure of black-box systems from a set of positive training examples. Our core idea is to use the system as oracle for repeated membership queries which we construct by systematically mutating the training examples using different string edit operations. We then learn the elements of the lexical structure by generalizing from the oracle responses to the mutated training examples.

We have applied this approach to several systems with target languages of varying complexity, such as TinyC, JSON, AMPL, Modula-2, Oberon, Zig, or C. We show that the learned tokens are similar, and in many cases identical, to those of the oracle lexers. Their inspection can indicate where the systems under test deviate from the lexical structure of their respective target languages; the learned tokens can thus serve as precise test oracles for lexers. We show that the segmentations induced by lexing the validation sets with the learned and oracle lexers, respectively, are identical in many cases and that the non-identical parts affect on average between zero and about 15 percent of the characters. We finally demonstrate that using the learned tokens substantially improves the quality of the context-free grammar that the state-of-the-art grammar learner Kedavra learns for cJSON, increasing the average F1-score of the grammars learned in five runs from 0.93 to 1.00.

CCS Concepts

• **Software and its engineering** → *Parsers*; **Domain specific languages**; • **Theory of computation** → *Rewrite systems*.

Keywords

Grammar learning, Lexical analysis, Mutation testing

ACM Reference Format:

Moeketsi Raselimo, Lars Grunske, and Bernd Fischer. 2026. Learning the Lexical Structure of Black-Box Systems by Parsing Systematic String Edit Mutations. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3834391>

1 Introduction

Knowing the *grammar* of the input language of a system under test (SUT) is useful for many applications, e.g., grammar-based fuzz

testing [49]; an explicit grammar obviously also serves as documentation. However, as with other types of documentation, grammars are surprisingly often inaccurate or simply not available [28, 29, 43], and there has been a long-time theoretical [26, 42] and more recent practical [5–7, 19, 27, 30] interest in learning context-free grammars (CFGs). Learning approaches typically use examples and feedback from the SUT (or *teacher*) [5, 6, 26, 27, 30, 42] but some also analyze the SUT's internal structure [7, 19].

Recent CFG learners have achieved good results [5, 30], but the problem remains far from solved. One major stumbling block is the unstructured nature of the training examples since learning directly from the character representation can cause scalability issues; the low level of abstraction can also impact the quality of the learned grammars. Some tools thus employ an ad-hoc pre-tokenization technique; Treevada [5] and Kedavra [30] have demonstrated the benefits of this strategy even with rudimentary pre-tokenizations. However, this has been seen only as an ancillary step to the more complex full CFG learning, and the problem of learning the lexical structure of an input language has not been addressed in general.

Here we focus on learning the lexical structure using a given set of syntactically valid training examples and a black-box SUT that can only answer membership queries, i.e., can tell whether an input is syntactically valid or not. Our core idea is to use the SUT as an oracle for repeated membership queries which we construct by systematically mutating the training examples using the basic string edit operations of deleting, inserting, and substituting individual characters. We then learn the lexical structure by generalizing from the oracle responses to the different mutations; more specifically, we identify maximal segments in the training examples whose elements produce similar responses. Our core assumption is that these responses reveal information about the target language that we can leverage to gain insight into its lexical structure.

Consider for example the following code fragment in a simple, Pascal-like language:

```
while Count < 1024 do skip
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Here, changing or deleting any of the first five characters leads to a syntax error, as do any insertions between positions 1 and 5, while some insertions before position 1 or after position 5 succeed. We can thus characterize the segment `while` as a keyword. Similarly, we can observe that (i) the same set $\Gamma = \{0, \dots, 9\}$ of characters can be used for substitutions at each position from 15 to 18, (ii) the set of characters that can be inserted between positions 15 and 18 is also Γ , (iii) the set of characters that can be inserted before position 15 or after position 18 includes Γ , and (iv) from positions 15 to 18 any segment can be deleted, if it leaves over at least one character. We can thus infer that 1024 is a sequence of digits, without any preconceptions of numeric literals; note that our approach also infers both the alphabet Γ and



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834391>

the start resp. end positions 15 and 18 of the lexeme. Finally, similar observations, with restrictions on the characters used at position 7, allow us to learn the usual lexical structure of identifier tokens from the single example Count.

Since the learned individual tokens are typically described by regular expressions (REs), our approach has some superficial similarities to RE learning [3, 16]. However, RE learning assumes that the training examples are complete sentences of a regular target language: it needs to be given the individual lexemes (such as Count) to learn the corresponding REs. This assumption does not hold in our scenario since the training examples are complete sentences only of the target language's CFG and consist of an unknown sequence of lexemes corresponding to unknown tokens. We therefore need to learn the segmentation into lexemes and the structure of the tokens simultaneously. This makes the lexical structure inference problem much harder than RE learning.

Consider for example the code fragment `WHILECount<1024DO` in a PL/I-like language where keywords do not require explicit separator symbols. An RE learner needs the list of individual lexemes Count and 1024 to learn the structure of identifiers and integers, respectively, but segmenting the input accordingly is difficult if the tokens are not known—for example, if the language has a keyword `WHILEC`, the first identifier lexeme would be `ount`. Our approach, in contrast, identifies the keyword `WHILE` as before through the limitations on the allowable edit operations within the keyword lexeme itself, independent of its immediate lexical context. Note how the larger syntactic context helps us in learning: for example, changing the third character to `A` induces a *token collusion* between the modified keyword lexeme and the subsequent identifier lexeme, with the result `WHALECount` identifiable as an identifier lexeme again. It is only the surrounding syntactic structure (i.e., the unexpected `DO` keyword) that shows this modification as invalid.

Our approach, sketched in Figure 1, comprises four distinct phases. (i) In the initial *training* phase, we systematically mutate ② the training set ① (see Section 2.3.1 for the applied mutations), and collect the valid mutation operations for each position ⑤ (see Section 2.3.2). We treat the SUT ③ as a black-box but we assume that we can identify whether the mutated training examples remains syntactically valid or not ④. This typically requires a small test harness that filters out messages about semantic errors (such as undeclared identifiers) caused by the mutations. If the SUT is part of a larger system (e.g., a parser embedded in a web application), then the harness also needs to drive the system to the SUT's entry point and extract its output. (ii) In the subsequent *basic munch* phase ⑥, we traverse the collected characteristic mutation sets and identify instances of the basic lexical meta-patterns ⑨ that we describe informally in Section 2.2 and formalize in Section 3. This produces an initial candidate stream of lexemes and corresponding tokens ⑦. (iii) In the main *refinement* phase, we (re-) compose these candidates into more complex structures (see Section 4) that are also captured in the lexical meta-patterns catalogue. We apply a sequence of different transforms ⑧ to the stream and iterate until a fixpoint is reached; we describe as examples how we detect nested comments and strings with escape sequences. We finally (iv) translate the learned patterns into the input notation of a lexical analyzer generator *backend* ⑩; here, we use ANTLR [37].

Note that a coarse approximate segmentation based on common lexical conventions, such as splitting on spaces and punctuation

symbols and grouping segments based on character ranges, misses many token boundaries and splits complex lexemes such as strings or comments. Adding more delimiters (e.g., operator characters) breaks complex operator lexemes and needs increasingly brittle work-arounds that do not generalize well. Our approach, in contrast, relies on a small set of generic meta-patterns that reflect common lexical design practices and uses the SUT's response to instantiate these.

We have implemented our approach in the *lexinfer* tool and used it to successfully learn the lexical structure for several SUTs with target languages of varying complexity, ranging from small languages such as Tiny C [14], JSON [8], and AMPL [44] to full-fledged programming languages such as Modula-2 [46], Zig [25], Oberon [47], and C. We show in Section 5 by inspection of the learned tokens that *lexinfer* indeed learns close approximations of the SUTs input languages' lexical structures; we also show that the token structure can indicate where the SUTs do not follow the lexical structure of their respective target language specifications. We numerically quantify the quality of the learned lexical structure by comparing the similarity of the segmentations that induced by scanning the validation sets with the learned and oracle lexers, respectively. We show that the segmentations are identical for many SUTs and validation inputs, with non-identical parts affecting on average between zero and approximately 15 percent of the program's characters. We finally demonstrate that using the learned tokens substantially improves the quality of the CFG that the state-of-the-art grammar learner Kedavra [30] learns for cJSON, increasing the average F1-score of the grammars learned in five runs from 0.93 to 1.00.

2 Lexical Structure Learning

2.1 Learning Model

Grammar inference methods that learn from examples typically follow one of two learning models. In the *identification in the limit* model [17, 26], the learner is presented with a sequence of observations and constructs a sequence of hypothesis, until it reaches a conclusion. In the *query* model [3], an *active learning* [40] technique, the learner can ask a *teacher* different queries and use the responses to guide the learning process. The widely used L^* algorithm [2] assumes a minimally adequate teacher (MAT) that can answer membership (i.e., $w \in^? L$) and language equivalence queries (i.e., $L \stackrel{?}{=} L'$).

We only require membership queries (where a failed query corresponds to the SUT rejecting it with a syntax error). Similarly to recent black-box grammar inference work [6, 27], we thus use a learning model that is weaker than the standard MAT model.

2.2 Lexical Structure Model

We follow common terminology [1] and distinguish tokens from lexemes. A *token* is an abstraction of a set of lexemes and serves as a terminal symbol in the SUT's grammar. It can be expressed by an RE. A *lexeme* is a concrete occurrence of a token in a specific sentence, i.e., a substring of a given SUT input. It matches the token's RE. We assume that the SUTs read the inputs as character *streams* and not as sequences of records with fixed-length fields, and identify tokens based on the characters only, not their *positions*.

Token confusion occurs when two different tokens can match a lexeme (e.g., when keywords are not lexically distinguished from identifiers) but only one of them is syntactically valid. It is typically resolved

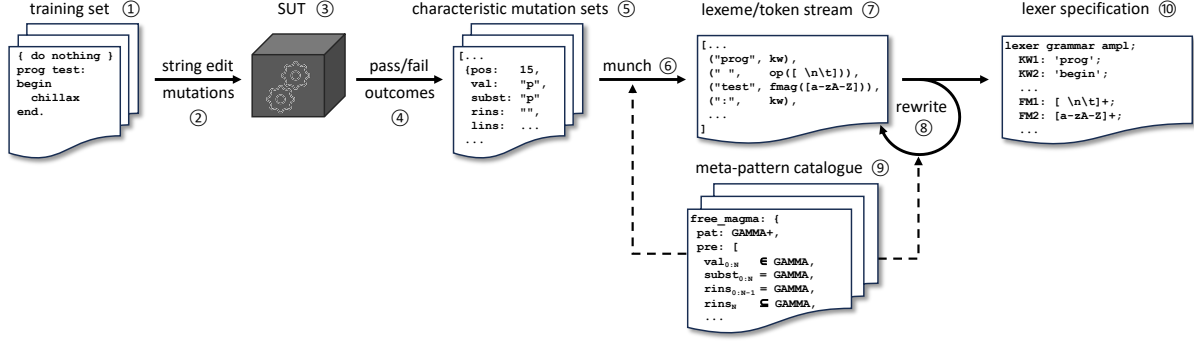


Figure 1: lexinfer: Overview of approach

by the SUT's internal lexer; it does thus not impact our approach. *Token collusion* occurs when a string edit operation leads to a different tokenization of the entire string that is also syntactically valid, e.g., changing an inner digit of an integer literal into a $+$ -character. Token collusion can impact the efficacy of our approach because it can mask the syntactic effect of the lexical changes that we exploit.

We assume that tokens are essentially regular, and do not support run-length encodings where the value of a lexeme's prefix determines its length. We distinguish two kinds of tokens, keyphrases and magmas. *Strict keyphrases* such as keywords or punctuation symbols are fixed-length elements with a unique lexical representation, while *relaxed keyphrases* such as case-insensitive keywords allow some lexical variance. *Magmas* are variable-length strings or character sequences over a given alphabet; we use the term magma here to avoid confusion with actual string tokens.

Most of a language's lexical complexity lies in its magmas. In *free magmas* any character sequence over a given set is valid (e.g., integer literals as sequences of digits) but magmas often have some additional internal structure. In *pointed magmas*, the first or last character of the valid lexemes are limited to designated subsets of the alphabet. *Left-pointed magmas*, which constrain only the first character, are common (e.g., JSON integer literals that cannot start with the digit 0) but Java's decimal integer literals, where underscores are allowed in the inner part of the literal but not at the outer ends are a general example (i.e., both points are the set of digits, while the alphabet also includes the underscore).

Guarded magmas impose similar restrictions on the lexemes' first or last characters (the *left* and *right guards*, respectively) but guards can consist of multiple characters that are not necessarily in the magma's alphabet. In *strictly guarded magmas*, they may in fact not even occur in the character sequence between the two guards, e.g., Modula-2 string literals, where the quotes cannot be escaped. Either of the two guards can be optional; for example, Zig uses @ as variable name prefix, while Modula-2 uses B and C suffixes for octal integer literals. In *escaped magmas* the guards' characters can occur between the guards if they are themselves appropriately guarded or *escaped*, e.g., the classic string escape mechanism using \ as an escape character. Finally, in *punctuated magmas* the character sequence is split into multiple sub-sequences connected by punctuation sequences that are comprised from characters outside the magma's alphabet. An example are floating point literals such as 1.23e-45; here, the magma sequences of digits are split using the punctuations . and e-. Note

that the lexeme comprises the entire sequence but the punctuations themselves do not constitute individual tokens.

2.3 Formalization

We assume a fixed, known *alphabet* Σ and a black-box SUT or *oracle* ω that answers membership queries over Σ^* , i.e., $\omega : \Sigma^* \rightarrow \mathbb{B}$. We also assume a set of non-empty, positive *training examples* Ξ over Σ , i.e., $\Xi \subseteq \Sigma^+$ and $\xi \in \Xi \Rightarrow \omega(\xi)$. Our goal is to learn from Ξ the lexical structure of the unknown CFG G_ω underlying ω , by analyzing ω 's response to a sequence of membership queries that we construct by systematically modifying the training examples Ξ .

2.3.1 String Edit Operations. A string $w = a_1 \dots a_n$ is a sequence of letters $a_i \in \Sigma$. We use $|w| = n$ for its length and $\Sigma(w) = \bigcup_i \{a_i\}$ for the set of its letters. We further use $w_1 \cdot w_2 = a_1 \dots a_n b_1 \dots b_m$ to denote the concatenation of two strings $w_1 = a_1 \dots a_n$ and $w_2 = b_1 \dots b_m$, ϵ for the empty string and $w_{i,j}$ to denote the substring $a_{\max(i,1)} \dots a_{\min(j,|w|)}$ and define $w_{i,j} = \epsilon$ if $i > j$.

We define the usual substitution, deletion, and insertion operations on strings that we use to mutate the training set. Specifically, a *substitution* $\sigma(w, i, a) = w_{1:i-1} \cdot a \cdot w_{i+1:|w|}$, $1 \leq i \leq |w|$, replaces w 's i -th character by a . An *insertion* $\iota(w, i, a) = w_{1:i} \cdot a \cdot w_{i+1:|w|}$, $0 \leq i \leq |w|$, inserts a to the right of w 's i -th character; hence, $\iota(w, 0, a)$ prefixes a to w , while $\iota(w, |w|, a)$ appends it. A *deletion* $\delta(w, i, j) = w_{1:i-1} \cdot w_{i+j:|w|}$, $1 \leq i \leq i+j \leq |w|$, removes j characters, starting from position i ; hence, $\delta(w, i, 0) = w$.

2.3.2 Characteristic Mutation Sets. In the training phase, we check for each position in each training example which of the string edit operations succeed, i.e., yield a modified string that is accepted by the SUT, and record the results as *characteristic mutation sets*. In order to simplify the subsequent analysis, we consider left and right insertion at (resp. deletion from) a position.

Specifically, for position i in a string w we define the set of *permissible substitutions* as $\sigma_{w,i} = \{a \in \Sigma \mid \omega(\sigma(w, i, a))\}$, the set of *permissible left and right insertions* as $\iota_{w,i}^{\leftarrow} = \{a \in \Sigma \mid \omega(\iota(w, i+1, a))\}$ and $\iota_{w,i}^{\rightarrow} = \{a \in \Sigma \mid \omega(\iota(w, i, a))\}$, respectively, and the *maximal permissible left and right deletion* as $\delta_{w,i}^{\leftarrow} = \max k \cdot 0 \leq k \leq i \Rightarrow \forall j \cdot 0 \leq j \leq k \Rightarrow \omega(\delta(w, i+1-j, j))$ and $\delta_{w,i}^{\rightarrow} = \max k \cdot 0 \leq k \leq |w| - i + 1 \Rightarrow \forall j \cdot i \leq j \leq i+k \Rightarrow \omega(\delta(w, j, k))$, respectively. Hence, $\delta_{w,i}^{\leftarrow}$ (resp. $\delta_{w,i}^{\rightarrow}$) is the length of the longest substring ending (resp. starting) at position i whose postfixes (resp. prefixes) can all be deleted. Thus, $\delta_{w,i}^{\leftarrow} = \delta_{w,i}^{\rightarrow} = 0$ iff the i -th character cannot be deleted.

3 Basic Munching

The training phase returns for each character in the training set the corresponding characteristic mutation sets (see Figure 1, ① to ⑤). The basic munch phase ⑥ passes over these sets once and identifies instances of the basic lexical patterns (i.e., keyphrases and free and pointed magmas). It detects the longest prefixes corresponding to each of the basic patterns, and returns an initial tokenization, with characters not munched included as singletons.

3.1 Munching Keywords

3.1.1 Strict Keywords. Keyphrases are fixed-length lexical elements with no or little lexical variability but specific syntactic roles. Keywords are typically composed from letters only but can also include special characters. Most languages require delimiters such as white space or operator symbols around keywords, but our approach does not hard-code this.

The defining characteristic of a *strict* keyword is that *any* modification of any of its lexeme occurrences in a syntactically valid input causes a syntax error, even if the result of the modification is a different keyword (e.g., modifying and to end). We can formalize this using the characteristic sets defined above and deem any lexeme satisfying the criteria to be a keyword lexeme.

Definition 1. $w_{i,j}$ is a *strict keyword* lexeme if it is a maximal sequence with

- (1) $\sigma_{w,k} = \{w_k\}$ for $i \leq k \leq j$, (no substitutions)
- (2) $\iota_{w,k}^{\leftarrow} = \emptyset$ for $i < k \leq j$, (no left insertions)
- $\iota_{w,k}^{\rightarrow} = \emptyset$ for $i \leq k < j$, (no right insertions)
- (3) $\delta_{w,k}^{\leftarrow} = \delta_{w,k}^{\rightarrow} = 0$ for $i \leq k \leq j$. (no deletions)

Note that insertions are allowed before and after the keyword lexeme, but not within. It is easy to see that the sequence while shown in the example in Section 1 satisfies the conditions of Definition 1.

3.1.2 Weak Keywords and Operators. Various languages use weaker forms of keyphrases where the same token can occur with different lexemes. One example are case-insensitive keywords where both lower- and upper-case letters can be used for the individual characters of the keyword. More generally, characters can be drawn independently from different but equivalent character sets. We thus assume an equivalence relation $\equiv \subseteq \Sigma \times \Sigma$, with $\llbracket x \rrbracket_{\equiv}$ denoting the equivalence class of x . For *weak keywords*, we replace the “no substitutions” restriction of Definition 1 with a relaxed version

$$\sigma_{w,k} = \llbracket w_k \rrbracket_{\equiv} \text{ for } i \leq k \leq j \quad (\text{equivalence substitutions})$$

but keep the other restrictions (with a corresponding maximality criterion) in place. Note that we do not require $\equiv$ up front but learn it during inference.

Operators are a yet weaker form of weak keywords, where each position allows a set of equivalent characters specific to that position. Consider for example a language with operators $\equiv$, $<=$, and $>=$, and a lexeme $\equiv$. Here, the equivalence class for the first position is $\{=, <, >\}$, while the second only allows $\{=\}$.

3.1.3 Munching. For strict keywords, the muncher simply iterates over the positions until one of the conditions in Definition 1 fails. For weak keywords, the muncher only checks the insertion and deletion conditions and returns a fixed-length token, where each position is annotated with the valid substitutions. In the example from Section 1

it thus identifies the single-character operator $<$ at position 13, with alternatives $\{<, >, =\}$.

3.2 Munching Magmas

Magma tokens are variable-length character sequences over a given alphabet. A typical example are (decimal) integer literals, represented as arbitrary non-empty digit sequences. However, for many tokens the lexeme structure is restricted; for example, in JSON, integer literals cannot start with the digit 0. Other tokens require more complex sequence combinations, e.g., comments, strings, or floating-point literals. We discuss some of these in Section 4.

3.2.1 Free Magmas. The defining characteristic of a *free magma* token is that the number, order, and specific choice of characters from the magma’s alphabet Γ in any of its lexeme occurrences does not impact the syntactic validity of the program (unless there is token collusion). We can formalize this in terms of the characteristic sets as well.

Definition 2. $w_{i,j}$ is a *free magma* lexeme over the alphabet Γ if it is a maximal sequence with

- (1) $\sigma_{w,k} = \Gamma$ for $i < k < j$, $\sigma_{w,i} \supset \Gamma$, $\sigma_{w,j} \supset \Gamma$ (Γ -substitutions)
- (2) $\iota_{w,i}^{\leftarrow} \supset \Gamma$, $\iota_{w,k}^{\leftarrow} = \Gamma$ for $i < k \leq j$, (left Γ -insertions)
- $\iota_{w,j}^{\rightarrow} \supset \Gamma$, $\iota_{w,k}^{\rightarrow} = \Gamma$ for $i \leq k < j$, (right Γ -insertions)
- (3) $\delta_{w,i}^{\leftarrow} > 0$, $\delta_{w,j}^{\leftarrow} = j - i$ if $i < j$, $\delta_{w,k}^{\leftarrow} > k - i$ for $i < k < j$, (left deletions)
- $\delta_{w,i}^{\rightarrow} = j - i$, $\delta_{w,j}^{\rightarrow} > 0$ if $i < j$, $\delta_{w,k}^{\rightarrow} > j - k$ for $i < k < j$. (right deletions)

The substitutions and insertions are relaxed at the lexeme’s outer positions, to allow additional characters (e.g., spaces or punctuations) in its context. The deletions must leave at least one character from the magma lexeme, but may also remove characters from its context.

Again, it is easy to see that the sequence 1024 shown in the example in Section 1 satisfies the conditions of Definition 2.

3.2.2 Pointed Magmas. In a *pointed magma*, some elements of the alphabet cannot occur at an “outer” end of the lexeme; for example, identifiers can typically not start with a digit. We thus assume two subsets $\Gamma_l, \Gamma_r \subseteq \Gamma$ of left and right *points*, and modify the substitution and left and right insertion rules accordingly:

$$\begin{aligned} \sigma_{w,i} \supset \Gamma_l, \sigma_{w,j} \supset \Gamma_r, \sigma_{w,k} \supset \Gamma \text{ for } i < k < j, & (\Gamma'\text{-substitutions}) \\ \iota_{w,i}^{\leftarrow} \supset \Gamma_l, \iota_{w,k}^{\leftarrow} = \Gamma \text{ for } i < k \leq j, & (\text{left } \Gamma'\text{-insertions}) \\ \iota_{w,j}^{\rightarrow} \supset \Gamma_r, \iota_{w,k}^{\rightarrow} = \Gamma \text{ for } i \leq k < j. & (\text{right } \Gamma'\text{-insertions}) \end{aligned}$$

Similarly, we relax the conditions for the deletions, to capture cases where the first and last character cannot be deleted because the remaining lexeme does not start and end, respectively, with a point:

$$\begin{aligned} \delta_{w,k}^{\leftarrow} &\geq k - i \text{ for } i < k < j, & (\text{left-limited deletions}) \\ \delta_{w,k}^{\rightarrow} &\geq j - k \text{ for } i < k < j. & (\text{right-limited deletions}) \end{aligned}$$

Hence, the Java integer literal $w_{1,5} = 1_2_3$ matches the definition of a pointed magma with left and right points $\Gamma_l = \{1, \dots, 9\}$ and $\Gamma_r = \Gamma_l \cup \{0\}$ and the full alphabet $\Gamma = \Gamma_r \cup \{_ \}$, but not that of a free magma over Γ . For example, $\delta_{w,2}^{\rightarrow} = 4$ (because the sequences $_$, $_2$, $_2_$, and $_2_3$ can all be deleted from position 2 without causing a syntax error), thus satisfying the right deletion condition $\delta_{w,2}^{\rightarrow} > 5 - 2$ of free magmas, but with $\delta_{w,3}^{\rightarrow} = 2$ (because only 2 and $2_$ can safely be deleted from position 3, while deleting 2_3 causes a syntax error) we only have $\delta_{w,3}^{\rightarrow} \geq 5 - 3$.

3.2.3 Munching. Ideally, the muncher should identify the longest possible lexeme (to minimize the number of tokens) with the largest possible alphabet (to allow generalizability), but a smaller alphabet may paradoxically lead to longer lexemes, since it decreases the chance of collusion with keyphrases, due to the substitution and insertion conditions in Definition 2. We therefore try three different munch strategies, with decreasing alphabets, and take the one that produces the longest match. The first two take $\sigma_{w,1}$ (or $\sigma_{w,2}$ to match left-pointed magmas, with $\sigma_{w,1}$ serving as points) as initial guess and munch until one of the conditions in Definition 2 (or the respective modified versions for left-pointed magmas) fails. The third one removes the characters that cause the failure from the alphabet, and munches until this becomes empty; it fails if a previously munched character must be removed. For lexemes of the same length, the muncher returns the result of the first successful strategy. All three strategies succeed only if the munched magma lexeme comprises at least two characters, to prevent overgeneralizations from specific contexts. However, singleton characters are generalized, if possible, in the subsequent refinement phase (see Section 4.3).

As an example, consider a language with a keyword `pop` and identifiers consisting of lower-case letters only. Given the right context for the fragment `tom`, the muncher constructs the correct free magma over the alphabet $\Sigma = \{a, \dots, z\}$ (since all substitutions with lower-case letters succeed) and matches the entire fragment as a single lexeme. For `top`, the first position does not allow substitution with `p` (because of the resulting token collusion with `pop`), and the muncher constructs a left-pointed magma with $\Sigma \setminus \{p\}$ as left points and Σ as alphabet. For `pep`, the substitution with `o` at the second position fails; the muncher thus shrinks the alphabet to $\Sigma \setminus \{o\}$, and, since no `o` has previously been munched, continues munching the entire fragment as in the first case. For `pot`, in contrast, munching fails to identify any magma, since the substitution with `p` fails at the third position, but `p` has already been munched. The muncher thus falls back to returning the first `p` as a singleton character.

4 Tokenization Refinement

The refinement phase takes the initial lexeme/token stream produced by the basic munch phase (see Figure 1 ⑥ and ⑦), and (re-) composes these identified lexemes and corresponding tokens in multiple passes into more complex structures. Each pass implements a specific transformation, which can use information from the previous passes, collated over the entire training set (e.g., the tokens identified so far). The transformations are arranged into multiple layers that each achieve a specific goal (e.g., detecting comments). The layers are run sequentially while the transformations within each layer are run iteratively, until they reach a fixpoint (see Figure 1 ⑧). The first three layers identify comment- and string-like tokens (see Sections 4.2.2 and 4.2.3) and extend keyphrases (see Section 4.1). The subsequent layers refine the structure of simple magmas (see Section 4.3) and identify complex *punctuated* magmas (see Section 4.2.4). A final “thin” layer eliminates rare magmas with occurrence counts below user-definable thresholds that are thus unlikely to generalize well.

4.1 Extending Keyphrases

Not all keyword occurrences can be detected by the basic munch phase. Consider for example the Modula-2 fragment `w = x AND (y).`

Here, the character `A` can be replaced by an operator character (e.g., `+`), leading to syntactically valid mutation because `ND (y)` can be parsed as a function call. However, no other mutations are possible for `A` and neither `N` nor `D` allow any mutations at all, so that the basic munch phase splits `w3:5 = AND` into an operator character with the set of possible alternatives $\sigma_{w,3} = \{A, *, +, \dots\}$ and a keyword `ND`.

We can exploit information from the full training set to re-compose the two tokens into a single keyword token `AND`, *even if the training set does not contain any other occurrence* of `AND`. More specifically, we check whether all operator alternatives other than `A` have been identified as operators in other syntactic contexts; if so, this “explains away” the alternatives, and we give precedence to the interpretation that `AND` is a single keyword. This also allows us to identify multi-character operators such as `>=` and `<=`.

4.2 Guarded and Punctuated Magmas

4.2.1 Guarded Magmas. Like pointed magmas, *guarded magmas* restrict the first or last characters of a magma lexeme; the restricted sequences are called the *left* and *right guards*, respectively, and must be of fixed length; either one of the two guards can be missing.

In *strictly guarded* magmas (e.g., Modula-2 string literals), the guards’ characters *cannot* occur in the character sequence between the two guards, in contrast to the points of pointed magmas.

Definition 3. $w_{i,j}$ is a *strictly guarded magma* lexeme over the generator set Γ with *left* and *right guards* γ_l and γ_r , respectively, if it is a maximal sequence with

- (1) $w_{i,l-1} = \gamma_l, w_{r+1,j} = \gamma_r$ (guard occurrences)
- (2) $(\Sigma(\gamma_l) \cup \Sigma(\gamma_r)) \cap \Gamma = \emptyset$ (guard disjointness)
- (3) $\iota_{w,l-1}^{\rightarrow} = \Gamma$ if $\gamma_l \neq \epsilon$, (left Γ -insertions)
- $\iota_{w,r+1}^{\leftarrow} = \Gamma$ if $\gamma_r \neq \epsilon$, (right Γ -insertions)
- (4) $w_{l,r}$ is a free magma over Γ . (free Γ -magma body)

Note that maximality is relevant only if one of the guards is missing; if both are missing, the guarded magma “collapses” into the underlying free magma. The guard disjointness condition ensures that the magma body cannot accidentally capture the right guard and so prevents the lexeme from extending beyond its first occurrence. The core of the definition are the two Γ -insertion conditions. These ensure that guards are not only immediately adjacent to the magma, but also that nothing can be inserted in-between, and so ensure that merging the guards and the magma into a single lexeme is correct.

4.2.2 Comments. We consider as comments any guarded magmas whose alphabet and guards’ characters together comprise the full character set, because they allow (almost) free text between the opening and closing guards.

We detect comments inductively, checking for guarded magmas with the appropriate alphabets and guards as base cases. For each previously identified comment we then repeatedly check for nested comment occurrences, which are characterized by sequences of magmas over the comment’s alphabet and comment tokens themselves, enclosed by the identified guards. We then look for additional instances of the identified comments, by checking for sequences of tokens between their respective guards; note that the previous checks for recursive comments ensures that any nesting structure remains intact. In a final pass we generalized the comment’s alphabet to the full character set. This mechanism can identify both C++-style line-end

comments (with end-of-line as right guard), block comments with single- or multi-character guards, and even recursive comments.

4.2.3 Strings with Escape Sequences. Like comments, strings allow unstructured text between their guards but limit the alphabet more than comments. We consider as strings with escape sequences any guarded magmas whose alphabet and guard’s characters comprise nearly the full character set, excluding only a single (printable) character (the *escape character*) in addition to characters labeled as non-printable. Within strings, *escape sequences* of the escape character and one or more characters from the alphabet are then used to represent the non-printable characters and other characters not in the alphabet, including the guards and escape character.

We detect such strings in a similar inductive way as comments, again with a check for a guarded magma with the appropriate alphabet and guards as base case. This requires a string without escape sequences to identify the escape character. We then scan for token sequences enclosed in string guards, and collect occurrences of escape sequences, using the fact that the escape character is not matched by the magma’s alphabet and remains a singleton character. Due to the conceptual lexical similarity between comment and string tokens some strings are mis-classified as comments; we patch these once we have identified the string’s guards and escape character.

4.2.4 Punctuated Magmas. Punctuated magmas use “punctuation sequences” to glue together multiple magma structures into a single lexeme (and corresponding token). The classic example are floating-point literals, such as in JSON, where (up to) three magmas over digits can be glued together using two punctuation sequences, the decimal stop `.` and various forms of exponentiation signifiers, such as `e` or `E-`. The formal definition follows Definition 3 for guarded magmas and is omitted here.

4.3 Refining Simple Magmas

Specific character sequences or syntactic contexts can cause the basic magma munch algorithm sketched in Section 3.2 to infer wrong alphabets or to split the lexemes. We thus improve the structure of the inferred magmas by extending the matched lexemes and normalizing the magma alphabets. Our implementation uses 24 different transformations in four stages, with some transformations applied in multiple stages. We thus focus here on a few example transformations.

The first case we consider is if the muncher fails to identify w as a magma (similar to the example shown in Section 3.2.3) and instead returns a singleton character w_1 followed by a magma $w_{2:n}$ with alphabet Γ . We can merge the singleton back into the magma if the operations on w_1 now satisfy the magma conditions with respect to the new Γ (see Sections 3.2.1 and 3.2.2), in particular $\sigma_{w,1} \supset \Gamma$ and $\iota_{w,1}^{\leftarrow} \supset \Gamma$. We handle cases where the singleton follows the magma similarly.

We merge two adjacent magma lexemes w and w' over alphabets Γ and Γ' , respectively, into a single magma lexeme $w \cdot w'$ over the alphabet $\Gamma \cap \Gamma'$, if $\Sigma(w \cdot w') \subseteq \Gamma \cap \Gamma'$, i.e., the intersection of the alphabets contains all characters occurring in w and w' . Otherwise, we identify the largest split position k for w such that $\Sigma(w_{1:k}) \subseteq \Gamma \setminus \Gamma'$ and $\Sigma(w_{k+1:|w|} \cdot w') \subseteq \Gamma \cap \Gamma'$, i.e., the left part of the first lexeme is captured by the difference between the two alphabets and its right part by the intersection; we then split the first lexeme and concatenate its

right part to the second lexeme, and adjust the two magmas’ alphabets to the difference resp. intersection. This “rebalancing” results in smaller alphabets.

We also replace singleton characters w_i by magmas with smallest alphabets that include $\sigma_{w,i}$ and regularize all magma alphabets. We remove single-character keywords and operators from the alphabets if they do not occur in the lexemes, and replace magmas by more common ones that match the same lexemes in different contexts, first trying magmas with smaller and then with larger alphabets.

5 Experimental Evaluation

5.1 Research Questions

We answer three research questions with our experimental evaluation. First, we investigate how “natural” the inferred lexers are:

RQ1 How similar are the inferred lexers to those of the SUTs’ underlying target languages?

We answer this by manually analyzing the structure of the inferred lexers. We also manually analyze whether the inferred lexers indicate where the SUTs deviate from the oracle grammar.

Second, we investigate how well lexinfer’s learned tokens generalize, more specifically:

RQ2 How similar are the segmentations produced by the lexers inferred by lexinfer to those produced by the SUTs?

We answer this by aligning the respective lexeme streams produced over unseen test files and computing the relative size of the unaligned segments. Note that different lexers can still produce identical segmentations; for example, the inferred lexer could have multiple patterns for the same token (e.g., floating point literals with and without exponent), or miss keywords and match identifiers instead, and still return the same segmentation.

Third, we investigate possible “downstream” benefits of using the inferred lexical structure, specifically full CFG inference:

RQ3 Can the use of inferred tokens improve the learning of context-free grammars?

5.2 Experimental Setup

We implemented our approach in the lexinfer tool, which is available at <https://github.com/gtoolsuite/lexinfer>. It comprises about 6k lines of Python code and follows the structure outlined in Figure 1. We used this in our evaluation to learn the lexical structure of SUTs for different target languages. We chose languages ranging from small languages previously used in grammar inference [27] and grammar-based fuzzing [32] (Tiny-C, JSON), over teaching examples (AMPL, ALAN, SIMPL), to fully-fledged programming languages (Modula-2, Oberon, C, Zig). They allow simple wrappers and fit our lexical model (see Section 2) but still cover a wide range of lexical constructs (see details below and in Table 1). We used various SUTs for the target languages (see details below); as oracles we used ANTLR lexers that we collected from the ANTLR grammar repository [11] (Tiny-C, JSON, Modula-2, Oberon, C) or manually transcribed from the target language documentation (AMPL, ALAN, SIMPL, Zig).

For comparison purposes, we also implemented a simple generic tokenizer that splits the input into segments based on character ranges (digits, letters, alpha-numerics, white space) only, similar to previous pre-tokenization work in grammar inference [5, 30].

5.2.1 Tiny-C. Tiny-C [14] is a greatly stripped down C version with only 13 keyphrases and 2 magmas (integers and white space); identifiers are limited to single lower-case characters. We use the `tiny` compiler as SUT. For training, we used the examples from the implementation; these are all single-line strings. For validation, we randomly generated 80 programs from the ANTLR grammar.

5.2.2 JSON. JSON is a common evaluation target for grammar inference tools because it has a highly recursive yet compact grammar; the ECMA standard defines 9 keyphrases and 3 magmas (integer and real numbers, strings with escapes, and white space) [8]. We used five JSON SUTs; three readers/writers (`gson` [18], `json-c` [24], and `JSON-java` [23]), a validator (`cJSON` [15]), and an ANTLR parser based on the ECMA standard (`json-antlr` [36]).

We used 11 training examples generated by ChatGPT that cover the full JSON syntax (except strings with escaped UTF-encodings, as our implementation does not handle long escape sequences yet). For validation we used the 23 test files from the `cJSON` implementation.

5.2.3 AMPL, ALAN, SIMPL. These are similar medium-sized teaching example languages. Their grammars mix aspects of Pascal and C and comprise 5 magmas (white space, nested comments, identifiers, strings with escape sequences, and integer literals). We used the instructor’s “golden” compiler implementations as SUTs.

We used the (passing) programs created by the instructor to test the students’ parsers as training data. These contain nested comments and escaped strings (with `\n` escapes only). We used the final marking test suite (AMPL) resp. the failing programs from the test suites (ALAN, SIMPL) for validation.

5.2.4 Modula-2 and Oberon. Both languages use typical Wirth-style syntax that comprise 67 and 58 keyphrases, 9 and 8 magmas (white space, nested comments, identifiers, single- and double-quoted strings without escape sequences, octal, decimal, and hexadecimal integer literals, and floating-point literals), respectively [46, 47]. We used ANTLR parsers for both languages [12, 13] as SUTs.

For Modula-2, we used different directories from a web repository [34] for training and validation. The training data does not contain nested comments, hexadecimal integer literals, floating-point literals with exponent parts, nor 12 of the keyphrases. We deliberately chose an incomplete training set to see how well the learned lexer generalizes in a realistic scenario where complete training data cannot be expected. For Oberon, we used a similar web repository [45].

5.2.5 Zig. Zig [25] is a recent general-purpose programming language with a rich syntax comprising 112 keyphrases and 11 different magmas, including white space, three different line-end comment styles, three different identifier kinds, binary, octal, decimal, and hexadecimal integer literals, decimal and hexadecimal floating-point literals, character and string literals, both with different escape sequences, and multi-line string literals.

We use two SUTs for Zig. `zig-fmt` calls the compiler (v0.14.1) with the `fmt` command line option and formats the input program, with only limited syntax checking. `zig-peg` is a stand-alone parser that is generated directly from the parsing expression grammar given in the language definition. We used 35 Zig “cookbook” programs [50] for training, and the `zig-peg` test suite [51] for validation.

Table 1: Summary of experimental data sets.

	Oracle Grammar			Training				Validation						
	P	N	T	T	files	chars.	tokens	T	files	chars.	tokens			
Tiny-C	19	7	13	2	13	2	6	208	168	13	2	80	4876	3568
JSON	12	6	9	3	9	3	11	2736	617	9	3	23	20852	5204
AMPL	45	22	43	5	43	5	19	6273	2571	41	5	43	11293	4221
Modula-2	117	68	67	9	51	6	18	22835	5406	62	6	35	100139	23762
Zig	211	102	112	11	58	11	35	61139	17323	106	10	104	426141	146820
ALAN	45	20	43	5	43	5	19	4659	2285	35	5	42	2820	1213
SIMPL	51	24	43	5	42	5	19	5951	2290	32	4	44	3378	1200
C	258	118	109	11	63	6	147	85370	29436	54	6	8	34272	10756
Oberon	123	57	58	8	41	6	19	10705	3945	49	7	10	10522	4062

|P| and |N| are the numbers of rules resp. non-terminals in the oracle grammar, while |T| shows the number of keyphrases and magma tokens defined in the oracle grammar and occurring in the training and validation sets, respectively. For the training and validation sets, the numbers of files and their total length in characters and tokens according to the oracles are shown, respectively. Languages listed in *italic* were only analyzed after the initial implementation of `lexinfer` was completed.

5.2.6 C. The C99 syntax has 109 keyphrases and 11 different magmas, including white space, line-end and block comments, identifiers, integer literals over various bases, decimal and hexadecimal floating-point literals, character and (multi-line) string literals, both with escape sequences including hexadecimal and UTF-8 character encodings. We use `gcc V11.3.0` as SUT, together with a harness that checks for error messages that indicate lexical or syntactic errors.

We used a set of simple educational C-programs for training and validation; we removed all pre-processor directives. The training set uses only basic constructs and lacks for example octal and hexadecimal literals, floating-point literals with exponents, or multi-line string literals.

5.3 Results

5.3.1 Tiny-C. `lexinfer` correctly learns all keywords and the magma structures for integer literals and white spaces, generalizing the latter from single-character examples only. It also learns the structure of identifiers as an operator with 26 variants, over the full set of lower-case letters, but does (correctly) not overgeneralize to multi-character identifiers. However, our ANTLR backend splits the variants into individual tokens and `lexinfer` thus constructs a lexer with 41 tokens; this nevertheless produces segmentations of the validation data that are identical to that of the oracle.

5.3.2 JSON. For JSON, the results vary substantially across the SUTs; we discuss them in turn.

`gson` and `json-antlr`. Both SUTs implement the ECMA-grammar exactly and `lexinfer` learns the same lexical structure for both. Figure 2 shows on top the ANTLR-notation for the learned lexer. It correctly identifies all keyphrases (K1-K9) and the free magma M13 for white space. It also learns a single guarded magma pattern M1 for strings that is largely equivalent to the more structured representation in the oracle lexer, except that the latter allows the full unicode-based range of characters, and supports unicode escape sequences `\uxxxx`.

The oracle lexer uses a single token and two fragment definitions for numbers, which encode 32 structurally different variants through top-level alternatives and optional elements. `lexinfer` learns an unfolded version with 14 different tokens, due to a lack of domain knowledge that these are the same lexical entity; note that these could

```
M1: '"' '\x\b' | '\xf' | '\n' | '\r' | '\t' | '\u' | '\'' | '\\\' | '\\\\' [ascii]*? ''';  
M2: '-'[1-9]+; M3: '-'[0-9]+;  
M4: [0-9]+'(' | 'E' | 'e') [0-9]+;  
M5: [0-9]+'(E+' | 'E-' | 'e+' | 'e-') [0-9]+;  
M6: [0-9]+'(') [0-9]+'(E' | 'e') [0-9]+;  
M7: [0-9]+'(') [0-9]+'(E+' | 'E-' | 'e+' | 'e-') [0-9]+;  
M8: '-'[0-9]+'(') [0-9]+;  
M9: '-'[0-9]+'(') [0-9]+'(E' | 'e') [0-9]+;  
M10: '-'[0-9]+'(') [0-9]+'(E+' | 'E-' | 'e+' | 'e-') [0-9]+;  
K1: ','; K2: ':'; K3: ';'; K4: '?'; K5: '['; K6: ']'; K7: 'null'; K8: 'true'; K9: 'false'; K10: '0';  
  
K11: [0-9] '(' | 'E' | 'e') [0-9];  
M11: [1-9][0-9]*; M12: [1-9][1-9]*; M13: [ \f\n\t][ \f\n\t]*;
```

```
M1: '"' '\x\b' | '\xf' | '\n' | '\r' | '\t' | '\u' | '\'' | '\\\' | '\\\\' [ascii]*? ''';  
M2: '-'[0-9]+; M3: '-'[.,0-9]+;  
M12: [0-9][.,0-9]*; M13: [0-9][0-9]*;
```

Figure 2: Inferred JSON lexers in ANTLR-notation: gson / json-antlr (top) and cJSON (bottom). Edited for brevity.

easily be merged by a subsequent full CFG inference phase. It captures positive integers correctly with three patterns, K10 for a single zero, the correct left-pointed magma M11, and a free magma M12 that is pre-empted by M11 and thus never matches. For negative integers, it internally constructs the correct pattern (i.e., a guarded magma over M11) but then overgeneralizes this to M3, since our implementation restricts guarded magmas to underlying free magmas. For floating-point literals, lexinfer constructs the punctuated magmas M4 to M10 for 7 out of the 8 cases encoded in the oracle lexer; the missing case (i.e., negative literals without a decimal point but a signed exponent) is also missing from the training set. Due to the implementation restriction the punctuated magmas also overgeneralize from the underlying left-pointed magmas, and allow leading zeros. K11 generalizes the training value 0.0 but is pre-empted by M4 and never matches.

cJSON. Figure 2 shows at the bottom parts of the lexer that lexinfer learns for cJSON, specifically the differences to gson. The guarded magma M1 captures the structure of strings accepted by cJSON; this shows that cJSON is not fully ECMA-compliant because it (unlike gson) accepts `\u` as an escape even if it is not followed by four hex digits; we confirmed this by code inspection.

M2 and M13 capture cJSON’s integer literals; M13 shows that these are not ECMA-compliant either because they incorrectly allow literals with leading zeroes (again unlike gson). The remaining patterns relate to the structure of cJSON’s floating-point literals. Specifically, lexinfer learns the left-pointed magma M12 that allows digit sequences with multiple decimal points. While this overgeneralizes inserting a single decimal point into different positions of an integer literal, it can be learned only because cJSON accepts floating-point literals such as 123. that end on the decimal point, in violation of the ECMA-grammar. Note that, unlike in gson, the leading zeroes in the punctuated magmas are not a consequence of lexinfer’s overgeneralization but reflect a violation of the ECMA-grammar. Overall, the grammar overgeneralizes but for the validation data it still produces segmentations that are identical to that of the oracle.

json-c. Here lexinfer learns that json-c uses case-insensitive versions of the keywords null, true, and false, without any example in the training set. This shows that the probing done during the training phase can extract “hidden” information from the SUT. lexinfer also learns a left-pointed magma $[0-9][\cdot 0-9Ee]*$, which shows

that `json-c` also accepts floating-point literals such as `123e` that end on the exponent separator, in violation of the ECMA-grammar. This in turn prevents `lexinfer` from learning punctuated magmas for some floating-point literals, and leads to a number of unmatched fragments that induce incorrect tokens (e.g., `E` and `e`) The resulting grammar has 26 tokens; unlike `cJSON`, this is not a pure overgeneralization, and the lexer correctly segments only 19 of the validation tests.

JSON-Java. While “adherence to the JSON specification” is one of JSON-Java’s stated design goals [23], it actually accepts a large superset of JSON; in particular, it allows unquoted keys and values, which subsume keywords and numeric values, but not strings. *lexer* thus correctly identifies only the six operators and the white space and string magmas (including a documented extension of unescaped tab characters), and finds different magmas over almost the complete printable character set, excluding only the operator symbols, and the characters `;`, `/`, and `=`. However, since these magmas also include the space character, most white space tokens are not identified separately, and the inferred lexer produces segmentations with many deviations from the ECMA specification.

5.3.3 *AMPL*. *lexinfer* identifies all but two of the 43 keyphrases, the boolean literals *true* and *false*. However, these are lexically and syntactically indistinguishable from identifiers.

lexinfer also learns the full white space structure as free magma over the space, newline, and various tab characters, even though the training set only uses spaces and newlines and the the full format is undocumented. This demonstrates the use of the learned lexical structure as explicit documentation. lexinfer learns the correct free magma over digits for integers and left-pointed magma for identifiers. However, it also constructs additional redundant tokens matching restricted instances, but the ANTLR backend arranges the tokens such that these are never actually used. Finally, lexinfer learns the correct magmas for nested comments and strings (with the correct escape set and the correct alphabet).

Overall, the inferred lexical structure is of high quality, with only two missed tokens that are unlearnable, and four extra tokens that are never applicable. The corresponding lexer thus produces identical segmentations to the oracle lexer.

5.3.4 Modula-2. For Modula-2, lexinfer identifies all but two of the keyphrases that actually occur in the training data. It misses the EXIT keyword because all mutations are syntactically valid, as identifiers are actually statements (i.e., parameter-free procedure calls) in Modula-2, and it mis-identifies the ^-operator as a free magma together with white space characters because the syntax allows arbitrarily many ^-characters at each occurrence in the training set. Hence, in both cases the syntactic context provided by the target language is too weak for our approach to work perfectly.

lexinfer learns the correct pattern for identifiers, but also a redundant free magma over letters only that stems from occurrences without digits in the training set. It correctly learns a free magma over digits to match decimal integer literals and guarded magmas for octal integer literals (with B and C postfixes) from just a single example of each in the training set.

lexinfer also finds the correct guarded magmas for comments, in addition to versions with different smaller alphabets due to different relaxations applied during the inference process. It also finds

two string magmas (using both single and double quotes); it (mis-) identifies / as escape character, but does not find any actual escape sequences. This is, however, due to a fault in the SUT, where / is not allowed as character in strings.

5.3.5 Zig. For both SUTs, lexinfer learn most keyphrases in the training set and infers a correct left-pointed magma for identifiers; it also learns appropriate magmas for single- and double-quoted strings.

`zig-fmt`. `zig-fmt` is overly permissive (e.g., the learned double-quoted string magma shows that incorrectly every character can be escaped) and in many cases lexinfer learns unsuitable token. It overgeneralizes, similarly to the negative integer literals in JSON, one of the two guarded magmas for built-in identifiers (which use a prefix-@), and approximates line comments as strings, with the doc-comment marker ! as escape character. However, crucially lexinfer learns a wrong magma for whitespace and fails to identify multi-line string literals, falling back to a free magma over most of the character set instead, which matches complete lines. This results in relatively frequent and long misaligned segments, leading to a misalignment rate in the range of ~5% to ~40% (see Figure 3).

`zig-peg`. `zig-peg`'s stricter syntax checking allows lexinfer to handle magmas much better. It now learns correct guarded magmas for both single- and double-quoted strings. It also learns two guarded magmas for line comments, but over-generalizes due to Zig's three distinct line comment forms: regular comments initiated with //, and two documentation-specific variants, /// and ///. These distinctions are not fully preserved in the inferred structures. It also learns correct magmas for integer and floating-point literals in the integer-mantissa variant (i.e., floats without an explicit decimal point), although standard decimal float forms are not inferred. Most importantly, however, lexinfer now correctly identifies the guarded magma for line strings, which eliminates the fall-back magma from `zig-fmt`. Overall, this leads to a much smaller misalignment rate, which is now mostly below 5%.

5.3.6 ALAN and SIMPL. ALAN and SIMPL are structurally similar to AMPL, despite some differences in the lexical surface structure (e.g., different keyphrases), and lexinfer performs very well here, even without any implementation modifications. For `alan`, it again learns a lexer that induces a perfect segmentation (cf. Figure 3). For `simpl`, it only fails to learn the structure of nested comments from the single example in the training set. This is caused by SIMPL's two-character guards (* / *), while AMPL and ALAN only use single characters ({ / }); it also leads to a spurious string pattern that matches the tail of nested comments. Hence, the inferred lexer correctly segments 41 of the 42 validation tests, with an overall misalignment of less than 1%.

5.3.7 C. lexinfer correctly learns appropriate patterns for the major structural elements of C: all keyphrases occurring in the training set, white space, identifiers, line-end and block comments, strings and character constants (both with escapes). However, due to the weaker syntactic structure of C, and gcc's interleaving of syntax and semantic checking, the lexer has several imperfections, e.g., typenames are typically identified as keywords and floating-point literals are represented as an overgeneralized left-pointed magma as in `json-c` (see Section 5.3.2). It also contains several magma and operator tokens that should be redundant but still impact the lexing process and the

inferred lexer correctly segments only 1 of 8 validation tests, with a small overall misalignment of about 2%.

5.3.8 Oberon. Here, lexinfer learns an almost perfect lexer, relative to the training set. It identifies all occurring keyphrases, although it mis-identifies the unary minus operator ~ as a free magma, similar to the case of Modula-2's ^-operator, and represents the infix operator MOD as left guard for a magma. It learns the oracle's patterns for white space, decimal integers, identifiers (with two additional redundant free magmas over smaller alphabets), strings, and comments. However, it classifies the string tokens as comment magmas, because Oberon has no string escape mechanism. For floating point literals, it constructs a simple punctuated magma, since all occurrences in the training set follow that pattern.

However, the inferred lexer produces correctly aligned segmentations only for 5 of the 10 validation tests. The overall misalignment is nevertheless small (about 1%), showing that lexinfer can work well even from an impoverished training set.

5.4 Integration with Grammar Learning

As a demonstration of the potential “downstream” benefits of inferring the lexical structure we consider the use of the inferred lexer in full grammar inference, as formulated in RQ3.

Ideally, we would use the lexer to tokenize the training set, and then simply run a black-box grammar learner over the token stream to learn the SUTs full CFG. Unfortunately, state-of-the-art grammar learners such as Treevada [5] or Kedavra [30] assume character-level inputs which they pre-tokenize internally, and are not designed to simply “swap in” a new lexer for the pre-tokenization. We thus modified the tokenizer module in Kedavra to investigate the effectiveness of using the inferred lexer. Specifically, we replaced the generic lexer definition and character-level generalization and fed the system with the lexical rules from our cJSON lexer in Figure 2; we kept the remaining framework as is. We chose cJSON as representative JSON system because the high quality of the tokens learned by lexinfer showcases the effect of a better pre-tokenization. We did not experiment with Tiny-C, because its grammar is trivial, and any pre-tokenization works well (see also Figure 3, right). On the other hand, JSON is the most complex grammar for which Kedavra's inference terminated (independent of the pre-tokenization), given our validation data.

We compare against Kedavra (revision 44b8cb6) as a baseline. Both Kedavra versions learn from our training set described in Section 5.2. We ran the experiments five times to obtain the average results in Table 2. The experiment shows that feeding the system with learned tokens improves the quality of the output grammar; the average F1 score goes from 0.93 to 1.00. Grammar readability also improves significantly, as reflected in the reduction in *NT*, *A*, and *S*. We can already see the benefit of learning the specific lexical structure of the SUT and, more generally, separating learning lexical and context-free structure into two phases.

5.5 Answers to Research Questions

RQ1. A manual inspection of the lexers learned for the SUTs shows that lexinfer can in many cases (`tincy`, `gson`, `json-antlr`, `ampl`, `alan`, and `simpl`) learn close approximations of the target language lexers. It correctly learns the structure of complex tokens such

Table 2: Grammar quality and readability across five Kedavra runs with cJSON.

	<i>P</i>	<i>R</i>	<i>F1</i>	t_{lex}	t_{cfig}	<i>NT</i>	<i>A</i>	<i>S</i>
Kedavra	0.91	0.95	0.93	-	126	139	8437.8	8673.4
Kedavra + lexinfer	1.00	1.00	1.00	50	2.5	18.6	142.2	171.2

P and *R* are the standard precision and recall metrics. t_{lex} is the inference time for lexinfer and t_{cfig} is inference time for kedavra, both in minutes. Training set for both versions is as described in Section 5.2. Note that t_{lex} reflects one run for lexinfer. *NT*, *A*, and *S* denote the number of non-terminals, rule alternatives, and sum of rule lengths, respectively.

as strings with escape characters, nested comments, and floating-point literals. Some minor differences and shortcomings are caused by smaller implementation limitations. The lexers can also contain a few redundant patterns that are either subsumed or pre-empted by more general patterns.

In some cases (ampl, alan, simpl, and m2-antlr) token collusion prevents lexinfer from learning some keywords, since all mutations are interpreted as identifiers. However, when the language or the SUT are too permissive (e.g., json-java and zig-fmt) and the SUT does not reject enough mutations lexinfer overgeneralizes magma patterns so that they match too large fragments.

The quality differences between the lexers inferred for SUTs in the test set (i.e., analyzed already during the development of lexinfer) and in the validation set (i.e., only analyzed afterwards) are relatively small. This indicates that our meta-pattern catalog (and rewrite rules) are sufficiently generic.

Interestingly, lexinfer has also identified several errors in SUTs where they deviate from the (purported) target language’s lexical syntax, e.g., case-insensitive keywords in json-c or the missing ‘/’-character in Modula-2 strings.

RQ2. Figure 3 shows on the left the relative misalignment of the segmentations produced by the inferred and oracle lexers, respectively. We see perfectly aligned segmentations for 6 of the 14 SUTs; the two outliers json-java and zig-fmt are also the most permissive SUTs (see above), for which lexinfer learns overly general magma tokens. The overall misalignment rate aggregated over all SUTs and validation sets is about 6%. However, this is biased by the outlier zig-fmt; the average misalignment rate over all SUTs is about 1.5%.

On the right, Figure 3 shows the corresponding data for the default tokenizer (see Section 5.2). We see a much higher fraction of mis-aligned input (typically caused by splitting complex tokens such as floating-point or string literals, or comments) and consequently a much lower number of correctly segmented files.

RQ3. Table 2 shows that using the learned cJSON-lexer instead of the built-in pre-tokenizer substantially improves Kedavras performance: the F1-score increases from 0.93 to 1.00, while the inference time and grammar size decrease by 98%.

6 Limitations and Threats to Validity

Limitations of learnable lexical structures. lexinfer treats the inputs as character streams; languages with fixed column-oriented formats or length-restricted or even run-length encoded tokens are therefore out of scope. It primarily learns lexical structures that are explicit in the training set (e.g., it does not learn keywords missing in the

Table 3: Training and inference times.

SUT	Train	Infer	SUT	Train	Infer
JSON-java	50 min	17 sec	TinyC	1 min	2 sec
json-antlr	50 min	10 sec	ampl	15 min	30 sec
json-c	45 min	11 sec	m2-antlr	12 hrs	70 sec
cJSON	50 min	10 sec	zig-fmt	9 hrs	9 min
gson	45 min	9 sec	zig-peg	8 hrs	11 min
alan	15 min	30 sec	simpl	15 min	30 sec
gcc	14 hrs	22 min	oberon-antlr	3 hrs	42 sec

training set). However, from the SUT feedback, lexinfer can also learn implied structures that are not explicit, e.g., missing operators.

Our approach learns the lexical structure only through the fact that the SUT reports syntax errors for some of the mutations. The inference results thus depend on which of the SUT messages the harness identifies as syntax errors. For example, we cannot identify reserved words as keywords, since SUTs report using undeclared names as a semantic error; conversely, gcc returns a syntax error (“type expected”) for mutations that should cause a semantic error (“undeclared type name”), leading to the mis-identification of some type identifiers as keywords. The harness could analyze the error messages, but this cannot be automated.

The basic munch phase generalizes the magma structures from single character substitutions and insertions. We thus assume that characters can be inserted repeatedly, which can, in combination with token collusion, lead to overgeneralizations. For example, if floating point literals require digits only before or after the decimal point (but not both), then lexinfer would generalize integer literals to a free magma over digits *and* the decimal point, i.e., allow repeated decimal points in digit sequences. This could be mitigated by re-running the SUT to check magmas for repeated insertions of characters that do not occur.

Scalability. The training phase checks substitutions and insertions for each character in the alphabet at each position of the training set; the number of checks (and thus calls to the SUT) is thus linear in the size of both the alphabet and the training set (i.e., $O(|\Sigma| \cdot |\Sigma|)$), both in the worst and average case. It also checks deletions of increasingly longer left and right segments from each position, until the first error is encountered. In the worst case, this grows quadratically in the length of each training sample (i.e., $O(\sum_i |\xi_i|^2)$) but in practice the number of successful deletions is limited by the length of the lexeme covering the position, and thus nearly constant.

However, the training phase is “embarrassingly parallel” [35] and linearly scalable with the number of cores as each character in the training set can be analyzed independently. For our evaluation, training took between 1 minute (for Tiny-C) and 15 hours (for gcc) on a compute server with 60 cores. Table 3 shows the training times for all SUTs, rounded to the next full respective time unit. SUTs that use wide character sets (e.g., UTF-8) can still cause scalability problems, similar to the situation in standard regex learning. We can mitigate these by several optimizations that trade accuracy for speed, e.g., checking pairwise character replaceability across the training set to identify character classes before training. We can also use an incremental approach, were occurrences of keywords already learned from earlier examples are no longer mutated in later examples. However, we leave these optimizations for future work.

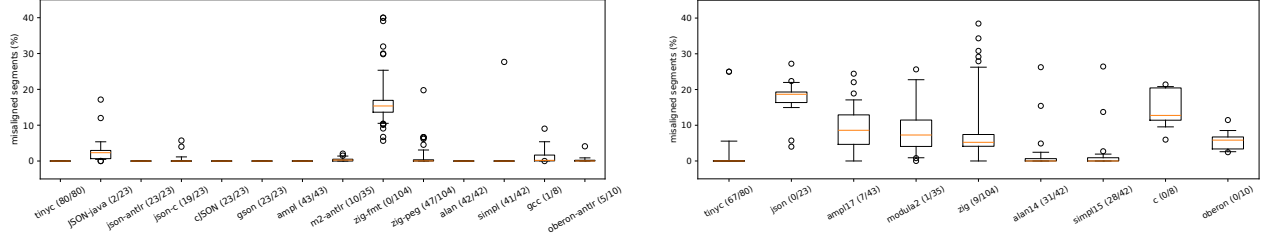


Figure 3: Segmentations produced by inferred (left) and generic (right) lexers.

The Y-axis shows the relative size (in characters) of the misaligned segments, the X-axis shows the SUTs and the respective numbers of perfectly lexed files over the sizes of the validation sets. Boxes represent the Q3/Q1 interquartile range, whiskers the 95/5 percentiles and circles outliers. The median is shown as horizontal line through the boxes.

The actual inference is much faster and we observed inference times from around 10 seconds on a MacBook Pro M4 for the smaller examples (JSON, AMPL/ALAN/SIMPL) to about 20 minutes (gcc). Table 3 again has the detailed numbers. The limiting factor here is the number of iterations required until rewriting reaches a fixpoint.

Threats to validity. Given the dependency on the training set, and the limitations outlined above, the results reported here may not generalize to other training sets, SUTs, or target languages.

7 Related Work

We are not aware of any work that directly shares our goal of learning the lexical structure of black-box systems. Perhaps closest in spirit to but entirely different in approach from our work are two white-box approaches to infer tokens for use as fuzzer dictionaries [33, 41]. Mathis et al. [33] use dynamic taint analysis to track and extract tokens from a SUT for more efficient fuzzing. Shastry et al. [41] statically infer dictionaries from source code in order to improve fuzzing.

We also draw on related work from grammatical inference. Approaches here often follow Gold’s model of *identification in the limit* [17] (e.g., the classic RPNI algorithm [10]), or Angluin’s *query model* [3]. We follow the query model, where the learner can ask the teacher membership and equivalence queries. Membership queries fit well with our approach since we assume that the SUT will either accept or reject a mutation. We do not need equivalence queries because we are not trying to learn the full CFG.

Most complete grammar learning algorithms work for regular languages only where all necessary properties such as equivalence are decidable. However, as already outlined in Section 1, RE learning algorithms like Angluin’s L^* [3] assume that the training examples are complete sentences of a regular target language, i.e., they need to be given the individual lexemes to learn the corresponding token structure. They are thus not applicable in our work.

Learning a full CFG can work from a token stream, so our approach can be seen as a first step towards this. CFG learning is computationally intractable [4], even though some restricted subclasses remain tractable [9, 22, 48]. Given the theoretical restrictions, more recent practical tools are based on heuristics, and we follow similar lines.

White- and gray-box tools like AUTOGram [20, 21] use dynamic tainting to learn a well-labeled grammar, while some approaches reconstruct grammars from execution traces [31]. Mimid [19] maps control-flow structure to the CFG structure, but assumes recursive-descent parser implementation. Symbolic execution [7] has also been used to reconstruct grammars without the need of training examples.

Recent black-box approaches show promising results. Glade [6] uses a two-phase method with regex and CFG generalization. Arvada [27] and Treevada’s [5] bubbling and merging operations improve performance on deeply nested structures. The most recent Kedavra [30] demonstrates the benefits of a generic tokenizer for CFG learning.

8 Conclusion and Future Work

Conclusion. In this paper, we described a new approach to learn the full lexical structure of black-box systems from a set of positive training examples. It does not try to learn arbitrary automata but instantiates RE meta-patterns based on SUT feedback during parsing the systematically modified examples. However, while our approach instantiates the patterns and simultaneously segments the training data according to the instantiated patterns, the segmentation is ancillary to the instantiation, and not a goal of the inference in itself.

We have evaluated our approach on several target languages that are input languages for different SUTs. We showed that (i) the learned tokens are similar, and in many cases identical, to those of the oracle lexers; (ii) we also identify bugs or inconsistencies where the SUTs does not correctly implement the lexical structure of their respective target languages; and (iii) as a proof-of-concept evaluation, the learned tokens improve the quality of grammars learned from a SOTA grammar inference tool.

Future Work. In addition to extending our experimental evaluation to more SUTs of different languages and addressing implementation limitations (including those sketched in Section 6), we plan to better integrate with SOTA grammar learning approaches. This, we believe, will enable these learners to scale to languages beyond the complexity of JSON. We will also borrow ideas from “active learning” techniques, where we only accept a generalization if it does not induce words outside the target language. We will also explore other downstream applications, such as token-level fuzzing [39].

Data Availability Statement

A replication package containing our implementation and experimental data is available on Zenodo [38]. Sources and updates are also available at <https://github.com/gtoolsuite/lexinfer>.

Acknowledgments

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under the project IdeFix (496588242) and the National Research Foundation (NRF) under grant SRUG-2204234463.

References

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, & Tools* (2nd ed.). Addison-Wesley.
- [2] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6
- [3] Dana Angluin. 1987. Queries and Concept Learning. *Mach. Learn.* 2, 4 (1987), 319–342. doi:10.1007/BF00116828
- [4] Dana Angluin. 1990. Negative Results for Equivalence Queries. *Mach. Learn.* 5 (1990), 121–150. doi:10.1007/BF00116034
- [5] Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. 2024. Fast Deterministic Black-box Context-free Grammar Inference. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 117:1–117:12. doi:10.1145/3597503.3639214
- [6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. doi:10.1145/3062341.3062349
- [7] Leon Bettscheider and Andreas Zeller. 2024. Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15–19, 2024*, Marcelo d’Amorim (Ed.). ACM, 522–526. doi:10.1145/3663529.3663790
- [8] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. <https://www.rfc-editor.org/rfc/rfc8259>. doi:10.17487/RFC8259 RFC 8259.
- [9] Alexander Clark. 2010. Distributional Learning of Some Context-Free Languages with a Minimally Adequate Teacher. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13–16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6339)*, José M. Sempere and Pedro García (Eds.). Springer, 24–37. doi:10.1007/978-3-642-15488-1_4
- [10] Alexander Clark, Rémi Eyraud, and Amaury Habrard. 2008. A Polynomial Algorithm for the Inference of Context Free Languages. In *Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI 2008, Saint-Malo, France, September 22–24, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5278)*, Alexander Clark, François Coste, and Laurent Miclet (Eds.). Springer, 29–42. doi:10.1007/978-3-540-88009-7_3
- [11] ANTLR contributors. 2012. grammars-v4. <https://github.com/antlr/grammars-v4>
- [12] Tom Everett. 2016. m2pim4.g4 ANTLR grammar. <https://github.com/antlr/grammars-v4/blob/master/modula2pim4/m2pim4.g4>
- [13] Tom Everett. 2020. ANTLR grammar for Oberon. <https://github.com/antlr/grammars-v4/blob/master/oberon/oberon.g4>
- [14] Marc Feely. 2002. tinycc: Tiny C Compiler Source File. <https://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinycc>. <https://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinycc> University of Montreal course material, accessed on 2025-05-22.
- [15] Dave Gamble. 2009. cJSON: Ultralightweight JSON parser in ANSI C. <https://github.com/google/gson>. <https://github.com/google/gson> Accessed: 2025-06-15.
- [16] Pedro García, Manuel Vázquez de Parga, Gloria Inés Álvarez, and José Ruiz. 2008. Learning Regular Languages Using Nondeterministic Finite Automata. In *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21–24, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5148)*, Oscar H. Ibarra and Bala Ravikumar (Eds.). Springer, 92–101. doi:10.1007/978-3-540-70844-5_10
- [17] E. Mark Gold. 1967. Language Identification in the Limit. *Inf. Control.* 10, 5 (1967), 447–474. doi:10.1016/S0019-9958(67)91165-5
- [18] Google. 2008. gson. <https://github.com/google/gson>. <https://github.com/google/gson> Accessed: 2025-05-22.
- [19] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8–13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. doi:10.1145/3368089.3409679
- [20] Matthias Hörschle and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 720–725. doi:10.1145/2970276.2970321
- [21] Matthias Hörschle and Andreas Zeller. 2017. Mining input grammars with AUTOGram. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 31–34. doi:10.1109/ICSE-C.2017.14
- [22] Hiroki Ishizaka. 1990. Polynomial time Learnability of Simple Deterministic Languages. *Mach. Learn.* 5 (1990), 151–164. doi:10.1007/BF00116035
- [23] JSON java contributors. 2010. json-java. <https://github.com/stleary/JSON-java>. <https://github.com/stleary/JSON-java> Accessed: 2025-06-15.
- [24] json-c contributors. 2012. json-c. <https://github.com/json-c/json-c/wiki>. <https://github.com/json-c/json-c/wiki> Accessed: 2025-06-15.
- [25] Andrew Kelley and Zig contributors. 2015. Zig Language Reference. <https://ziglang.org/documentation/master/>. <https://ziglang.org/documentation/master/> Accessed: 2025-07-14.
- [26] Bruce Knobe and Kathleen Knobe. 1976. A Method for Inferring Context-free Grammars. *Inf. Control.* 31, 2 (1976), 129–146. doi:10.1016/S0019-9958(76)80003-4
- [27] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15–19, 2021*. IEEE, 456–467. doi:10.1109/ASE51524.2021.9678879
- [28] Ralf Lämmel and Chris Verhoef. 2001. Semi-automatic grammar recovery. *Softw. Pract. Exp.* 31, 15 (2001), 1395–1438. doi:10.1002/SPE.423
- [29] Ralf Lämmel and Vadim Zaytsev. 2011. Recovering grammar relationships for the Java Language Specification. *Softw. Qual. J.* 19, 2 (2011), 333–378. doi:10.1007/S11219-010-9116-5
- [30] Feifei Li, Xiao Chen, Xi Xiao, Xiaoyu Sun, Chuan Chen, Shaohua Wang, and Jitao Han. 2024. Incremental Context-free Grammar Inference in Black Box Settings. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 – November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1171–1182. doi:10.1145/3691620.3695494
- [31] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9–14, 2008*, Mary Jean Harrold and Gail C. Murphy (Eds.). ACM, 83–93. doi:10.1145/1453101.1453114
- [32] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Hörschle, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 548–560. doi:10.1145/3314221.3314651
- [33] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18–22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 27–37. doi:10.1145/3395363.3397348
- [34] Pascal Michel. 2022. Modula-2 Examples. <https://github.com/michelou/m2-examples>. Accessed: 2025-07-19.
- [35] Clive Moler. 1986. Matrix computation on distributed memory multiprocessors. In *Hypercube Multiprocessors*, Michael T. Heath (Ed.). Society for Industrial and Applied Mathematics, 181–195.
- [36] Terrence Parr. 2025. json.g4 ANTLR grammar. <https://github.com/antlr/grammars-v4/tree/master/json>. <https://github.com/antlr/grammars-v4/tree/master/json> Accessed: 2025-07-14.
- [37] Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicted-LL(k) Parser Generator. *Softw. Pract. Exp.* 25, 7 (1995), 789–810. doi:10.1002/SPE.4380250705
- [38] Moeketsi Raselimo, Lars Grunske, and Bernd Fischer. 2026. *Learning the Lexical Structure of Black-Box Systems by Parsing Systematic String Edit Mutations*. doi:10.5281/zenodo.21722193
- [39] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11–13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2795–2809. <https://www.usenix.org/conference/usenixsecurity21/presentation/salls>
- [40] Burr Settles. 2012. *Active Learning*. Springer. doi:10.1007/978-3-031-01560-1
- [41] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017. Proceedings (Lecture Notes in Computer Science, Vol. 10453)*, Marc Dacier, Michael D. Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer, 26–47. doi:10.1007/978-3-319-66332-6_2
- [42] Ray J. Solomonoff. 1959. A new method for discovering the grammars of phrase structure languages. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15–20 June 1959*. UNESCO (Paris), 285–289.
- [43] M. G. J. Van Den Brand, M. P. A. Sellink, and C. Verhoef. 1997. Obtaining a COBOL grammar from legacy code for reengineering purposes. In *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications (Amsterdam, The Netherlands)*. BCS Learning & Development Ltd., Swindon, GBR, 1–15. doi:10.14236/ewic/AL1997.6
- [44] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-based testing for little languages: an experience report with student compilers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16–17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 253–269. doi:10.1145/3426425.3426946

- [45] vishapoberon. 2024. Oberon by Example. <https://github.com/vishapoberon/oberonbyexample>. Accessed: 2026-01-29.
- [46] Niklaus Wirth. 1988. *Programming in Modula-2, 4th Edition*. Springer. doi:10.1007/978-3-642-83565-0
- [47] Niklaus Wirth. 1988. The Programming Language Oberon. *Softw. Pract. Exp.* 18, 7 (1988), 671–690. doi:10.1002/SPE.4380180707
- [48] Ryo Yoshinaka. 2006. Polynomial-Time Identification of an Extension of Very Simple Grammars from Positive Data. In *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4201)*, Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita (Eds.). Springer, 45–58. doi:10.1007/11872436_5
- [49] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISA Helmholtz Center for Information Security. <https://www.fuzzingbook.org/> Retrieved 2024-07-01 16:50:18+02:00.
- [50] Zig Community Contributors. 2023. Zig Cookbook. <https://github.com/zigcc/zig-cookbook>. Accessed: 2025-07-19.
- [51] Zig Language Contributors. 2024. Zig Language Specification: Grammar. <https://github.com/ziglang/zig-spec/tree/master/grammar>. Accessed: 2025-07-19.