Static Analysis for Efficient Streaming Tokenization

Angela W. Li
awl@rice.edu
Rice University
Houston, Texas, USA

Yudi Yang
yudi.yang@rice.edu
Rice University
Houston, Texas, USA

Konstantinos Mamouras
mamouras@rice.edu
Rice University
Houston, Texas, USA

Abstract

Tokenization, also referred to as lexing or scanning, is the computational task of partitioning an input text into a sequence of substrings called tokens. Tokenization is one of the first stages of program compilation, it is used in natural language processing, and it is also useful for processing unstructured text or semi-structured data such as JSON, CSV, and XML. A tokenizer is typically specified as a list of regular expressions, which is called a tokenization grammar. Each regular expression describes a class of tokens (e.g., integer, floating-point number, variable identifier, string literal). The semantics of tokenization employs the longest match policy to disambiguate among the possible choices. This policy says that we should prefer a longer token over a shorter one. It is also known as the maximal munch policy.

Tokenization is an important computational task when processing semi-structured data, as it often precedes parsing, querying, or data transformations. Due to the abundance of large-scale semi-structured data, which can be too large to load in memory, it is desirable to perform tokenization in a streaming fashion with a small memory footprint. First, we observe that some tokenization grammars are inherently more difficult to deal with than others, and we provide a static analysis algorithm for recognizing them. We continue to propose the StreamTok algorithm, which relies on this analysis to enable efficient tokenization. StreamTok is asymptotically better than the standard algorithm of flex. Our experimental results show that our implementation of StreamTok outperforms state-of-the-art tools for tokenization.

CCS Concepts: • **Theory of computation** → **Formal languages and automata theory**; **Regular languages**; • **Software and its engineering** → **Compilers**; **Parsers**.

Keywords: lexical analysis, lexing, scanning, tokenization, maximal munch semantics, backtracking, data streams

ACM Reference Format:
Angela W. Li, Yudi Yang, and Konstantinos Mamouras. 2026. Static Analysis for Efficient Streaming Tokenization. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790227

1 Introduction

Many modern data-driven applications rely on the processing of massive amounts of unstructured (e.g., text corpora and system logs) or semi-structured data, such as JSON, YAML, XML, and CSV. *Tokenization*, also known as scanning or lexing, is often the first step in processing unstructured and semi-structured data. Tokenization is the splitting of a string into a sequence of tokens. The rules that specify how the string should be split into tokens are typically given by a list of regular expressions.

Many applications consume streaming data, which is generated in real time, and need to process and respond with low latency. Streaming applications have to deal with large streams that cannot be stored in system memory. For this reason, stream processing requires the development of techniques that have a small memory footprint. There are situations where the data is available offline (e.g., on a hard disk), but is too large to load in the system memory. Streaming techniques are also valuable in such situations because the data can be read and processed block-by-block without loading the entire dataset at once.

Here, we focus on the computational problem of tokenization in the streaming model. Our techniques can benefit applications that deal with streaming or static (but too large to load in memory) unstructured and semi-structured data.

A tokenizer (scanner or lexer), is a program that tokenizes a given input string. Such a program can be handcrafted or automatically generated by tools commonly referred to as lexer generators. In particular, lexer generators take as input user-defined grammars and output a lexer that will perform tokenization according to the supplied user-defined grammar. Some popular lexer generators used in practice include flex [47], JFlex [23] and Ocamllex [43].

A lexer generator provides useful flexibility compared to a handcrafted implementation for a fixed grammar. Some grammars are often adapted by users. E.g., CSV/TSV grammars can vary based on how we delimit fields, how we delimit records, and how we use escaping in fields. Changing a tokenizer grammar is a lot easier than changing a handcrafted implementation of a tokenizer. Even when a grammar is standardized, adaptation may be useful for a specific application. For example, JSON minification (removing unnecessary whitespace) can be performed with a simplified (and more efficient) lexical grammar that identifies whitespace. Moreover, some applications (e.g., log parsing) require the continual addition of new formats, which are conveniently handled with grammars and lexer generators. Finally, there are cases where we may want to adapt a grammar based on runtime information. E.g., a CSV parsing tool can use *schema information* (given at runtime as a command-line argument) to adapt the grammar for recognizing the types of the fields.

The aforementioned lexer generator tools implement the maximal-munch disambiguation policy, where longer tokens are preferred over shorter tokens. When there is a tie, the earliest tokenization rule is preferred. We formulate our tokenization problem (§2) according to this policy. It is known that the standard backtracking-based tokenization algorithm, which these tools implement, has a worst-case time complexity of $O(n^2)$, where n is the size of the input text.

Tokenization is often a preprocessing step for parsing and AST generation, but it has more uses. Tokenization can perform simple transformations on data and reduce data volume. E.g., to process a specific column in a streaming CSV file, we can first extract the desired column through tokenization before propagating the reduced data to the next stage of the pipeline. Tokenization enables simple queries and aggregations on (streaming) data, such as counting the number of numeric fields in a JSON file. Performing queries directly over the token stream (i.e., without full parsing) is valuable because processing streaming data is challenging due to latency and memory constraints.

Efficient tokenization over streaming data presents a challenge. As we show in §2, streaming tokenization requires (for some grammars) a memory footprint that is linear in the stream length. We can, however, restrict our attention to a class of tokenization grammars that admit low-memory streaming tokenization. This class contains many grammars used in practice. Moreover, they can be identified by a static analysis[1] (see §4). For this class of tokenization grammars, we propose a time- and space-efficient tokenization algorithm that is appropriate in the streaming setting (see §5).

Main Contributions:

(1) We consider maximal-munch tokenization in the streaming model and show that it requires (in the worst case) space that is proportional to the input stream.
(2) We introduce the novel notion of *(maximum) token neighbor distance*. This semantic notion is used to indentify the tokenization grammars to which our efficient streaming tokenization algorithm can be applied. We observe that this notion also sheds light on the worst-case performance of backtracking-based tokenization.
(3) We show that computing the maximum token neighbor distance for a tokenization grammar is PSPACE-complete (Theorems 13, 14). We also propose a static analysis that takes as input a tokenization grammar and outputs its maximum token neighbor distance (Theorem 15).
(4) We propose an efficient streaming tokenization algorithm (*StreamTok*) applicable to the class of tokenization grammars with bounded maximum token neighbor distance.
(5) We collect tokenization grammars for data exchange formats and a dataset from GitHub. Our static analysis sheds some light on the complexity of grammars used in practice. We find that our streaming tokenization algorithm applies to a significant portion of these grammars.
(6) Our experimental evaluation shows that the proposed streaming tokenization algorithm offers a performance benefit, namely a 2× to 3× speedup, over existing tokenization tools such as flex [47]. The memory footprint of our algorithm is in the order of kilobytes, regardless of the length of the input stream (which could be infinite).
(7) We consider higher-level applications (log parsing, format conversions, and data validation) that use tokenization and show the performance benefit of our approach.

[1]We use the term *static analysis* because it applies to the tokenization grammar *before* execution and is independent of the input text.

2 The Streaming Tokenization Problem

We start this section by introducing the tokenization problem. Then, we discuss tokenization in the streaming model and present a relevant complexity result.

Let Σ be a finite alphabet of symbols (letters, characters). A unary predicate $\sigma \subseteq \Sigma$ is called a *character class*. The set $\mathrm{Reg}(\Sigma)$ of *regular expressions* (regexes) is defined by the grammar $r, r_1, r_2 ::= \varepsilon \mid \sigma \mid (r_1 \mid r_2) \mid r_1 \cdot r_2 \mid r^*$. Concatenation is also written as $r_1 r_2$ to reduce notational clutter. The notation r^+ ("repetition of r at least once") is an abbreviation for rr^*. The notation $r?$ is an abbreviation for $r \mid \varepsilon$. For a regular expression r, the notation r^n is an abbreviation for the concatenation $r \cdot r \cdots r$ (n times). The notation $r\{n\}$ is also commonly used to describe the repetition of r exactly n times. More generally, we write $r\{m, n\} = r^m (r?)^{n-m}$ to denote the repetition of r from m to n times. Following PCRE conventions, we also use notation for character classes. For instance, the regular expression $[abc]$ accepts a character a, b, or c, which is equivalent to the regular expressions $[a\text{-}c]$ and $a \mid b \mid c$. Negation can be used inside character classes. For example, the regex $[\hat{}abc]$ matches any character that is *not* a, b, or c. Every regular expression r denotes a language $\mathcal{L}(r) \subseteq \Sigma^*$, defined as usual.

We write $|w|$ to denote the length of a string w. The empty string (i.e., the string of length 0) is denoted by ε. For a string $w \in \Sigma^*$, we will call a pair $[i, j]$ with $0 \leq i \leq j \leq |w|$ a *location* in w. A *position* in $|w|$ is an index in the range $0, 1, \ldots, |w|$. We write $w[i..j]$ for the substring of w at location $[i, j]$. E.g., for the string $w = abbcabab$ (length $|w| = 8$), we have that $w[0..3] = abb$, $w[1..5] = bbca$, $w[4..7] = aba$, and $w[5..8] = bab$. We also use the abbreviations $w[..i] = w[0..i]$ and $w[i..] = w[i..|w|]$.

Let $u, v \in \Sigma^*$. We say that u is a *prefix* of v, and we write $u \leq v$, if there is a string $w \in \Sigma^*$ such that $uw = v$. When $u \leq v$, we also say that v is an *extension* of u. The prefix relation $\leq$ is a partial order (i.e., reflexive, antisymmetric, and transitive). The empty string ε is the least element of the prefix relation. We write $u < v$ to denote that $u \leq v$ and $u \neq v$, and we say that u is a *strict prefix* of v (and that v is a *strict extension* of u). When $u \leq v$, we write $u^{-1}v$ for the unique string such that $u \cdot (u^{-1}v) = v$.

Definition 1 (Tokenization). Let $\bar{r} = [r_0, r_1, \ldots, r_{\kappa-1}]$ be a nonempty sequence of regular expressions. We call each r_β a *(tokenization) rule* and $\bar{r}$ a *tokenization grammar.*

We write $\mathbb{T} = \mathbb{N}$ for the set of all token ids. We define $\text{token}(\bar{r}) : \Sigma^* \to \text{Option}(\Sigma^+ \times \mathbb{T})$ as follows: $\text{token}(\bar{r})(u) = \text{Some}(v, \beta)$, where v is the longest nonempty prefix of u that matches some token regex and $\beta \in \{0, \ldots, \kappa - 1\}$ is the least index such that $v \in \mathcal{L}(r_\beta)$. We prefer the rule with the least index if there are several rules that match the longest token.

We also define $\text{tokens}(\bar{r}) : \Sigma^* \to \text{List}(\Sigma^+ \times \mathbb{T})$ as follows:

$$\text{tokens}(\bar{r})(u) = [\,],\ \text{if}\ \text{token}(\bar{r})(u) = \text{None}$$

$$\text{tokens}(\bar{r})(u) = [(v, \beta)] \cdot \text{tokens}(\bar{r})(v^{-1}u),$$

if $\text{token}(\bar{r})(u) = \text{Some}(v, \beta)$, where $v^{-1}u$ is the result of removing the prefix v from u.

The *tokenization problem* is the following: Given a tokenization grammar $\bar{r}$ and a string $w \in \Sigma^*$ as input, compute the list $\text{tokens}(\bar{r})(w)$. For a fixed tokenization grammar $\bar{r}$, the *$\bar{r}$-tokenization problem* is the following: Given an input string $w \in \Sigma^*$ as input, compute the list of tokens $\text{tokens}(\bar{r})(w)$.

Example 2 (Tokenization). Consider the tokenization grammar $\bar{r} = [a, ba^*, c[ab]^*]$ and the text $w = abaabacabaa$. Then, $\text{token}(\bar{r})(w) = \text{Some}(a, 0)$ because the string a matches the regular expression a, and no other strings starting at the beginning of w match any of the three rules. Then, we notice that $\text{token}(\bar{r})(w[1..]) = \text{Some}(baa, 1)$, $\text{token}(\bar{r})(w[4..]) = \text{Some}(ba, 1)$. Finally, $\text{token}(\bar{r})(w[6..]) = \text{Some}(cabaa, 2)$ because the entire remaining part of the string can be matched for the regex $c[ab]^*$, which is the longest match possible. So, $\text{tokens}(\bar{r})(w) = [(a, 0), (baa, 1), (ba, 1), (cabaa, 2)]$. The entire input string is tokenized.

Let $\bar{r} = [r_0, r_1, \ldots, r_{\kappa-1}]$ be a tokenization grammar. We sometimes represent $\bar{r}$ as a single regular expression $r_0 \mid r_1 \mid \cdots \mid r_{\kappa-1}$, where the top-level operator is κ-ary nondeterministic choice. E.g., the grammar [`a`, `a*b`, `[ab]*[^ab]`] with three token rules is represented as the regex `a|a*b|[ab]*[^ab]`.

Definition 3 (Tokenization DFA). A DFA over the alphabet Σ is a tuple $\mathcal{A} = (Q, \delta, q_{init}, F)$, where Q is the finite set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_{init} \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states.

Let $p, q \in Q$. If $q = \delta(p, a)$, then we write $p \to^a q$. We define the function $\delta : Q \times \Sigma^* \to Q$ as follows: $\delta(q, \varepsilon) = q$ and $\delta(q, ua) = \delta(\delta(q, u), a)$ for $u \in \Sigma^*$ and $a \in \Sigma$. We also define $\delta : \Sigma^* \to Q$ as $\delta(u) = \delta(q_{init}, u)$. For a string $u \in \Sigma^*$, the notation $p \to^u q$ indicates that $\delta(p, u) = q$.

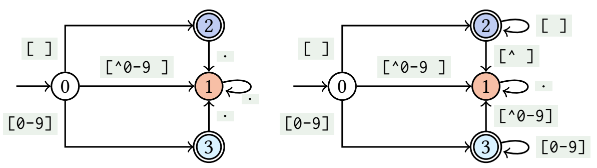


Figure 1. DFAs for the grammars `[0-9]|[ ]` and `[0-9]+|[ ]+`.

A *tokenization DFA* is equipped with a function $\Lambda : F \to \mathbb{T}$, where $\Lambda(q)$ is the preferred token id for a final state $q \in F$.

For a state q of a DFA, we say that q is a *reject* or *failure* state if it cannot reach a final state. During DFA execution, reaching a reject/failure state means that no extension of the input string can lead to acceptance.

Example 4. Fig. 1 (left) shows the tokenization DFA for the grammar `[0-9]|[ ]`. Throughout this paper, the states of the automata will be colored based on their associated grammar rule. In this example, state 2 corresponds to the rule `[]` and is assigned a dark blue color, while state 3 corresponds to the rule `[0-9]` and is assigned a lighter blue color. State 1 is the reject state and is assigned an orange color. The starting state 0 is not associated with any grammar rule and is uncolored.

From the initial state 0, if we receive a character that is neither a digit nor a space, then the automaton reaches the rejecting state 1 since it is impossible to form a token with the received character. A space character leads to the final state 2, which corresponds to the rule `[ ]`. From 2, receiving any character will cause a transition to the rejecting state 1. Similarly, receiving a digit at the initial state 0 leads to the final state 3, which corresponds to the rule `[0-9]`.

Example 5. We consider the grammar `[0-9]+|[ ]+`. The tokenization DFA for this grammar is shown in Fig. 1 (right). This is similar to the previous example. However, from state 2, we have a self-loop labeled with the space character, which reflects the repetition operator `+` in the rule `[0-9]+`. However, as before, upon receiving a non-space character in state 2, the automaton will reach the rejecting state 1. A similar change is also seen in state 3.

Tokenization in the Streaming Model. In the streaming model, the input is a potentially unbounded sequence of items that are revealed gradually during the computation instead of becoming available all at once at the beginning. Each input item is read only once as it arrives. While we can store input items seen in the past, storing the entire stream using $O(n)$ space (where n is the size of the input stream) is highly undesirable. Moreover, in streaming tokenization, we would like to emit each token as early as possible as we consume the input stream. However, this is not easy according to the maximal-munch semantics where we prefer longer tokens; it is possible that the algorithm "waits" indefinitely without an output in search of a longer token. In Lemma 6 below, we prove a space lower bound for any generic streaming tokenization algorithm and demonstrate the inherent difficulty of the tokenization problem in the streaming model.

Lemma 6 (Space Lower Bound). There is a tokenization grammar $\bar{r}$ for which the streaming tokenization problem requires $\Omega(n)$ space, where n is the length of the input stream.

Proof. Consider the three-letter alphabet $\Sigma = \{a, b, c\}$ and the tokenization grammar $\bar{r} = [a, b, (a \mid b)^* c]$. We focus on streams that consist of only a's and b's. In this case, the algorithm cannot emit any output until the end-of-stream symbol is encountered because the rule $(a \mid b)^* c$ will match if the final letter is c. So, the algorithm has to store the entire stream, in order to correctly produce the output in the case when the stream ends without the letter c. □

Specifying Tokenizers. For our formulation of the tokenization problem, we chose to specify tokenizers as lists of rules that are regular expressions. Alternatively, tokenizers could be specified using DFAs or NFAs. Our syntax of regexes uses only the classical constructs (choice, concatenation, and Kleene star). Regexes used in practice often use constructs that make them more succinct (e.g., bounded repetition [25, 26] and lookaround assertions [8, 31]).

3 Token Neighbor Distance

This section starts by introducing novel semantic concepts that capture the challenges in efficient streaming tokenization. In particular, the notion of *maximum token neighbor distance* tells us how many more input characters we need to read in order to confirm whether a token is maximal or not. Since this is semantically defined (in particular, without reference to a specific tokenization algorithm), it captures an aspect that is inherent in the tokenization grammar. The concepts introduced in this section are used in our static analysis of §4, which in turn enables our efficient streaming tokenization algorithms of §5.

In §3.2, we note that this semantic notion also tells us something valuable about the standard backtracking tokenization algorithm used by many popular lexer-generator tools such as flex [47]. It gives us a bound on the amount of backtracking that the algorithm performs, which in turn places a bound on the running time. In particular, Lemma 12 states that if the maximum token neighbor distance is bounded by k, then the standard backtracking algorithm has running time $O(k \cdot n)$ (in general, the worst-case time complexity is $O(n^2)$).

3.1 Token Neighbor Distance

Let $S \subseteq \mathbb{N}$ be a subset of natural numbers. A number $M \in \mathbb{N}$ is said to be an *upper bound* of S if $n \leq M$ for every $n \in S$. The subset S is called *bounded* if it has some upper bound. It is called *unbounded* if it is not bounded. That is, S is unbounded iff for every $M \in \mathbb{N}$ there exists some $n \in \mathbb{N}$ such that $n > M$.

Definition 7 (Token Neighbor Distance). Let $L \subseteq \Sigma^*$. We will define the *token neighbor relation* $\rightarrowtail \subseteq \Sigma^* \times \Sigma^*$ w.r.t. the language L. We define $u \rightarrowtail v$ iff the following hold:

1. both u and v belong to L and are nonempty,
2. $u \leq v$ (i.e., u is a prefix of v), and
3. $w \notin L$ for every w with $u < w < v$ (i.e., for every w that is a strict extension of u and a strict prefix of v).

A pair $(u, v) \in \rightarrowtail$ is called a *token neighbor pair*. We also say that u, v are neighboring tokens, and we define the *distance* from u to v to be the length of $u^{-1}v$. That is, we define

$$\mathsf{TkDist}(u, v) = |u^{-1}v|$$

for u, v with $u \rightarrowtail v$. The string $u^{-1}v$ is called the *token increment* from u to v. We also define

$$\begin{aligned} \mathsf{TkDist}(L) &= \sup \mathsf{DSet}(L), \text{ where} \\ \mathsf{DSet}(L) &= \{\mathsf{TkDist}(u, v) \mid u \rightarrowtail v\} \text{ and} \end{aligned}$$

sup is the supremum operator. In particular, the definition says that $\mathsf{TkDist}(L) = \infty$ when the set $\mathsf{DSet}(L)$ of token neighbor distances is unbounded. We call $\mathsf{TkDist}(L)$ the *maximum token neighbor distance* for L.

Intuition: $u \rightarrowtail v$ means that u, v are tokens (w.r.t. the language L) so that u is a prefix of v and there is no other token that is strictly between them (w.r.t. the prefix order).

Abbreviations: We sometimes write TND as abbreviation for "token neighbor distance". We also write max-TND to abbreviate "maximum TND".

Definition 8 (Maximum TND). Let $\bar{r}$ be a tokenization grammar. We define $\mathsf{TkDist}(\bar{r}) = \mathsf{TkDist}(\mathcal{L}(\bar{r}))$.

Example 9. In the table below, we list some tokenization grammars and their maximum token neighbor distance.

	grammar $\bar{r}$	$\mathsf{TkDist}(\bar{r})$
1.	`[0-9]|[ ]`	0
2.	`[0-9]+|[ ]+`	1
3.	`[0-9]+(\.[0-9]+)?|[ \.]`	2
4.	`[0-9]+([eE][+-]?[0-9]+)?|[ ]+`	3
5.	`[0-9]*0|[ ]+`	∞
6.	`a|a*b|[ab]*[^ab]`	∞

For the first grammar, every token has length 1. This means that $u \rightarrowtail v$ implies $u = v$ for all u, v. It follows that the max-TND is 0.

For the second grammar, the token neighbors (u, v) can be of only two forms: (i) u matches `[0-9]+` and $v = ux$ with x being a decimal digit, or (ii) u matches `[ ]+` and $v = ux$ with x being a space symbol. So, the max-TND is 1.

For the third grammar, we observe that the token neighbors (u, v) with the longest increment $u^{-1}v$ are of the following form: u matches `[0-9]+` and $v = u\,.\,x$ with x being a decimal digit (e.g., `9` $\rightarrowtail$ `9.9`). So, the max-TND is 2.

For the fourth grammar, the token neighbors (u, v) with the longest increment satisfy: u matches `[0-9]+` and $v =$

```
   // Input: grammar $\bar{r} = [r_0, \ldots, r_{\kappa-1}]$ and stream $text$
1  Function Tokenize($\bar{r}$ : List(Reg($\Sigma$)), $text$ : Stream($\Sigma$)):
2      $\mathcal{A} \leftarrow$ Tokenization DFA for $\bar{r}$
3      $\mathbb{N}$ $startP \leftarrow 0$   // start position for token
4      while $startP < text$.len do // pass with backtracking
5          Option($\mathbb{N} \times \mathbb{T}$) $tk \leftarrow$ None   // token found
6          $\mathbb{S}$ $q \leftarrow$ initial state of $\mathcal{A}$   // current DFA state
7          $\mathbb{N}$ $pos \leftarrow startP$   // current position in stream
8          while $pos < text$.len do // left-to-right pass
               // $\delta$: transition function of $\mathcal{A}$
9              $q \leftarrow \delta(q, text[pos])$
10             $pos \leftarrow pos + 1$   // move to the next symbol
11             if $q$ is final then
                   // $\Lambda(q)$ is the preferred token id
12                 $tk \leftarrow$ Some($pos - startP, \Lambda(q)$)
13             if $q$ is a failure state then break
14         if let Some($\ell, \beta$) = $tk$ then
               // emit token to the output stream
15             emit ($text[startP..startP + \ell], \beta$)
16             $startP \leftarrow startP + \ell$   // continue after token
17         else break
```

Figure 2. The standard DFA-based backtracking algorithm for tokenization (see flex [47]).

$uxyz$, where $x \in \{$`e`, `E`$\}$, $y \in \{$`+`, `-`$\}$, and z is a decimal digit. So, the max-TND is 3.

For the fifth grammar, note that `0` $\rightarrowtail$ `0` `1`^{i} `0` for every $i \geq 0$. It follows that the max-TND is ∞.

For the sixth grammar, we observe that `a` $\rightarrowtail$ `a` `a`^{i} `b` for every $i \geq 0$. So, the max-TND is ∞.

Lemma 10. Let $L \subseteq \Sigma^*$ and $k \in \mathbb{N}$. We have that $\mathsf{TkDist}(L) > k$ iff there exist u, v such that the following hold: (1) $u \rightarrowtail v$ (i.e., u and v are token neighbors w.r.t. L) and (2) $|u^{-1}v| > k$.

Lemma 11 (Dichotomy). Let $L \subseteq \Sigma^*$ be regular. Suppose that m is the number of states of the minimal DFA that recognizes L. Then, $\mathsf{TkDist}(L) = \infty$ or $\mathsf{TkDist}(L) \leq m + 1$.

3.2 Bounding the Backtracking Distance

In this subsection, we briefly explore how the notion of token neighbor distance may help us understand the performance of the standard backtracking-based tokenization algorithm. Fig. 2 shows the essence of this algorithm. The grammar is an input argument for the procedure `Tokenize`. The outer while loop (Line 4) performs one repetition for every token that is identified. The variable *startP* keeps track of the start position for the token that will be identified next. The inner while loop (Line 8) reads the input symbols from left to right, starting with position *startP* until the longest token for the suffix *text*[*startP*..] is found.

We say that the algorithm of Fig. 2 ***backtracks*** because the index *pos* that is used to read from the input string *text* can move backwards. This can happen when a maximal token is confirmed, which can trigger the execution of Line 16 and then Line 7. These statements can result in *pos* being decremented, which we refer to as "backtracking".

The possibility of backtracking means that the `Tokenize` algorithm of Fig. 2 can end up reading parts of the input several times. This is a well-known source of inefficiency. See, for example, [38]. In fact, the worst-case running time of this algorithm is $\Theta(n^2)$, where n is the length of the input text, for some tokenization grammars.

Lemma 12 below captures our observation that, if the maximum token neighbor distance is bounded, then the standard backtracking algorithm is guaranteed to run in linear time despite the worst-case quadratic time complexity in general.

Lemma 12. Let $\bar{r}$ be a tokenization grammar. Suppose that $\mathsf{TkDist}(\bar{r}) = k < \infty$. Then, the worst-case time complexity of the algorithm `Tokenize`$(\bar{r})$ (i.e., for fixed grammar $\bar{r}$) is $O(k \cdot n)$, where n is the length of the input text.

The proof of Lemma 12 establishes that $\mathsf{TkDist}(\bar{r}) \leq k$ implies that the algorithm of Fig. 2 backtracks by at most k positions on the input string. In particular, this means that $\mathsf{TkDist}(\bar{r}) \leq \infty$ implies a uniform bound on the amount of backtracking, which can be used to prove the $O(k \cdot n)$ upper bound for the running time of the algorithm.

4 Static Analysis

Before we present our static analysis that computes the maximum token neighbor distance, as defined in Section 3, we consider the complexity of the problem in Theorem 13 below.

Theorem 13 (Complexity Lower Bound). Let $\textsc{TokenDist}_1$ be the decision problem of checking whether a tokenization grammar $\bar{r}$ satisfies $\mathsf{TkDist}(\bar{r}) \leq 1$ (i.e., it has max-TND at most 1). $\textsc{TokenDist}_1$ is PSPACE-hard.

Proof. We describe a reduction f from the universality problem for regular expressions to the problem $\textsc{TokenDist}_1$. Let r be a regular expression over Σ. We will define the regular expression $f(r)$ over the alphabet $\Gamma = \Sigma \cup \{\square\}$. For the definition of $f(r)$ we distinguish cases, based on whether r accepts the empty string ε or not.

(i) Case $\varepsilon \notin \mathcal{L}(r)$: Define $f(r) = \square|\square\square\square$.
(ii) Case $\varepsilon \in \mathcal{L}(r)$: Define $f(r)$ to be the regex that accepts a string w if and only if (1) w is the empty string, or (2) w ends with $\square$, or (3) w ends with a symbol in Σ and $w|_\Sigma$ (remove all $\square$ occurrences) is in $\mathcal{L}(r)$. The construction of $f(r)$ can be done with a straightforward recursive algorithm that replaces each symbol a in r by $\square^* a \square^*$. The intuition for the construction is that the symbol $\square$ triggers acceptance, but in its absence we consider the original language (and ignore all $\square$ occurrences).

We will show now that r is universal iff $f(r) \in \textsc{TokenDist}_1$. First, suppose that r is universal. Then, case (ii) applies and $f(r)$ is universal. Therefore, $f(r)$ has max-TND at most 1. Suppose now that r is not universal, which means that there

```
// Input: tokenization grammar $\bar{r} = [r_0, \ldots, r_{\kappa-1}]$
Function AnalysisMaxTND($\bar{r}$ : List(Reg($\Sigma$))):
    $\mathcal{A} \leftarrow$ Tokenization DFA for $\bar{r}$
    $S \leftarrow \{q \mid q$ is final and $q = \delta(u)$ for some $u \in \Sigma^+\}$
    $CoAcc \leftarrow \{q \mid q$ is co-accessible, i.e., can reach a final state$\}$
    $dist \leftarrow 0$   // (tentative) max-TND
    while $dist < |\mathcal{A}| + 2$ do
        $T \leftarrow \{\delta(q, a) \mid q \in S$ and $a \in \Sigma\}$
        if $T \cap CoAcc = \emptyset$ then
            // $T$ contains no co-accessible state
            // maximum token neighbor distance = $dist$
            return $dist$
        $S \leftarrow \{q \in T \mid q$ is not final$\}$
        $dist \leftarrow dist + 1$
    return $\infty$
```

Figure 3. Static analysis: Computing the maximum token neighbor distance $\mathsf{TkDist}(\bar{r})$ for a tokenization grammar $\bar{r}$.

exists a string $x \in \Sigma^*$ such that $x \notin \mathcal{L}(r)$. If $x = \varepsilon$, then $f(r) = \square|\square\square\square$, which has max-TND 2. So, $f(r) \notin$ TokenDist$_1$. It remains to consider the case where x is non-empty, that is, $x = ya$ is in Σ^+. We claim that $f(r)$ has max-TND at least 2. Here is the witness: (1a) $u = y\square$ belongs to $\mathcal{L}(f(r))$ because it ends with $\square$, (1b) $v = y\square a\square$ belongs to $\mathcal{L}(f(r))$ because it ends with $\square$, (2) $u \leq v$, and (3) $y\square a$ does not belong to $\mathcal{L}(f(r))$ because $(y\square a)|_\Sigma = ya \notin \mathcal{L}(r)$. Since $|u^{-1}v| = 2$, we conclude that r has max-TND at least 2. □

With Theorem 13 we have established that the decision problem TokenDist$_1$ is PSPACE-hard. It follows that computing the max-TND, which is a more general problem, is computationally hard. We will proceed now to describe a DFA-based algorithm for computing the max-TND of a tokenization grammar.

Let $\mathcal{A}$ be a DFA. A state q is *accessible* if it is reachable from the initial state of $\mathcal{A}$. We say that q is *co-accessible* if there is a path from q to some final state of $\mathcal{A}$.

Fig. 3 shows our proposed algorithm to compute $\mathsf{TkDist}(\bar{r})$ for a tokenization grammar $\bar{r}$. Let $L = \mathcal{L}(\bar{r})$ and $\mathcal{A}$ be the tokenization DFA for $\bar{r}$. The main idea behind the algorithm is that it explores all paths in $\mathcal{A}$ that witness $\mathsf{TkDist}(L) \geq dist$ for increasing values of $dist$, i.e., $dist = 0, 1, \ldots, |\mathcal{A}| + 2$. We will continue to explain this idea.

Let $k \in \mathbb{N}$. Recall from Lemma 10 that $\mathsf{TkDist}(L) \geq k + 1$ iff there exist strings u, v such that $u \rightarrowtail v$ and $|u^{-1}v| \geq k + 1$. This means that v is of the form $v = u \cdot a_1 a_2 \ldots a_k a_{k+1} \cdot w$, where $w \in \Sigma^*$ and $a_i \in \Sigma$ for every i. The condition $u \rightarrowtail v$ says that $u \in L$, $v \in L$, and $ua_1 \ldots a_i \notin L$ for every $i \in \{1, 2, \ldots, k\}$. So, $\mathsf{TkDist}(L) \geq k + 1$ iff there exists a path

$$q_{init} \xrightarrow{u \in \Sigma^+} q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_k} q_k \xrightarrow{a_{k+1}} q_{k+1}$$

in $\mathcal{A}$ (where q_{init} is the initial state of $\mathcal{A}$) such that q is final, every q_i is non-final for $i = 1, 2, \ldots, k$, and q_{k+1} is co-accessible.

Theorem 14 (Complexity Upper Bound). Computing the maximum token neighbor distance for tokenization grammars can be done in (deterministic) polynomial space.

Proof. First, consider the problem of checking $\mathsf{TkDist}(\bar{r}) \geq k + 1$. Based on the earlier characterization in terms of the existence of a path in the tokenization DFA, a nondeterministic polynomial-space reachability algorithm (over the potentially exponentially large DFA) can determine the existence of such a path. By Savitch's theorem [41], there is a deterministic polynomial-space algorithm for this problem. Checking that $\mathsf{TkDist}(\bar{r}) = k + 1$ is the same as checking that $\mathsf{TkDist}(\bar{r}) \geq k + 1$ and $\mathsf{TkDist}(\bar{r}) \not\geq k + 2$. So, this can also be done in polynomial space. Finally, using Lemma 11 we see that computing $\mathsf{TkDist}(\bar{r})$ can be done in polynomial space. □

Theorem 15 (Correctness of Static Analysis). Let $\bar{r}$ be a tokenization grammar. The algorithm of Fig. 3 computes the maximum token neighbor distance $\mathsf{TkDist}(\bar{r})$.

Proof. Let $L = \mathcal{L}(\bar{r})$ be the language of the tokenization grammar and $\mathcal{A} = (Q, \delta, q_{init}, F)$ be the tokenization DFA for $\bar{r}$. For every $u \in \Sigma^*$, $u \in L$ iff $\delta(u) \in F$.

The correctness of the algorithm hinges on the following *loop invariant* for the main while loop (starting at Line 6):

1. $\mathsf{TkDist}(L) \geq dist$
2. $dist \leq |\mathcal{A}| + 2$
3. S contains a state q iff there exist a token $u \in L \cap \Sigma^+$ and a string $v \in \Sigma^{dist}$ such that $\delta(uv) = q$ and $w \notin L$ for every w with $u < w \leq uv$.

First, we have to show that the loop invariant holds when execution reaches the while loop. After Line 3 executes, S contains a state q iff there exists a token $u \in L \cap \Sigma^+$ such that $\delta(u) = q$. Before entering the loop, we also have that $dist = 0$. These two conditions imply Part (3) of the invariant (choose $v = \varepsilon$ and notice that the last part holds vacuously because there is no w with $u < w \leq uv$). Part (1) and Part (2) of the invariant are easily seen to hold because $dist = 0$.

Now, we continue to establish that the body of the loop preserves the loop invariant. After Line 7 executes, we have:

4. T contains a state q iff there exist a token $u \in L \cap \Sigma^+$, a string $v \in \Sigma^{dist}$ and a letter $a \in \Sigma$ such that $\delta(uva) = q$ and $w \notin L$ for every w with $u < w \leq uv$.

We will focus now on the test "$T \cap CoAcc = \emptyset$" that is performed in the conditional starting at Line 8. First, we examine the case $T \cap CoAcc \neq \emptyset$ where the test fails. This means that there is a state $q \in T \cap CoAcc$, i.e., $q \in T$ and q is co-accessible. From $q \in T$ and Property (4) we get that there exist $u \in L \cap \Sigma^+$, $v \in \Sigma^{dist}$ and $a \in \Sigma$ such that $\delta(uva) = q$ and $w \notin L$ for every w with $u < w \leq uv$. Since q is co-accessible, it follows that $\delta(q, z) \in F$ for some string $z \in \Sigma^*$ of minimal length (minimality here means that $\delta(q, z') \notin F$ for every strict prefix z' of z). So, $\delta(uvaz) = \delta(\delta(uva), z) = \delta(q, z) \in F$

***Example* 16**

dist	0	1	2	3
S	2 3 6	1 4	1 5	1
T	1 2 3 4 6	1 5 6	1 6	1
test	ff	ff	ff	tt

***Example* 17**

dist	0	1	2	···	6	7
S	2 3	1 4	1 4	···	1 4	return ∞
T	1 2 3 4	1 3 4	1 3 4	···	1 3 4	
test	ff	ff	ff	...	ff	

Figure 4. Execution traces of the static analysis algorithm.

and hence $uvaz \in L$. From all these facts, we obtain that $u \rightarrowtail uvaz$, i.e., u and $uvaz$ form a token neighbor pair. Notice that $\mathsf{TkDist}(u, uvaz) = |vaz| \geq dist + 1$. We obtain that

5. $\mathsf{TkDist}(L) \geq dist + 1$.

If the test of the conditional succeeds, i.e., $T \cap CoAcc = \emptyset$ we obtain that it cannot be that $\mathsf{TkDist}(L) \geq dist + 1$. Therefore, $\mathsf{TkDist}(L) \leq dist$. Using Part (1) of the invariant, we conclude that $\mathsf{TkDist}(L) = dist$ and therefore the algorithm correctly returns with the value $dist$ at Line 9.

We will now consider the execution after the conditional, that is, when Line 10 is reached. The variable S is updated to include those states of T that are non-final. We thus obtain:

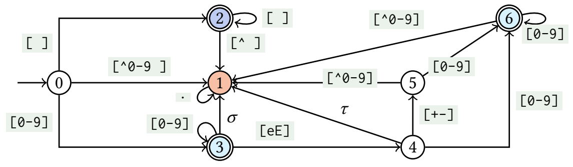
6. S contains a state q iff there exist a token $u \in L \cap \Sigma^+$ and a string $va \in \Sigma^{dist+1}$ such that $\delta(uva) = q$ and $w \notin L$ for every w with $u < w \leq uva$.

The next statement increments $dist$ by 1, and we conclude that the loop invariant holds at end of the body of the loop.

After the main loop terminates, we know that the invariant holds. Since the loop guard has failed, we also know that $\neg(dist < |\mathcal{A}| + 2)$, i.e., $dist \geq |\mathcal{A}| + 2$. Using Part (2) of the invariant, we obtain that $dist = |\mathcal{A}| + 2$. Then, using Part (1) of the invariant, we see that $\mathsf{TkDist}(L) \geq |\mathcal{A}| + 2$. Lemma 11 implies that it must be $\mathsf{TkDist}(L) = \infty$. So, the algorithm correctly returns ∞ at the end. □

Complexity of Static Analysis. Let M be the size $|\mathcal{A}|$ of the DFA for the tokenization grammar. The running time of the analysis is dominated by the execution of the while loop. First, notice that the loop executes $O(M)$ times, where M is the number of DFA states. Every repetition of the loop involves computing the set of successors of S and performing the check $T \cap CoAcc$. Using the appropriate data structures to represent S, T, and $CoAcc$ (e.g., $CoAcc$ can be kept as a Boolean array with M entries), the amount of work performed in a loop repetition is $O(M)$. So, the running time of the analysis is $O(M^2)$.

Example 16. Consider `[0-9]+([eE][+-]?[0-9]+)?|[ ]+`, which has max-TND 3 (see Example 9). Below, we show the tokenization DFA, where σ = `[^0-9eE]` and τ = `[^0-9+-]`.



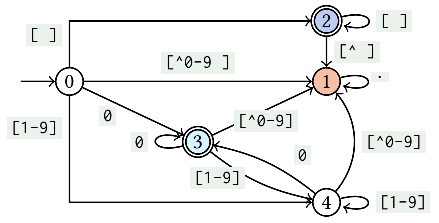
The execution trace of the static analysis algorithm (Fig. 3) for this grammar is shown in Fig. 4 (left). The row labeled with "test" refers to the condition "$T \cap CoAcc = \emptyset$" that is checked by the algorithm. Note that the set of co-accessible states of the automaton is $CoAcc = \{0, 2, 3, 4, 5, 6\}$. In line 3, we initialize S to all reachable final states, which is $\{2, 3, 6\}$. At the first iteration of the loop, we compute T to be the set of all successors of states in S. The successors of state 2 are 2 and 1; the successors of state 3 are 3, 1, and 4; and the successors of state 6 are 6 and 1. So, T is set to be $\{1, 2, 3, 4, 6\}$. Since T intersects $CoAcc$, execution continues to line 10. At this point, S is updated to contain the non-final states of T, i.e., $S = \{1, 4\}$. Moreover, $dist$ is updated to 1.

The loop execution continues in the same manner until $dist$ becomes 3, in which case the only state left in S is the rejecting state 1. The only successor of 1 is 1, so T also becomes $\{1\}$. Now, since $T \cap CoAcc = \emptyset$, we return $dist = 3$. The path witnessing $\mathsf{TkDist}(L) \geq 3$ is

$$0 \xrightarrow{\texttt{[0-9]}} 3 \xrightarrow{\texttt{[eE]}} 4 \xrightarrow{\texttt{[+-]}} 5 \xrightarrow{\texttt{[0-9]}} 6 .$$

The transitions are annotated with character classes σ, because for any letter choice we get a witnessing path.

Example 17. Consider `[0-9]*0|[ ]+`, which has max-TND ∞ (Example 9). We show the tokenization DFA for the grammar:



The execution trace of the static analysis algorithm (Fig. 3) for this grammar is shown in Fig. 4 (right). Initially, $S = \{2, 3\}$ consists of all reachable final states. The successors are $T = \{1, 2, 3, 4\}$. Since $CoAcc = \{0, 2, 3, 4\}$, we have $T \cap CoAcc \neq \emptyset$.

For $dist = 1$, we get $S = \{1, 4\}$ by filtering out the final states in T. Taking the successors of each state in S, we update $T = \{1, 3, 4\}$, which has a nonempty intersection with $CoAcc$. Then, we notice that the execution stabilizes in the sense that S and T no longer change. However, as soon as $dist$ becomes 7, the loop guard at line 6 of Fig. 3 fails, and we jump to line 12 to return ∞.

```
// Input: grammar $\bar{r}$ with TkDist($\bar{r}$) ≤ 1 and stream $text$
Function Tokenize($\bar{r}$ : List(Reg(Σ)), $text$ : Stream(Σ)):
    $\mathcal{A} \leftarrow$ tokenization DFA for $\bar{r}$
    $T \leftarrow$ token-extension table for $\mathcal{A}$
    $\mathbb{N}$ $startP \leftarrow 0$   // start position for token
    $\mathbb{N}$ $pos \leftarrow 0$   // current position in stream
    $\mathbb{S}$ $q \leftarrow$ initial state of $\mathcal{A}$   // current DFA state
    while true do // left-to-right pass, no backtracking
        // $\delta_{\mathcal{A}}$: transition function of $\mathcal{A}$
        $q \leftarrow \delta_{\mathcal{A}}(q, text[pos])$
        $pos \leftarrow pos + 1$   // move position to the next symbol
        if $T[q][text[pos]]$ then // check for maximal token
            // $\Lambda(q)$ is the preferred token id
            emit $(text[startP..pos], \Lambda(q))$
            $startP \leftarrow pos$   // start position for next token
            $q \leftarrow$ initial state of $\mathcal{A}$
```

Figure 5. Tokenization for max-TND at most 1.

We discussed in Example 9 that this grammar has max-TND ∞ because `0` $\rightarrowtail$ `0 1`^{i}`0` for every $i \geq 0$. In fact, these tokens correspond to paths in the automaton of the following form: $0 \rightarrow^{0} 3 \rightarrow^{1} 4 \rightarrow^{1} 4 \rightarrow^{1} \cdots \rightarrow^{1} 4 \rightarrow^{1} 4 \rightarrow^{0} 3$.

5 Efficient Streaming Tokenization

The previous section has presented a static analysis that computes the maximum token distance of an arbitrary tokenization grammar. In this section, we propose an efficient streaming tokenization algorithm that is applicable to grammars with bounded max-TND. We call our algorithm ***StreamTok***. It has time complexity $O(n)$, where n is the length of the input stream. We will start by examining the simple case of max-TND 1. Then, we will delve into the more general case where the max-TND is any number $K < \infty$.

5.1 Streaming Algorithm for Token Distance 1

Now, we study the case where the maximum token neighbor distance is 1. E.g., the grammar `[0-9]+|[ ]+` has max-TND 1.

Let $\bar{r}$ be a tokenization grammar with $\mathsf{TkDist}(\bar{r}) = 1$ and $\mathcal{A}$ be its tokenization DFA. Suppose that the execution of $\mathcal{A}$ over the input string reaches a final state q. There are two possibilities for the matched token u: (1) it is a maximal token, or (2) its extension ua with the next symbol a is also a token, which means that u is not maximal. In order to make this distinction, we pre-compute a table T, which is indexed by the states of $\mathcal{A}$ and the symbols of the alphabet. We call T the ***token-extension table*** for $\mathcal{A}$. For a state q of $\mathcal{A}$ and a symbol $a \in \Sigma$, we define $T[q][a]$ = true iff (1) q is final and (2) $\delta(q, a)$ is not final.

The algorithm of Fig. 5 performs a simulation of the execution of the tokenization DFA $\mathcal{A}$. The difference here is that we check in line 10 whether we have identified a token (i.e., the current state is final) that is maximal (i.e., it cannot be extended to a longer one). When a maximal token is found, the algorithm emits a token to the output stream. The output pair includes both the substring that constitutes the token, as well as its token identifier. The algorithm then proceeds to restart $\mathcal{A}$ from its initial state in order to tokenize the rest of the input stream.

Example 18. Consider the grammar `[0-9]+|[ ]+`, which we have seen in Example 5 and Fig. 1. It has max-TND 1, and we can apply the algorithm of Fig. 5. For ease of presentation, we consider character classes for the token-extension table instead of individual characters. The token-extension table T has two true entries: $T[2][$`[^ ]`$]$ and $T[3][$`[^0-9]`$]$. We have $T[2][$`[^ ]`$]$ = true because, once we have a token matching the rule `[ ]+` and are therefore in the final state 2, seeing a non-space character (`[^ ]`) next ensures that the token cannot be extended and should be emitted.

To illustrate the execution of the algorithm in Fig. 5, consider the input text `12␣` (as the prefix of some input stream). When the tokenization DFA (shown in Fig. 1) reads `1` and reaches the final state 3, we look up the table entry $T[3][$`2`$]$, which is false. Thus, we know that the token `1` is not maximal. After reading the next character `2`, the current DFA state remains 3. The table entry $T[3][$`␣`$]$ is true, which means that the token `12` found so far is maximal and should be emitted.

The algorithm of Fig. 5 has the argument *text*, which is the input stream of alphabet symbols. Even though our programming syntax allows random access into *text*, the algorithm only performs a left-to-right pass over *text* (*pos* gets incremented at every step). So, our syntax can be easily translated into a more pure syntax for streaming algorithms.

The main work that the algorithm performs are table lookups for the transition ($\delta_{\mathcal{A}}$) and for checking whether we have a maximal token (T). So, we perform $O(1)$ work per input symbol. It follows that the algorithm has time complexity $O(n)$, where n is the length of the input stream.

5.2 StreamTok: General Streaming Tokenization

Let $\bar{r}$ be a tokenization grammar. We will now describe a streaming tokenization algorithm, which we call ***StreamTok***, for the case where $\mathsf{TkDist}(\bar{r}) = K < \infty$. The main idea is that determining whether an identified token is maximal requires information about the K characters that lie ahead.

Let $\mathcal{A}$ be the tokenization DFA for $\bar{r}$. A *token-extension path* π in $\mathcal{A}$ is a sequence

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} q_{k-1} \xrightarrow{a_k} q_k$$

where $k \geq 1$, q and q_k are final, and every q_i with $i = 1, \ldots, k-1$ is non-final. Our assumption $\mathsf{TkDist}(\bar{r}) = K$ implies that $k \leq K$. We define $\mathsf{fst}(\pi) = q$ to be the first state of the path π and $\mathsf{label}(\pi) = a_1 a_2 \ldots a_k$ to be the string that can be read from the path. Let $\mathsf{TePaths}(\mathcal{A})$ be the set of all token-extension paths in $\mathcal{A}$. Let $\mathsf{TeNFA}(\mathcal{A})$ be the NFA that recognizes the finite set of regular expressions $\{\mathsf{pad}(\mathsf{label}(\pi)) \mid$

```
   // Input: grammar $\bar{r}$ with TkDist($\bar{r}$) = $K < \infty$ and stream text
1  Function Tokenize($\bar{r}$ : List(Reg($\Sigma$)), text : Stream($\Sigma$)):
2      $\mathcal{A} \leftarrow$ tokenization DFA for $\bar{r}$
3      $\mathcal{B} \leftarrow$ token-extension DFA for $\mathcal{A}$
4      $T \leftarrow$ token-maximality table for $\mathcal{A}$ and $\mathcal{B}$
5      $\mathbb{N}$ startP $\leftarrow 0$   // start position for token
6      ($\mathbb{S}$ $q$, $\mathbb{S}$ $S$) $\leftarrow$ (initial state of $\mathcal{A}$, initial state of $\mathcal{B}$)
7      for $pos = 0, 1, \ldots, K-1$ do // traverse first window (size $K$)
8          $S \leftarrow \delta_{\mathcal{B}}(S, text[pos])$ // $\delta_{\mathcal{B}}$: transition function of $\mathcal{B}$
9      $\mathbb{N}$ pos $\leftarrow 0$   // start position for stream
10     while true do // left-to-right pass without backtracking
11         $S \leftarrow \delta_{\mathcal{B}}(S, text[pos+K])$ // $\mathcal{B}$ is $K$ symbols ahead of $\mathcal{A}$
12         $q \leftarrow \delta_{\mathcal{A}}(q, text[pos])$ // $\delta_{\mathcal{A}}$: transition function of $\mathcal{A}$
13         $pos \leftarrow pos + 1$   // move position to the next symbol
14         if $T[q][S]$ then // check for maximal token
               // $\Lambda(q)$ is the preferred token id
15             emit ($text[startP..pos], \Lambda(q)$)
16             startP $\leftarrow$ pos   // start position for the next token
17             $q \leftarrow$ initial state of $\mathcal{A}$
```

Figure 6. Streaming tokenization when $\mathsf{TkDist}(\bar{r}) = K < \infty$.

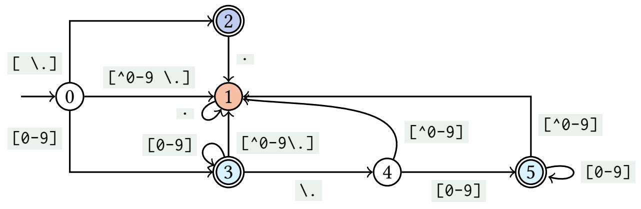
$\pi \in \mathsf{TePaths}(\mathcal{A})\}$, where $\mathrm{pad}(a_1a_2 \ldots a_k) = a_1a_2 \ldots a_k \cdot \Sigma^{K-k}$. The idea is that the pad function "pads" the string $a_1a_2 \ldots a_k$ with $K - k$ character classes Σ (to accept any character) so as to reach length exactly equal to K. We call $\mathsf{TeNFA}(\mathcal{A})$ the *token-extension NFA* for $\mathcal{A}$. Every state s of $\mathsf{TeNFA}(\mathcal{A})$ is associated with a specific path π of $\mathsf{TePaths}(\mathcal{A})$, and we define the label $\Lambda(s) = \mathsf{fst}(\pi)$, which records the first state of the path. The ***token-extension DFA*** for $\mathcal{A}$, denoted by $\mathsf{TeDFA}(\mathcal{A})$, results from $\mathsf{TeNFA}(\mathcal{A})$ by using a modified powerset construction that, informally, "restarts" the NFA at every step. Every state S of $\mathsf{TeDFA}(\mathcal{A})$ corresponds to a subset of states of $\mathsf{TeNFA}(\mathcal{A})$.

Fig. 6 shows our streaming tokenization algorithm for grammars with bounded max-TND, which uses the token-extension DFA described above. The token-extension DFA $\mathcal{B} = \mathsf{TeDFA}(\mathcal{A})$ can be used to compute some information about the K characters that lie ahead of the current position for tokenization. This can be easily implemented by executing the tokenization automaton $\mathcal{A}$ with a delay of K characters relative to $\mathcal{B}$ (see lines 11–12 of Fig. 6). Then, we can use the current states of $\mathcal{A}$ and $\mathcal{B}$ to check whether a maximal token has been found. To elaborate on this, suppose that the execution of $\mathcal{A}$ ends up at some final state q, which means that we have just found a token u (i.e., substring that matches the grammar). The token u is maximal iff the next few characters (at most K of them) cannot extend it to a longer token. This can be checked by considering the current state S of $\mathcal{B}$. Notice that $\mathcal{B}$ is exactly K characters ahead of $\mathcal{A}$, so it has already seen the characters that could potentially result in an extension. There is an extension of u iff the DFA state S ("powerstate") is final and contains some NFA state s such that $\Lambda(s) = q$. This condition says that there is a token-extension path starting with q that uses at most K of the characters that follow.

Based on the observations of the previous paragraph, we see that we can pre-compute a table T that tells us whether we have a maximal token based on the current state q of $\mathcal{A}$ and the current state S of $\mathcal{B}$. We call this the ***token-maximality table*** (for $\mathcal{A}$ and $\mathcal{B}$) and we define $T[q][S] =$ true iff (1) q is final and (2) there exists no token-extension NFA state $s \in S$ such that s is final and $\Lambda(s) = q$. In particular, line 14 of Fig. 6 performs this check: if $T[q][S]$ is true, then we know that the token is maximal and we emit it.

Note: since the token extension paths may share vertices, the token-extension paths are stored in a compact data structure in our implementation, which takes advantage of sharing. In fact, this data structure can be directly used to build the TeDFA without an explicit enumeration of the paths.

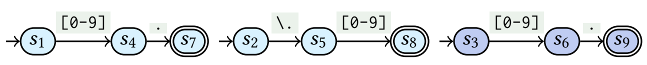
Example 19. Consider the grammar `[0-9]+(\.[0-9]+)?|[ \.]`. Its tokenization DFA $\mathcal{A}$ is shown below. We know from Example 9 that this grammar has max-TND 2.



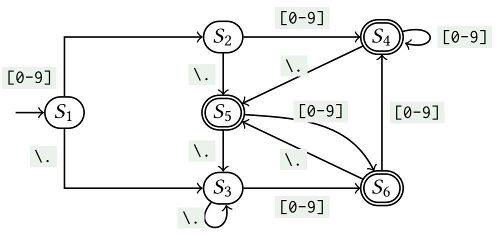
Observe that $\mathsf{TePaths}(\mathcal{A})$ contains the following three paths:

$$\begin{tikzcd} 3 \arrow[r, "{[0\text{-}9]}"] & 3 & 3 \arrow[r, "\backslash."] & 4 \arrow[r, "{[0\text{-}9]}"] & 5 & 5 \arrow[r, "{[0\text{-}9]}"] & 5 \end{tikzcd}$$

We construct $\mathsf{TeNFA}(\mathcal{A})$ as shown below. We pad the shorter paths with the character class that accepts any character (denoted `.` in PCRE notation). Observe the close correspondence between $\mathsf{TeNFA}(\mathcal{A})$ and $\mathsf{TePaths}(\mathcal{A})$. We color each state in the token-extension NFA based on the first state of each token-extension path, in order to reflect the label of each state in $\mathsf{TeNFA}(\mathcal{A})$. In particular, states s_1, s_4, s_7, s_2, s_5, and s_8 are labeled with state 3 of the original tokenization DFA $\mathcal{A}$; and states s_3, s_6, s_9 are labeled with state 5 of $\mathcal{A}$.



Applying a modified powerset construction to $\mathsf{TeNFA}(\mathcal{A})$ above, we obtain $\mathsf{TeDFA}(\mathcal{A})$ as shown below. In particular, the token-extension DFA has the states $S_1, \ldots, S_6$. We view each of these DFA states as a set of NFA states in $\mathsf{TeNFA}(\mathcal{A})$. In particular, $S_1 = I = \{s_1, s_2, s_3\}$ is the set of initial states of $\mathsf{TeNFA}(\mathcal{A})$. We also have $S_2 = \{s_4, s_6\} \cup I$, $S_3 = \{s_5\} \cup I$, $S_4 = \{s_4, s_6, s_7, s_9\} \cup I$, $S_5 = \{s_5, s_7, s_9\} \cup I$, and $S_6 = \{s_4, s_6, s_8\} \cup I$. Note that we take the union of each constructed subset with I (i.e., the set of the initial states in $\mathsf{TeNFA}(\mathcal{A})$) to simulate "restarting" the NFA at each step.



As a simple illustration of the algorithm in Fig. 6, consider the input text `1.4..` (as the prefix of some input stream). The first maximal token to be emitted is `1.4`. Note that during execution, TeDFA($\mathcal{A}$) is two input characters ahead of $\mathcal{A}$ because the max-TND is $K = 2$. Thus, when $\mathcal{A}$ reads the first character `1` and reaches a final state 3, the token-extension DFA TeDFA($\mathcal{A}$) has seen `1.4` and reached state S_6. We know that $S_6 = \{s_1, s_2, s_3, s_4, s_6, s_8\}$, where s_8 is final and is associated with state 3. This means that $T[q][S] = T[3][S_6] =$ false, and therefore `1` is not a maximal token. When the tokenization DFA $\mathcal{A}$ sees `1.4` and reaches the final state 5, the token-extension DFA has seen `1.4..` and reached $S_3 = \{s_1, s_2, s_3, s_5\}$. None of s_1, s_2, s_3 and s_5 is final, therefore $T[q][S] = T[5][S_3] =$ true and the token `1.4` is maximal.

The tokenization algorithm of Fig. 6 requires a *bounded buffer* of size K, because the automaton $\mathcal{B}$ is K symbols ahead of the automaton $\mathcal{A}$. This means that the symbols from the input stream have to be delayed by K steps for $\mathcal{A}$, and a buffer of size K can implement this delay.

Theorem 20 (Correctness). The algorithm of Fig. 6 performs maximal-munch tokenization for every grammar $\bar{r}$ with $\mathsf{TkDist}(\bar{r}) = K < \infty$.

Proof. Let $\mathcal{A}$ be the tokenization DFA and $\mathcal{B} = \mathsf{TeDFA}(\mathcal{A})$. The correctness of the algorithm relies on the following invariants for the main tokenization loop that starts at Line 10:

1. $startP \le pos$
2. the prefix $text[0..startP]$ has been correctly tokenized
3. each strict prefix of $text[startP..pos]$ is ***not*** a maximal token
4. $q = \delta_{\mathcal{A}}(init_{\mathcal{A}}, text[startP..pos])$
5. $S = \delta_{\mathcal{B}}(init_{\mathcal{B}}, text[pos..pos + K])$

We will examine the case where $text[startP..pos]$ is a maximal token. The case where it's not a maximal token can be handled similarly, using the fact that the non-maximality of the token is witnessed by some extension $text[pos..pos + k]$ with $k \le K = \mathsf{TkDist}(\bar{r})$ (i.e., of length $\le K$).

Since $text[startP..pos]$ is a token, it must be the case that q is final. S is the state of the DFA $\mathcal{B} = \mathsf{TeDFA}(\mathcal{A})$, which is constructed from TeNFA($\mathcal{A}$) through the (modified) powerset construction we described earlier. So, we can view it as a "powerstate", i.e., a set of states of TeNFA($\mathcal{A}$). Since $text[startP..pos]$ is maximal, there exists no extension

$$text[startP..pos + k]$$

that is a token (i.e., belongs to the language of the grammar).

Now, we claim that $T[q][S] =$ true. Assume for the sake of contradiction that $T[q][S] =$ false. Since we already know that q is final, it must be (from the definition of the token-maximality table T) that there exists some TeNFA($\mathcal{A}$) state $s \in S$ such that s is final that $\Lambda(s) = q$. By the definition of TeNFA($\mathcal{A}$), this implies that there exists some token-extension path π in $\mathcal{A}$ that (i) starts with q and (ii) is labeled with some nonempty prefix $text[pos..pos + k]$ of $text[pos..pos + K]$ (i.e., $k \le K$). But this means that

$$text[startP..pos] \cdot text[pos..pos + k] = text[startP..pos + k]$$

is a token, contradicting the maximality of $text[startP..pos]$.

We have thus established that $T[q][S] =$ true. We can see in Line 14 that the algorithm proceeds to correctly identify $text[startP..pos]$ as a maximal token and to appropriately update the value of the index $startP$. The preservation of the invariants follows from these observations. □

Time Complexity. The algorithm performs three table lookups ($\delta_{\mathcal{A}}$, $\delta_{\mathcal{B}}$, and T) for each symbol of the input stream. This means that the time per symbol is $O(1)$. So, the total time is $O(n)$, where n is the length of the input stream. In contrast to our algorithm, the backtracking algorithm of flex has time complexity $\Theta(mn)$, where m is the size of the tokenization grammar. This is because, in the worst case, it may have to backtrack by m steps for every symbol of the input stream. In our experimental results of §6 (see Fig. 8), we see the dependence of flex's running time on m.

Memory Footprint. The StreamTok algorithm has a small memory footprint for typical tokenization grammars. The DFAs $\mathcal{A}$, $\mathcal{B}$, and the table T have sizes that are independent of the stream length. A buffer of size K (max-TND) is needed to implement the delay. For practical workloads, the memory footprint of StreamTok is in the order of kilobytes.

6 Experiments

We have implemented the static analysis algorithm of Fig. 3 and the StreamTok algorithm for streaming tokenization (Fig. 6) in Rust. In this section, we describe experiments we have conducted to answer the following research questions:

(1) What is the maximum token neighbor distance of tokenization grammars that are used in practice?
(2) Is our static analysis algorithm (Fig. 3) sufficiently efficient for analyzing real-world grammars?
(3) Is the performance of StreamTok competitive against state-of-the-art lexers/tokenizers?
(4) How do the input stream buffer size and the average token length influence tokenization performance?
(5) What is the performance benefit that StreamTok offers to high-level real-world applications?

Table 1. Max-TND for data exchange formats and programming/query languages.

	NFA/Grammar Size	DFA Size	Max-TND
JSON	32	34	3
CSV	8	9	1
TSV	5	7	2
XML	36	32	6
C	310	263	∞
R	141	100	∞
SQL	504	398	∞

(6) What are the tradeoffs between StreamTok and the *offline* tokenization algorithms of [29] (OOPSLA'25)?

Experimental Setup. The experiments for RQ1 and RQ2 were conducted on a Linux server equipped with an Intel Xeon E5-2640 v4 CPU clocked at 2.40 GHz and 128 GB of RAM. The experiments for the rest of the RQs were conducted on a Linux workstation with an AMD EPYC 7252 8-core processor at 3.10 GHz and 256 GB of RAM.

RQ1: Analysis of Real-world Tokenization Grammars

Our goal here is to understand whether tokenization grammars used in practice are amenable to streaming tokenization with StreamTok. So, we collect and analyze (using our algorithm of Fig. 3) grammars for standard formats and programming/query languages and grammars from GitHub.

Data Formats & Programming Languages. We have executed our static analysis on the tokenization grammars of popular data exchange formats: JSON (JavaScript Object Notation) [5], CSV (comma-separated values) [42], TSV (tab-separated values) [22], and a subset of XML (Extensible Markup Language) [2]. Table 1 shows the results of our static analysis. Table 1 includes the results for tokenization grammars used in parsers for the popular programming languages C, R, and SQL. These grammars are more complex compared to the data formats: the NFA and DFA sizes are larger, and max-TND is infinite. We will not consider grammars of programming languages in our evaluation of StreamTok (RQ3 to RQ6), because program parsing involves small source files that do not need to be tokenized in a streaming fashion.

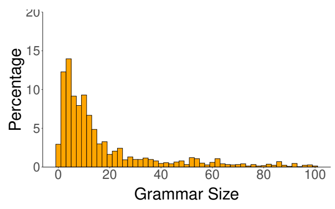
It is no surprise that simple formats such as CSV and TSV have smaller automata sizes than JSON and XML. We also notice that JSON, CSV, TSV, and XML have bounded max-TND and are therefore suitable for streaming tokenization. The variant of the CSV tokenization grammar that uses the rule `"([^"]|"")*"` for quoted fields (as per the RFC [42]) has unbounded max-TND. The set of token neighbor pairs $\{$`""` $\rightarrowtail$ `"""a"`, `""` $\rightarrowtail$ `"""aa"`, `""` $\rightarrowtail$ `"""aaa"`, $\ldots\}$ witnesses this fact. The string `""` is the empty quoted field. The first quote in the string `"""` is the opening quote of the field, and the next two quotes form an escape sequence that represents a single quote character. In our CSV grammar variant (with max-TND 1), we use the rule `"([^"]|"")*"?` instead, which makes the closing quote optional. Our variant has the same behavior for syntactically well-formed CSV documents. We can easily confirm well-formedness of a quoted-field token by checking that it contains an even number of `"` symbols.

GitHub-sourced Grammars. To obtain a large collection of real-world grammars, we have randomly sampled grammars that are available in public GitHub repositories. We have also performed de-duplication of the downloaded grammars, as there are cases where the same grammar is used across several repositories. We have created a dataset of 2669 grammars. Fig. 7 shows the results of our analysis. The size of a grammar is taken to be the number of states of its NFA. About 81% of the collected grammars are of size at most 100. Fig. 7a shows the histogram that visualizes the distribution of grammar sizes at most 100. In particular, grammars of size less than 20 are the most prevalent. The largest grammar in our dataset is of size 2496. As reported by our static analysis, 32% of the grammars have unbounded max-TND. Among the grammars with bounded max-TND (68%), 53% have max-TND 1, making up 36% of the entire dataset. Fig. 7b visualizes the distribution of max-TND over the collected grammars. Most grammars with bounded max-TND have max-TND at most 4. There are 8 outliers (i.e., grammars with bounded max-TND greater than 20) that are not shown. The largest bounded max-TND we observe is 51. We also explore the relationship between DFA size and NFA/grammar size, which is shown in Fig. 7c. For our dataset, this relationship can be decently approximated by a linear regression, with only a few prominent outliers. In theory, the size of the DFA can be exponential in the NFA size. Our data set suggests that such a blowup is uncommon in practice.

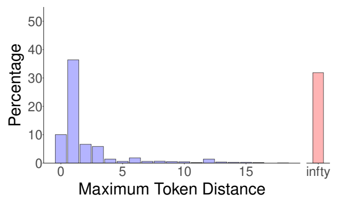
Summary of RQ1: About two-thirds of the tokenization grammars in our GitHub-sourced dataset have bounded max-TND. Popular data exchange formats can be tokenized using grammars with bounded max-TND.

RQ2: Performance of Static Analysis

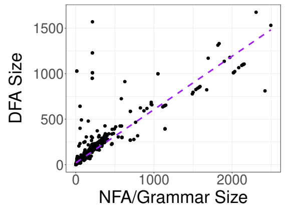
Fig. 7d shows the execution time of our static analysis (average across several trials) w.r.t. grammar size. The variance in execution time is negligible and therefore not visible in the plot. Both axes are in logarithmic scale, and we observe that the running-time growth of our algorithm w.r.t. grammar/NFA size is roughly polynomial. This empirical observation is consistent with the asymptotic complexity of the algorithm. Recall that the algorithm of Fig. 3 is quadratic w.r.t. DFA size. We also see in Fig. 7c that the relationship between DFA and NFA/grammar size is roughly linear for our dataset. So, the algorithm of Fig. 3 could appear to be quadratic in NFA/grammar size, which is consistent with the plot of Fig. 7d. More specifically, 88.7% of the grammars are analyzed in under 1 ms; 97.9% in under 10 ms; 99.4% in under 100 ms; and 99.96% in under 1 sec. The most challenging



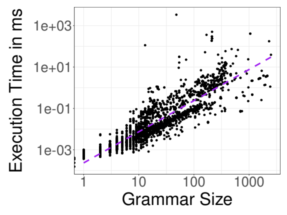
(a) Grammar size.



(b) Max-TND.



(c) DFA & NFA size.



(d) Analysis time.

Figure 7. Analysis results for our GitHub-sourced dataset of tokenization grammars.

grammar in the dataset has size 48, its DFA has size 10703, and our tool takes 3.38 seconds to analyze it.

Summary of RQ2: The worst-case polynomial time complexity of our static analysis is consistent with our experimental observations. Our analysis executes in under 100 ms (resp., 1 sec) on 99.4% (resp., 99.96%) of the grammars, so it is suitable for practical use.

RQ3: StreamTok Performance Against Existing Tools

We evaluate the performance of StreamTok against existing lexers/tokenizers. We consider both (i) synthetic instances to explore the worst-case behavior of the tools, and (ii) realistic instances to investigate performance for practical workloads.

Baseline Tools. For the experimental comparison against StreamTok, we consider the following baseline approaches: flex [47], the algorithm of Reps [38], Rust nom [12], Rust plex [44], Rust regex [14], and ExtOracle [29]. The lexer generator **flex** [47] produces a C implementation of the DFA-based backtracking algorithm that we described in Fig. 2. Even though our pseudocode description is not streaming (for the sake of simplicity), flex explicitly supports streams. It can process an input stream (read with system calls to the OS) in a block-by-block fashion. It tokenizes one block as much as possible before moving on to processing the next block. The other tools either do not explicitly support streaming input or they have semantic differences from maximal-munch tokenization. For this reason, we consider flex to be the most suitable baseline for comparing with StreamTok. The tokenization algorithm of **Reps** [38] builds upon flex's algorithm by adding a memoization table that helps avoid excessive backtracking. Since StreamTok is implemented in Rust, we also consider the Rust libraries **nom** [12], **plex** [14], and **regex** [14]. These libraries do not provide support for the streaming setting, which raises two issues: (i) the user has the additional burden of writing code for block-by-block stream processing, and (ii) tokens can straddle the boundaries of stream blocks, an error-prone situation that the user might handle incorrectly. The Rust crates nom and regex do not target the maximal-munch tokenization problem as they perform greedy matching, but we have been able to encode our benchmark tasks with them. For example, Rust::regex uses

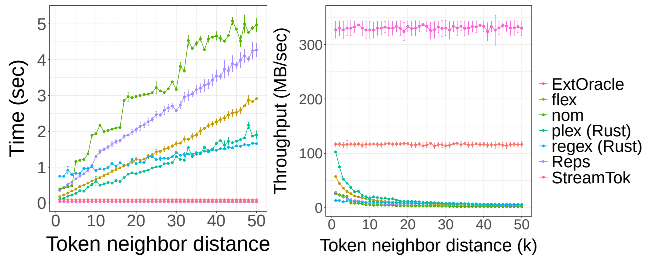


Figure 8. RQ3: Performance comparison on the family of grammars $\bar{r}_k$ = `(a{0,k}b)|a` with TkDist$(\bar{r}_k) = k$. StreamTok and ExtOracle have $\Theta(1)$ time-per-symbol, i.e., independent of k. For other tools, time-per-symbol is $\Theta(k)$.

the greedy (PCRE) semantics for disambiguation [13, 19], which does not always coincide with the maximal-munch semantics [32]. The tokenization algorithm **ExtOracle** [29] is *inherently offline* and therefore cannot be used with streaming input. To run ExtOracle, the entire input data has to be first loaded in memory, which can make it impractical in the streaming setting.

Microbenchmark: Worst-case Behavior. We consider the family of grammars $\bar{r}_k$ = `(a{0,k}b)|a` with TkDist$(\bar{r}_k) =$ k. That is, the parameter k is the max-TND of the grammar $\bar{r}_k$. Also notice that the size m of the grammar is linear in k (bounded repetition is treated as an abbreviation). We use a 10 MB input string consisting of only `a` letters. We know that flex backtracks by k characters for each input symbol, which means that it has $\Theta(k)$ time-per-symbol complexity.

In Fig. 8, we see the results for all tools. The left plot shows the execution time w.r.t. k. Notice that StreamTok and ExtOracle have constant (w.r.t. k) running time. For all other tools, the running time is proportional to k. The right plot shows the throughput of the tools w.r.t. k. StreamTok and ExtOracle have constant throughput. For the rest of the tools, throughput drops substantially as the max-TND k increases. The main observation is that the performance of all prior streaming tokenization approaches can degrade for certain difficult instances. The performance of StreamTok is robust w.r.t. the choice of tokenization grammar.

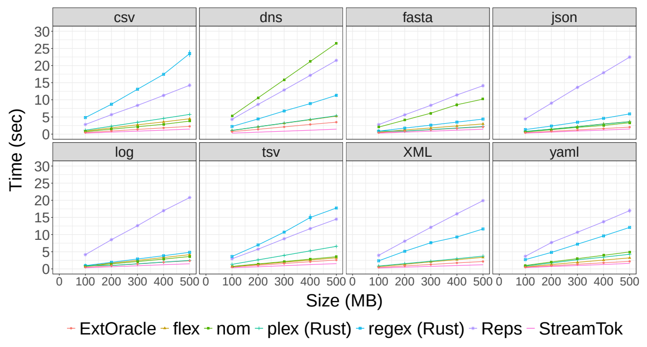


Figure 9. RQ3: Tokenization time for textual data formats.

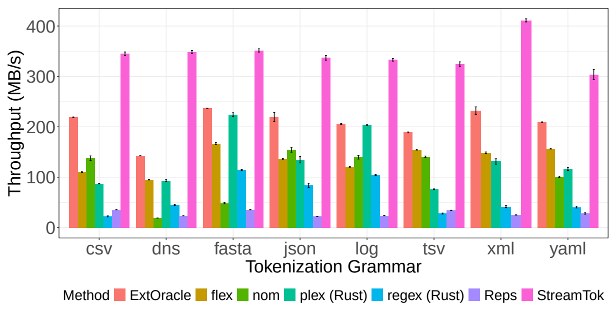


Figure 10. RQ3: Throughput of StreamTok and baseline tools for tokenizing various text-based data formats.

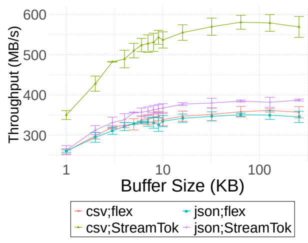
Practical Workloads. In addition to common data formats (JSON, CSV, TSV, and XML), we also include grammars for tokenizing YAML, FASTA, and DNS zone files. YAML [37] (max-TND 2) is an extension of the JSON format that uses indentation to enhance the human readability of documents. FASTA [36] (max-TND 1) is a textual format that is used to encode protein and DNA sequences in bioinformatics applications. We also include a tokenization grammar for log files (max-TND 1) stored under the directory `/var/logs/` of Linux. In RQ5, we discuss more log parsing applications and how they benefit from StreamTok. DNS zone files [33] (max-TND 1) include DNS records that are separated by newlines and whitespace that follow the definitions in RFC 4034 [40].

Fig. 9 shows the running time of StreamTok and all baseline tools for the previously-described formats. There is a plot for each data format. All grammars have bounded max-TND. In each plot, the horizontal axis is the length of the input stream (in MB = 10^6 bytes) and the vertical axis is the execution time (in seconds). The plots of Fig. 9 show that all tools exhibit linear-time behavior (w.r.t. stream length) across all workloads. Fig. 10 shows the throughput for each tool and workload. We notice that StreamTok outperforms all other tools. The second fastest tool is ExtOracle, but it is offline. As mentioned earlier, flex [47] is the only tool (apart from StreamTok) that truly supports streaming tokenization and is therefore the most suitable baseline for our performance comparison. It can be seen in Fig. 10 that StreamTok is 2–3 times faster than flex.

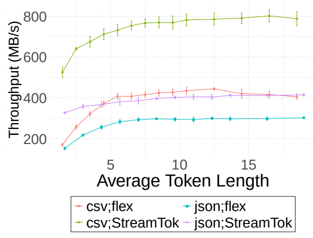
Summary of RQ3: StreamTok is asymptotically faster than prior streaming tokenizers and there are grammars for which this is reflected in tokenization performance (Fig. 8). StreamTok outperforms other tokenizers on practical workloads involving various data formats (Fig. 9). It is 2–3 times faster than flex, which is a popular lexer generator with explicit streaming support (Fig. 10).

RQ4: Impact of Buffer Size and Token Length

We identify two parameters in the streaming setting that can affect performance, namely buffer capacity (for the input stream) and average token length. Flex is the only existing tool among all considered ones that is inherently streaming. Hence, in this section, we only consider flex and StreamTok.



(a) Buffer capacity.



(b) Average token length.

Figure 11. RQ4: Examine different buffer and token sizes.

Effect of the Input Stream Buffer Capacity. For streaming tokenization, buffer capacity matters to both the performance and latency, as the input stream is delivered chunk-by-chunk to the tokenizer. Whenever we refill the buffer, we need to perform a `read` system call and move any unprocessed input from the end of the buffer to the start. Thus, if the buffer is too small, this overhead becomes significant and negatively affects the throughput. However, a larger buffer causes higher latency and uses more memory. Therefore, there is a tradeoff between throughput and latency.

Fig. 11a examines different buffer capacities for JSON and CSV using flex and StreamTok. The performance improves as the buffer increases to 64 KB and it plateaus beyond this buffer size. Therefore, we conclude that 64 KB is the most suitable buffer capacity, which also aligns with the default Unix pipe buffer capacity.

Effect of Token Length. We investigate the relationship between average token length and tokenization performance. Fig. 11b shows the results for tokenizing CSV and JSON with flex and StreamTok. The input stream buffer capacity is 64 KB. When the tokens are shorter, the tokenization performance drops. This happens because token generation (and consumption) requires additional work to be performed.

Summary of RQ4: A buffer capacity of 64 KB is a good choice for StreamTok. The number aligns with the Unix pipe buffer capacity. In addition, we observe that streams with shorter tokens result in reduced performance.

RQ5: Higher-level Applications that Use Tokenization

Tokenization is typically part of a larger pipeline that implements a higher-level application. We consider here several such applications to demonstrate that optimizing tokenization is worthwhile.

Log Parsing. Computing systems employ logging to record runtime information that can be used later for analysis and debugging. Logs are typically lists of events, and they are presented as loosely structured text. Log parsing is the task of converting raw logs into a semi-structured representation that is more suitable for further processing. Due to their simplicity, the parsing of logs can be performed with just a tokenizer, without using complex stack-based parsing algorithms. There are settings where the timely analysis of logs demands the use of a streaming log parsing technique. Even in cases where log events are batched in files, stream processing can be essential when the log files are too large to load in memory, or they are fed to pre-processing pipelines (e.g., decompression, filtering, etc.) that generate streams.

We consider commonly used log formats, including Android, Apache HTTP Server, and Linux. For each log format, we have handcrafted a suitable tokenization grammar that can tokenize real logs from LogHub [50] and Kaggle [21].

We investigate whether tokenization takes up a significant amount of time in log parsing, and whether we can reduce the running time by using StreamTok. We consider the computational task of converting raw logs into a semi-structured TSV representation. Table 2 shows the execution time when flex or StreamTok are used to tokenize. We observe that StreamTok is substantially faster than flex. Most importantly, the use of StreamTok instead of flex gives a 2.5× to 3.1× speedup for the overall log-to-TSV application.

Format Conversions and Validation. Tokenization is an important part of implementing format conversions and data validation. Format conversions are common in data migration, and data validation is crucial for ensuring that data is of an appropriate form before it is processed. For format conversions, we consider CSV to JSON, JSON to CSV, JSON minification, JSON to SQL, and SQL loads. JSON minification refers to the removal of unnecessary whitespace from a JSON file. JSON to SQL refers to creating SQL commands that load a JSON file into a database. SQL loads refer to loading SQL migration files that consist of SQL INSERT INTO statements. For validation, we consider CSV schema inference and validation. CSV schema inference refers to inferring the data type for each column (we implement inference that agrees with the csvstat tool provided in csvkit [20]). CSV validation is about validating that the CSV file contains columns of specified data types. We observe in Table 2 that tokenization is a significant part of the total execution time in all applications. For example, for JSON minification, it takes $4.55 + 0.14 = 4.69$ s to minify a JSON file with flex, but only $0.73 + 0.14 = 0.87$ s with StreamTok. So, JSON minification is 5.39× faster when StreamTok is used instead of flex.

Table 2. RQ5: Application speedup when using StreamTok instead of flex. The columns 'flex' and 'StreamTok' give tokenization times (in seconds). The column 'rest' gives the time (in seconds) spent processing the token stream.

Application	flex	StreamTok	rest	speedup
Android	0.249	0.081	0.003	2.98
Apache	0.138	0.054	0.001	2.52
BGL	0.299	0.091	0.015	2.95
Hadoop	0.321	0.113	0.008	2.71
HDFS	0.271	0.082	0.008	3.10
Linux	0.180	0.064	0.003	2.74
Mac	0.283	0.093	0.004	2.97
Nginx	0.482	0.172	0.016	2.65
OpenSSH	0.174	0.064	0.005	2.61
Proxifier	0.204	0.067	0.005	2.89
Spark	0.161	0.056	0.005	2.71
Windows	0.230	0.085	0.003	2.64
JSON to CSV	4.55	1.40	0.16	3.02
JSON Minify	4.55	0.73	0.14	5.39
CSV to JSON	2.06	0.60	0.50	2.33
CSV Schema Validation	2.06	0.646	0.014	3.14
CSV Schema Infer	2.06	0.65	0.04	3.04
JSON to SQL	4.55	1.40	0.67	2.52
SQL loads	3.91	1.10	0.61	2.64

Summary of RQ5: Tokenization is a significant part of the total running time for higher-level applications: log parsing, format conversions, and data validation. StreamTok can offer a substantial performance benefit (2.5× to 5× compared to flex) for these applications by speeding up tokenization.

RQ6: Comparison with OOPSLA'25 Algorithms

The work [29] proposes the ExtOracle and TokenSkip algorithms. Both are *inherently offline*, i.e., they require the entire input to be available before the computation can start, and cannot be used in a streaming setting. More specifically, they first perform a backwards (right-to-left) pass and then a forward (left-to-right) pass over the input. The backwards pass would have to start from the end of the stream. For an unbounded stream, this is impossible. For a bounded stream, this would require buffering the entire stream before any processing can start. For large or fast streams both algorithms of [29] are impractical: they can easily run out of memory and the latency would be unbounded.

It was demonstrated in [29] that ExtOracle is the more competitive algorithm in practice, so we will only consider ExtOracle here. We have seen in RQ3 (Fig. 10) that ExtOracle has competitive throughput on many workloads. Regarding memory usage, ExtOracle buffers the entire stream and hence needs much more memory than StreamTok. We empirically verify this claim regarding memory usage (in MB) by using the same set of grammars and files as in RQ3:

Method	csv	json	tsv	log	fasta	yaml
StreamTok	0.1	0.1	0.1	0.1	0.1	0.1
ExtOracle	2003.0	2004.6	2003.0	2007.3	2003.1	2019.0

The input size is always 1000 MB (prefix from each file). For StreamTok, the memory footprint consists of the 64 KB input buffer and the DFA representation. We measure the Resident Set Size (RSS) for ExtOracle. We observe that ExtOracle needs orders of magnitude more memory to store the stream and the "lookahead tape" computed during the backwards pass. ExtOracle has the advantage that it can be applied to any grammar regardless of max-TND, while StreamTok cannot work with grammars with unbounded max-TND. This means that there is a tradeoff between efficiency and generality. While StreamTok can only be used on a restricted subset of tokenization grammars (which our static analysis of Fig. 3 can identify), it is more space-efficient and can be used on large or unbounded input streams.

Summary of RQ6: The algorithms of [29] are not suitable for the streaming setting, as they buffer the entire stream. The benefit of ExtOracle [29] is that it applies to *all* tokenization grammars, including those with unbounded max-TND. StreamTok, on the other hand, applies only to grammars with bounded max-TND. For this subclass of grammars, StreamTok is much more memory-efficient than ExtOracle, because StreamTok supports block-by-block processing for streams.

7 Related Work

Tokenization is also called lexing or scanning, especially in lexical analysis. Some early relevant works are [10, 24]. Conway [10] proposes structuring a compiler as a pipeline of stages ("coroutines") that communicate with data streams. In particular, [10] considers lexical analysis and syntactical analysis as two separate stages. Johnson et al. [24] propose the use of regular expressions and finite-state automata for the automatic generation of "lexical processors".

The standard textbook algorithm for lexing is automata-based [1]. This standard algorithm may exhibit quadratic-time behavior (in the worst case) due to backtracking. The lexer generator flex [47] implements the technique described in [1]. Reps [38] proposed a linear-time lexing algorithm that uses memoization to avoid exploring paths that are already known to fail. A disadvantage of the approach of [38] is its high memory cost, which is $O(Mn)$, where M is the size of the DFA and n is the length of the input. Two linear-time tokenization algorithms were recently proposed in [29]. The main idea is to first perform a right-to-left pass to compute lookahead information and then perform a left-to-right pass for backtracking-free tokenization. The approach of [29] solves both the quadratic backtracking issue of flex [47] and the high memory costs of [38].

Some popular lexer generators are flex [47], Ragel [9], RE/flex [39], and re2c [7]. Flex is a state-of-the-art lexer generator that supports streaming input and implements the DFA-based backtracking algorithm. Ragel, RE/flex, and re2c implement the same tokenization algorithm as flex.

Algorithms for parallel lexing are investigated in [4] and [30]. Verbatim [15], Verbatim++ [16], and Coqlex [35] are formally verified lexing tools. They produce verified lexers that are mathematically guaranteed to conform to a standard maximal-munch specification. These works are based on Brzozowski derivatives [6]. Derivative-based approaches may suffer in performance due to potential blowups in the sizes of the derivatives. However, Brzozowski derivatives are convenient for functional implementations and give rise to simple proofs of correctness. So, it is not surprising that several works in verification choose derivatives [3, 11, 45, 46, 51] over alternatives (e.g., that use automata).

8 Conclusion

We have studied the problem of maximal-munch tokenization in a streaming setting. The tokenizer is specified using a grammar that consists of regular expressions (tokenization rules). We have seen that (in the worst case) tokenization requires space that is proportional to the stream length. We have introduced the notion of maximum token neighbor distance (max-TND), which is key for identifying grammars that admit streaming tokenization. Computing max-TND is PSPACE-complete. We have designed a static analysis algorithm for computing the max-TND, which is efficient when the tokenization DFA is small. Finally, we have proposed an efficient streaming tokenization algorithm (StreamTok) that can be applied to grammars with bounded max-TND. Our experimental evaluation has shown that StreamTok has a performance advantage over existing tools and it can benefit several higher-level applications that rely on tokenization.

Future Work. One direction for future work could be to parallelize the StreakTok algorithm so that it can take advantage of multi-core CPUs when handling high-speed data streams. Parallelization is expected to be easier for bounded max-TND, as the information needed to check token maximality is more local. The techniques of [34] may be relevant. Tokenization may also benefit from acceleration with GPUs [17, 18, 27] or custom hardware [48, 49]. Finally, StreamTok could be used to accelerate data processing (e.g., JSON validation) with application-specific tokenizers [28].

Acknowledgments

This research was supported in part by the US National Science Foundation award CCF 2313062.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques and Tools* (2nd ed.). Addison Wesley.

[2] Miguel Angel García and Gonzalo Camarillo. 2008. Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists. RFC 5364. https://doi.org/10.17487/RFC5364

[3] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Interactive Theorem Proving (ITP 2016) (LNCS, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, Cham, 69–86. https://doi.org/10.1007/978-3-319-43144-4_5

[4] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel Parsing Made Practical. *Science of Computer Programming* 112 (2015), 195–226. https://doi.org/10.1016/j.scico.2015.09.002

[5] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. https://doi.org/10.17487/RFC8259

[6] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[7] Peter Bumbulis and Donald D. Cowan. 1993. RE2C: A More Versatile Scanner Generator. *ACM Letters on Programming Languages and Systems* 2, 1–4 (1993), 70–84. https://doi.org/10.1145/176454.176487

[8] Agnishom Chattopadhyay, Angela W. Li, and Konstantinos Mamouras. 2025. Verified and Efficient Matching of Regular Expressions with Lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25)*. ACM, New York, NY, USA, 198–213. https://doi.org/10.1145/3703595.3705884

[9] Colm Networks. 2021. Ragel State Machine Compiler. https://www.colm.net/open-source/ragel/. [Online; accessed January 25, 2026].

[10] Melvin E. Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (1963), 396–408. https://doi.org/10.1145/366663.366704

[11] Thierry Coquand and Vincent Siles. 2011. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Certified Programs and Proofs (CPP 2011) (LNCS, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, Berlin, Heidelberg, 119–134. https://doi.org/10.1007/978-3-642-25379-9_11

[12] Geoffroy Couprie. 2025. Nom, Eating Data Byte by Byte. https://docs.rs/nom/latest/nom/

[13] Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html. [Online; accessed January 25, 2026].

[14] Alex Crichton and Andrew Gallant. 2024. regex. https://crates.io/crates/regex

[15] Derek Egolf, Sam Lasser, and Kathleen Fisher. 2021. Verbatim: A Verified Lexer Generator. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, USA, 92–100. https://doi.org/10.1109/SPW53761.2021.00022

[16] Derek Egolf, Sam Lasser, and Kathleen Fisher. 2022. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*. ACM, New York, NY, USA, 27–39. https://doi.org/10.1145/3497775.3503694

[17] Tianao Ge, Xiaowen Chu, and Hongyuan Liu. 2025. Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/3725843.3756052

[18] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24)*. ACM, New York, NY, USA, 268–285. https://doi.org/10.1145/3617232.3624848

[19] Google. [n. d.]. GitHub - google/re2: RE2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. It is a C++ library. https://github.com/google/re2

[20] Christopher Groskopf and contributors. 2016. *csvkit*. https://csvkit.readthedocs.org/

[21] Philip Hazel. 2015. Web Server Access Logs. https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs

[22] Intenet Assigned Numbers Authority. 1993. Definition of Tab-separated-values (tsv). https://www.iana.org/assignments/media-types/text/tab-separated-values

[23] JFlex. 2024. Fast Scanner Generator for Java with Full Unicode Support. https://jflex.de/. [Online; accessed May 22, 2024].

[24] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. 1968. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Commun. ACM* 11, 12 (1968), 805–813. https://doi.org/10.1145/364175.364185

[25] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

[26] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching Using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

[27] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU Acceleration of Multi-pattern Regex Matching using Bit Parallelism. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 331 (2024), 30 pages. https://doi.org/10.1145/3689771

[28] Alexis Le Glaunec, Angela W. Li, and Konstantinos Mamouras. 2025. Streaming Validation of JSON Documents Against Schemas. *Proceedings of the VLDB Endowment* 19, 3 (2025), 509–522. https://doi.org/10.14778/3778092.3778109

[29] Angela W. Li and Konstantinos Mamouras. 2025. Efficient Algorithms for the Uniform Tokenization Problem. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1, Article 133 (2025), 27 pages. https://doi.org/10.1145/3720498

[30] Le Li, Shigeyuki Sato, Qiheng Liu, and Kenjiro Taura. 2021. Plex: Scaling Parallel Lexing with Backtrack-Free Prescanning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, USA, 693–702. https://doi.org/10.1109/IPDPS49936.2021.00079

[31] Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 92 (2024), 31 pages. https://doi.org/10.1145/3632934

[32] Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. 2024. Static Analysis for Checking the Disambiguation Robustness of Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 231 (2024), 25 pages. https://doi.org/10.1145/3656461

[33] Paul V. Mockapetris. 1987. Domain Names - Implementation and Specification. RFC 1035. https://doi.org/10.17487/RFC1035

[34] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel Finite-state Machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 529–542. https://doi.org/10.1145/2541940.2541988

[35] Wendlasida Ouedraogo, Gabriel Scherer, and Lutz Straßburger. 2023. Coqlex: Generating Formally Verified Lexers. *The Art, Science, and Engineering of Programming* 8, 1, Article 3 (2023), 30 pages. https://doi.org/10.22152/programming-journal.org/2024/8/3

[36] William R. Pearson and David J. Lipman. 1988. Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Sciences* 85, 8 (1988), 2444–2448. https://doi.org/10.1073/pnas.85.8.

2444

[37] Roberto Polli, Erik Wilde, and Eemeli Aro. 2024. YAML Media Type. RFC 9512. https://doi.org/10.17487/RFC9512

[38] Thomas Reps. 1998. "Maximal-Munch" Tokenization in Linear Time. *ACM Transactions on Programming Languages and Systems* 20, 2 (1998), 259–273. https://doi.org/10.1145/276393.276394

[39] Robert van Engelen. 2016. RE/flex: A high-performance C++ regex library and a lexical analyzer generator. https://www.genivia.com/doc/reflex/html/. [Online; accessed May 22, 2024].

[40] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. 2005. Resource Records for the DNS Security Extensions. RFC 4034. https://doi.org/10.17487/RFC4034

[41] Walter J. Savitch. 1970. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. System Sci.* 4, 2 (1970), 177–192. https://doi.org/10.1016/S0022-0000(70)80006-X

[42] Yakov Shafranovich. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. https://doi.org/10.17487/RFC4180

[43] Joshua B. Smith. 2007. Ocamllex and Ocamlyacc. In *Practical OCaml*. Apress, Berkeley, CA, USA, 193–211. https://doi.org/10.1007/978-1-4302-0244-8_16

[44] Geoffry Song. 2024. plex, a parser and lexer generator. https://crates.io/crates/plex

[45] Chengsong Tan and Christian Urban. 2023. POSIX Lexing with Bitcoded Derivatives. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2023.27

[46] Christian Urban. 2023. POSIX Lexing with Derivatives of Regular Expressions. *Journal of Automated Reasoning* 67, 3, Article 24 (2023), 24 pages. https://doi.org/10.1007/s10817-023-09667-1

[47] Vern Paxson. 1987. Flex: The Fast Lexical Analyzer. https://github.com/westes/flex. [Online; accessed May 22, 2024].

[48] Ziyuan Wen, Lingkun Kong, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/3620665.3640412

[49] Ziyuan Wen, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2025. RAP: Reconfigurable Automata Processor. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. ACM, New York, NY, USA, 1140–1154. https://doi.org/10.1145/3695053.3731106

[50] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, USA, 355–366. https://doi.org/10.1109/ISSRE59848.2023.00071

[51] Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)*. ACM, New York, NY, USA, 118–131. https://doi.org/10.1145/3636501.3636959