Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives

Derek Egolf
Northeastern University
Boston, USA
egolf.d@northeastern.edu

Sam Lasser
Tufts University
Medford, USA
samuel.lasser@tufts.edu

Kathleen Fisher
Tufts University
Medford, USA
kfisher@cs.tufts.edu

Abstract

Lexers and parsers are attractive targets for attackers because they often sit at the boundary between a software system's internals and the outside world. Formally verified lexers can reduce the attack surface of these systems, thus making them more secure.

One recent step in this direction is the development of Verbatim, a verified lexer based on the concept of Brzozowski derivatives. Two limitations restrict the tool's usefulness. First, its running time is quadratic in the length of its input string. Second, the lexer produces tokens with a simple "tag and string" representation, which limits the tool's ability to integrate with parsers that operate on more expressive token representations.

In this work, we present a suite of extensions to Verbatim that overcomes these limitations while preserving the tool's original correctness guarantees. The lexer achieves effectively linear performance on a JSON benchmark through a combination of optimizations that, to our knowledge, has not been previously verified. The enhanced version of Verbatim also enables users to augment their lexical specifications with custom semantic actions, and it uses these actions to produce semantically rich tokens—i.e., tokens that carry values with arbitrary, user-defined types. All extensions were implemented and verified with the Coq Proof Assistant.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Software and its engineering** → Parsers; • **Theory of computation** → *Regular languages.*

Keywords: lexical analysis, formal verification, Brzozowski derivatives, semantic actions

ACM Reference Format:
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 '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3497775.3503694

CPP '22, January 17–18, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9182-5/22/01.
https://doi.org/10.1145/3497775.3503694

1 Introduction

Lexers and parsers often serve as a boundary between the internals of a software system and the outside world. For this reason, they are attractive targets for attackers, who seek to gain access to the system's internals by exploiting bugs in the outward-facing components. Recent literature on software vulnerabilities includes many examples of this phenomenon [8–10, 14, 19–21]. Formally verified lexers thus have the potential to make software systems more secure by reducing their attack surface.

One recent contribution in service of this goal is the development of Verbatim [7], a verified lexer based on the concept of Brzozowski derivatives. While this tool serves as a useful starting point for further work on verified lexing, two limitations restrict its usefulness in real-world settings. First, the lexer's running time is quadratic in the length of its input string; unverified alternatives typically offer asymptotically better performance. Second, the lexer produces tokens with a simple "tagged string" representation, where the tag indicates which lexical rule produced the token and the string is a portion of the input string. Ideally, a lexer should be able to produce tokens that carry values of different types; this feature supports easy integration with a downstream parser that produces a result with a user-defined type.

In this work, we present a suite of optimizations and extensions to Verbatim that overcomes these limitations and that retains Verbatim's original correctness guarantees. Our contributions are as follows:

- We present three verified optimizations to Verbatim:
  - Compiling regular expressions to deterministic finite automata (DFAs) for faster regex matching
  - Memoizing the results of lexer subroutine calls to avoid redundant computations
  - Replacing memoized strings with string *lengths* for faster cache lookups

  We have implemented these optimizations with the Coq Proof Assistant. Each one comes with a mechanized correctness proof—i.e., a proof that the optimized lexer satisifies the same high-level specification as the

original one. To our knowledge, this combination of optimizations has not been previously verified.
- We demonstrate that these optimizations significantly improve the lexer's performance on a JSON benchmark. The optimized lexer outperforms the naive version in terms of both asymptotic behavior and point-for-point lexing speed; performance improves from quadratic to effectively linear, and the optimized lexer is faster than the naive one by a factor of roughly 200 on the largest data point.
- We extend Verbatim with the ability to produce semantically rich tokens—tokens that carry values with user-specified types—and we demonstrate empirically that using this extension need not increase the lexer's performance overhead.

In addition, we believe that this work highlights the benefits of a multi-stage approach to developing verified software, in which proving functional correctness and optimizing performance occur in separate stages. By gradually adding layers of optimization to an existing reference implementation, we were able to keep the proofs about the optimizations relatively simple and tractable. In fact, most of these proofs simply show that the optimizations preserve the extensional behavior of the "reference" lexer, which is already known to be correct.

The paper is organized as follows. §2 contains background material on regular expressions and regex matching with Brzozowski derivatives; it also outlines Verbatim's high-level structure and correctness specification. §3, §4, and §5 present the three verified lexer optimizations that are among this work's main contributions. Each of these sections explains the intuition behind one optimization, describes its implementation, sketches its mechanized correctness proof, and presents experimental results that demonstrate the optimization's contribution to performance. §6 describes our scheme for extending the lexer with semantic actions, which enable the lexer to produce semantically rich tokens. We discuss related work in §7 and future work in §8. Finally, we summarize the paper and its impact in §9.

The Coq development that accompanies this paper is publicly available online [6]. The development consists of roughly 1900 lines of specification and 3100 lines of proof—more than twice the size of the previous version of Verbatim.

2 Background

This paper expands on the previous work on Verbatim. The following section explains the overhead required to understand Verbatim.

2.1 Regular Expressions

Regular expressions (regexes) inductively denote regular languages. They are commonly used to specify lexical rules. If a string z is in the language represented by regex e, we say that z matches e and write $z \simeq e$. Verbatim uses the canonical definitions of regexes (Figure 1) and regex matching (Figure 2) from Software Foundations [23], a textbook on interactive theorem proving.

$$
\begin{array}{ll}
\textit{Symbol} & a, b \in \Sigma \\
\textit{String} & z ::= \lambda \mid az \\
\textit{Regex} & e ::= \varnothing \mid \varepsilon \mid [\![a]\!] \mid e + e \mid e \cdot e \mid e^* \\
\textit{Rule} & r ::= (l, e) \\
\textit{Token} & t ::= (l, z)
\end{array}
$$

Figure 1. Definition of strings, regular expressions, lexical rules, and tokens over an alphabet Σ. Metavariable l ranges over labels, a type that the user defines as part of a lexical specification. For brevity, we write non-empty strings without a terminal λ. For example, we write a instead of $a\lambda$.

$$
\begin{array}{ll}
\text{(MEmpty)} \quad \lambda \simeq \varepsilon & \text{(MChar)} \quad a \simeq [\![a]\!] \\
\text{(MApp)} \quad \dfrac{z_1 \simeq e_1 \qquad z_2 \simeq e_2}{z_1 +\!\!+ z_2 \simeq e_1 \cdot e_2} & \text{(MUnionL)} \quad \dfrac{z \simeq e_1}{z \simeq e_1 + e_2} \\
\text{(MUnionR)} \quad \dfrac{z \simeq e_2}{z \simeq e_1 + e_2} & \text{(MStar0)} \quad \lambda \simeq e^* \\
\text{(MStarApp)} \quad \dfrac{z_1 \simeq e \qquad z_2 \simeq e^*}{z_1 +\!\!+ z_2 \simeq e^*} &
\end{array}
$$

Figure 2. Formal specification of string-regex matching, where a string is a λ-terminated list of symbols from alphabet Σ and $z_1 +\!\!+ z_2$ is the concatenation of strings z_1 and z_2.

We make a distinction between the empty string λ and the empty regex ε, which denotes the language $\{\lambda\}$. Also, a is the string consisting solely of symbol a, but $[\![a]\!]$ is the regex that denotes $\{a\}$, the language containing only that string. The empty language is denoted by the regex $\varnothing$. Given e_1 and e_2, which denote languages L_1 and L_2 respectively, we say that $e_1 + e_2$ denotes $L_1 \cup L_2$. Additionally, $e_1 \cdot e_2$ denotes

$$\{z_1 +\!\!+ z_2 \mid z_1 \in L_1 \text{ and } z_2 \in L_2\}$$

Finally, if e denotes L, then e^* denotes

$$\{\varepsilon\} \cup \bigcup_{n=1}^{\infty} \{z_1 +\!\!+ z_2 +\!\!+ \ldots +\!\!+ z_n \mid \forall i, z_i \in L\}$$

Verbatim represents a lexical rule as a label-regex pair. A string z matches a rule (l, e) iff $z \simeq e$. The notation $z \simeq (l, e)$ represents such a match.

$$\text{(Prefix)}\quad \frac{p +\!\!+ s = z}{\texttt{Prefix}\ p\ z}$$

$$\text{(MaxPref)}\quad \frac{\texttt{Prefix}\ p\ z \qquad r \in R \qquad p \simeq r \qquad \forall p', \texttt{Prefix}\ p'\ z \wedge \texttt{len}\ p < \texttt{len}\ p' \rightarrow \forall r' \in R, \neg(p' \simeq r')}{\texttt{MaxPref}_R\ p\ z}$$

Figure 3. Definition of the maximal prefix of a string z with respect to a list of lexical rules R.

2.2 Maximal Prefixes

The concept of a *maximal prefix* is central to Verbatim's implementation and correctness specification. If string $z = p +\!\!+ s$, we say that p is a prefix of z and we write $\texttt{Prefix}\ p\ z$. Given a list of rules R and a string z, we say that p is the maximal prefix of z with respect to R iff p is the longest prefix of z that matches some rule in R. Under those conditions, we write $\texttt{MaxPref}_R\ p\ z$. Figure 3 gives the formal definition.

2.3 Brzozowski Derivatives

Verbatim uses a regex matching algorithm based on the concept of Brzozowski derivatives [3]. The derivative of a language L with respect to symbol a is defined as follows:

$$\partial_a L = \{z \mid az \in L\}$$

Informally, the derivative operation removes the leading a from those strings in L that begin with a and includes only the resulting suffixes.

The operation can be extended to strings recursively:

$$\partial_\lambda L = L$$
$$\partial_{az} L = \partial_z(\partial_a L)$$

So if we have a string z and a regular language L, we can conclude by induction on the string that

$$z \in L \iff \lambda \in \partial_z L$$

One can extend the concept of a derivative from a regular language to a regular expression. Intuitively, if regex e represents language L, then $\partial_a e = e'$ represents $\partial_a L$. The following algorithm recursively computes the derivative of a regular expression with respect to a character a:

$$\begin{aligned}
\partial_a \varnothing &:= \varnothing \\
\partial_a \varepsilon &:= \varnothing \\
\partial_a [\![b]\!] &:= \texttt{if}\ a == b\ \texttt{then}\ \varepsilon\ \texttt{else}\ \varnothing \\
\partial_a (e_1 + e_2) &:= \partial_a e_1 + \partial_a e_2 \\
\partial_a (e_1 \cdot e_2) &:= (\partial_a e_1 \cdot e_2) \\
&\quad + (\texttt{if nullable}\ e_1\ \texttt{then}\ \partial_a e_2\ \texttt{else}\ \varnothing) \\
\partial_a (e^*) &:= \partial_a e \cdot e^*
\end{aligned}$$

where $\texttt{nullable}\ r$ evaluates to $\texttt{true}$ if $\lambda \simeq r$ and $\texttt{false}$ otherwise:

$$\begin{aligned}
\texttt{nullable}\ \varnothing &:= \texttt{false} \\
\texttt{nullable}\ \varepsilon &:= \texttt{true} \\
\texttt{nullable}\ [\![b]\!] &:= \texttt{false} \\
\texttt{nullable}\ (e_1 + e_2) &:= \texttt{nullable}\ e_1 \vee \texttt{nullable}\ e_2 \\
\texttt{nullable}\ (e_1 \cdot e_2) &:= \texttt{nullable}\ e_1 \wedge \texttt{nullable}\ e_2 \\
\texttt{nullable}\ e^* &:= \texttt{true}
\end{aligned}$$

2.4 Verbatim Definitions

The Verbatim implementation has three main components: a top-level `lex` function, a maximal prefix finder, and a regex matcher.

The top-level interface has the following signature:

```
lex : String -> list Rule
    -> (list Token) * String
```

The function takes as input a string and the lexical rules with which to lex that string, and returns as output a list of tokens and the unprocessed suffix of the input string.

The `lex` function tokenizes the input by repeatedly calling the maximal prefix finder function `max_pref`:

```
max_pref : String -> list Rule
         -> Label * option (String * String)
```

This function takes a string and a list of lexical rules, and it returns the longest prefix matching any rule, the complementary suffix, and the label associated with the earliest matching rule. The function returns $(\Delta, \texttt{None})$—where Δ is a user-defined default label—if no maximal prefix can be found for any rule.

The `max_pref` function is defined in terms of a *singleton* maximal prefix finder:

```
maxpref_one : Rule -> String
            -> option (String * String)
```

which returns the maximal prefix that matches a *single* lexical rule and its complementary suffix, or `None` when no prefix matches the rule.

Finally, `maxpref_one` finds maximal prefixes with the help of a regex matcher. The matcher takes a regex and a string as input, and it recursively takes derivatives of the regex for each character in the string. The matcher returns `true` if the resulting regex is nullable and `false` otherwise.

2.5 Verbatim Correctness

Verbatim is correct with respect to a standard lexer specification based on the "maximal munch" principle [4]. In essence, this principle says that each token should be formed by taking a "maximal munch" of the remaining input string—i.e., the maximal prefix that matches a lexical rule. When

$$\frac{p \neq \lambda \qquad \texttt{MaxPref}_R\ p\ z \qquad p \simeq (l,e) \qquad (l,e) \in R \qquad \forall r', \texttt{Index}_R\ r' < \texttt{Index}_R\ (l,e) \rightarrow \neg(p \simeq r')}{\texttt{FirstToken}_R\ (l,p)\ z} \quad \text{(FirstToken)}$$

$$\frac{\forall t, \neg\texttt{FirstToken}_R\ t\ z}{\texttt{Tokens}_R\ ([\ ], z)\ z} \quad \text{(TokensNil)}$$

$$\frac{z = p +\!\!+ s \qquad \texttt{FirstToken}_R\ (l,p)\ z \qquad \texttt{Tokens}_R\ (ts,u)\ s}{\texttt{Tokens}_R\ ((l,p) :: ts, u)\ z} \quad \text{(TokensCons)}$$

Figure 4. Formal specification of the maximal munch principle applied to a string z and a list of lexical rules R. In TokensCons, the unprocessed suffix is u; in TokensNil, all of z is unprocessed.

multiple rules match the same maximal prefix, the rule that appears first in the list of rules should apply.

Verbatim's formalization of the maximal munch principle appears in Figure 4. The `FirstToken` predicate holds for a token (l,p) and a string z with respect to a list of rules R when

- p is non-empty
- p is the maximal prefix of z
- p matches some rule with label l: $(l,e) \in R$
- (l,e) is the first rule in R that matches p

in which case we write $\texttt{FirstToken}_R\ (l,p)\ z$.

The `FirstToken` predicate is used to define the `Tokens` inductive predicate, Verbatim's main correctness specification. This latter predicate says that a string's correct tokenization consists of the initial token that `FirstToken` produces, followed by the correct tokenization of the resulting suffix (see the TokensCons rule). When no first token exists, the entire remaining suffix should be returned unprocessed (see the TokensNil rule). We write $\texttt{Tokens}_R\ (ts,u)\ z$ to mean that ts are the correct tokens for some prefix z of input string $z +\!\!+ u$, where u is the (often empty) part of the string that could not be processed.

With this specification in hand, we can state Verbatim's high-level correctness properties:

1. **Soundness**: If Verbatim produces a tokenization for its input, then that tokenization is correct according to the maximal munch specification.
2. **Uniqueness**: According to the specification, there is only one way to tokenize a string with a given list of rules.
3. **Completeness**: If a tokenization for a given input string is correct according to the specification, then Verbatim outputs exactly that tokenization.

3 Optimization #1: A DFA-Based Matcher

The singleton maximal prefix finder `maxpref_one` (Section 2.4) is parametric in the regex matcher that it uses. Verbatim's original regex matcher computed Brzozowski derivatives at runtime, but this approach is expensive. The first optimization we make to the naive lexer is to replace the derivative-based matcher with a DFA-based matcher. Because the lexical rules are still defined with regexes, it is necessary to convert those regular expressions to DFAs. Brzozowski gave an algorithm for this conversion [3]; our verified implementation uses a modified version of this algorithm.

3.1 DFA Representation

Recall that a DFA is typically defined as a 5-tuple

$$(Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states
- Σ is a finite alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

We define the DFA corresponding to a regular expression as follows. First, our states are denoted by regular expressions. We treat q_0 as a pointer while doing incremental matching; for example, if we want to know if the string az is accepted by the DFA $(Q, \Sigma, \delta, q_0, F)$, we compute $\delta(q_0, a) = q_1$ and then recursively determine if z is accepted by the updated machine $(Q, \Sigma, \delta, q_1, F)$. As a base case, $(Q, \Sigma, \delta, q_n, F)$ accepts ε if q_n is an accepting state. We also relax the usual finiteness requirements. Namely, we say that Q is the set of all regular expressions (an infinite set) and we define F as a function from regular expressions to booleans, rather than as a finite set. This function classifies a regex as an accepting or non-accepting state. Finally, the following definitions for the transition function and the accepting states complete the construction of a DFA that mirrors the behavior of the naive matcher on the input regex.

$$\delta(e, a) = \partial_a e$$

$$F(e) = \texttt{nullable}\ e$$

In the optimized matcher, we define δ in terms of a *transition table*: a 2D array whose rows are labeled with regexes and whose columns are labeled with the elements of Σ. An entry $T[e, a]$ of the table is an `option regex`—`Some` e' (filled) or `None` (empty). If every cell of the table is filled and if every entry appears as the label of a column, we say that the table is complete. Otherwise we say that the table is partial. A complete table can be used directly as the transition function:

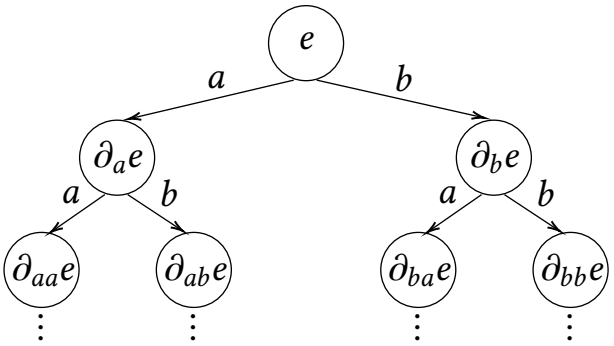
$$\delta(e, a) = T[e, a]$$

3.2 Brzozowski's Construction

In this section we describe Brzozowski's procedure for converting a regex into a theoretically complete transition table. Brzozowski's construction for converting a regex to a DFA is based on the following correspondence:

- Regular expressions are states.
- Nullable regular expressions are accepting states.
- The Brzozowski derivative is the transition function.

The idea is to recursively label states with derivatives and transition to those states on the appropriate symbol. States labeled with nullable regexes are designated as accepting states. The state diagram might look like this:



Some of those nodes may share the same label, so the resulting state diagram may not be a tree. Futhermore, Brzozowski showed that the algorithm produces a finite number of states modulo a notion of regex equivalence [3]. In other words, the recursion depth is bounded by some finite number, d_e. Therefore, the procedure will eventually terminate, producing a finite state diagram.

3.3 Implementation

In this section we describe our implementation of Brzozowski's construction and explain how we soundly sidestep arguments about the recursion depth bound. Given a regex e, we use Brzozowski's construction to create a transition table T. The crux of our implementation is a function

```
fill_T : table -> regex -> list Σ -> ℕ -> table
```

that recursively take derivatives with respect to all elements of a finite alphabet until some finite recursion depth is achieved. We will write Σ to represent the list containing all elements of type Σ.

Definition 3.1. The function

$$\texttt{fill\_T}\ T\ e\ \Sigma\ n$$

is computed as follows. For each $a \in \Sigma$ perform the following operations and update T.

1. Compute $\partial_a e$.
2. Update T so that $T[e, a] = \partial_a e$.
3. If $\partial_a e$ appears as a row label in T, do not make a recursive call. Continue iterating through Σ.
4. Otherwise, compute $T' = \texttt{fill\_T}\ T\ \partial_a e\ \Sigma\ (n-1)$ and set $T = T'$.

Let $\boxtimes$ be the empty table and let d_e be Brzozowski's recursion depth. Once we have computed $T = \texttt{fill\_T}\ \boxtimes\ e\ \Sigma\ d_e$, we can compute the set of nullable states:

$$E = \{e \mid e \text{ is a row label in } T \text{ and } \texttt{nullable}\ e = \texttt{true}\}$$

We could then define δ and F as

$$\delta(e, a) = T[e, a]$$

$$F(e) = e \in E$$

but these definitions would complicate the proof of correctness. In particular, we know that $T[e, a] = \partial_a e$ *only if* the entry at $[e, a]$ is non-empty, so the above definitions only work when T is complete. In order to prove the table complete, we would need to formalize Brzozowski's upper bound proof in Coq. We avoid this complication by defining δ and F as functions of T as follows:

Definition 3.2.

$$\delta_T(e, a) = \begin{cases} \partial_a e & \text{if } T[e, a] = \mathsf{None} \\ T[e, a] & \text{otherwise} \end{cases}$$

Definition 3.3.

$$F_T(e) = \begin{cases} e \in E & \text{if } e \text{ is a row label in } T \\ \texttt{nullable}\ e & \text{otherwise} \end{cases}$$

In other words, we use information from the table whenever it is present. When it is not, we compute derivatives (or check nullability) on the fly. The only proofs required are that $T[e, a] = \partial_a e$ when that entry is not empty and that all elements of E are nullable. The rest follows from the correctness of the naive matcher.

We do not prove that the initial fuel argument to `fill_T` is correct (i.e., sufficient to produce a complete transition table). Insufficient fuel would result in an incomplete table, and would therefore require the lexer to compute derivatives at lex time, similar to the naive implementation. This possibility is an optimization concern, not a correctness concern. In our evaluations, the DFA-based lexer does not need to compute derivatives on the fly, and its performance improves considerably relative to the naive version (see Section 3.5).

3.4 Sketch of Mechanized Correctness Proof

The correctness proof for this optimization shows that the automaton-based matcher is extensionally equivalent to the naive matcher. Therefore, because the naive matcher is correct, so too is the optimized version.

Intuitively, we use `fill_T` to replace entries of the empty table with derivatives. Therefore it is necessarily true that every entry of the table is either empty or an appropriate derivative. We refer to this property as Brzozowski-ness.

Definition 3.4. A table T is a Brzozowski table if for all e, a either

$$T[e, a] = \texttt{None}$$

or

$$T[e, a] = \partial_a e$$

We first show that Brzozowski-ness is closed under `fill_T`:

Lemma 3.5. For all n, e, T, Σ, if T is a Brzozowski table, then

$$T' = \texttt{fill\_T}\ T\ e\ \Sigma\ n$$

is also a Brzozowski table.

Proof. By induction on n. All other terms are general in the inductive hypothesis.
When $n = 0$, $T' = T$ and is therefore a Brzozowski table.
Before `fill_T` makes a recursive call, it updates the table to contain an appropriate derivative (step 2 of Definition 3.1). This step is the only time the function modifies an entry of the table, outside of recursive calls. Now suppose T' is a Brzozowski table for $n - 1$. Then the intermediate recursive calls (step 4 of Definition 3.1) all produce Brzozowski tables. The last of these recursive calls is the output of the function, so the function as a whole returns a Brzozowski table. □

It follows immediately that filling an empty table results in a Brzozowski table.

Corollary 3.6. The table

$$T = \texttt{fill\_T}\ \boxtimes\ e\ \Sigma\ d_e$$

is a Brzozowski table.

Proof. The empty table is trivially a Brzozowski table. It follows directly from Lemma 3.5 that T is a Brzozowski table. □

We now show that the result of δ_T is equivalent to the derivative and that F_T is equivalent to `nullable`.

Lemma 3.7. Let e be a regular expression over alphabet Σ. Additionally, let

$$T = \texttt{fill\_T}\ \boxtimes\ e\ \Sigma\ d_e$$

Then for all e' and a: $\delta_T(e', a) = \partial_a e'$.

Proof. Either $T[e', a]$ is `None` or it is not. Suppose it is not `None`. Then we know from Corollary 3.6 that T is a Brzozowski table and therefore that

$$\delta_T(e', a) = T[e', a] = \partial_a e'$$

But in the case that $T[e', a]$ is `None`, it follows immediately from the definition of δ_T that

$$\delta_T(e', a) = \partial_a e'$$

□

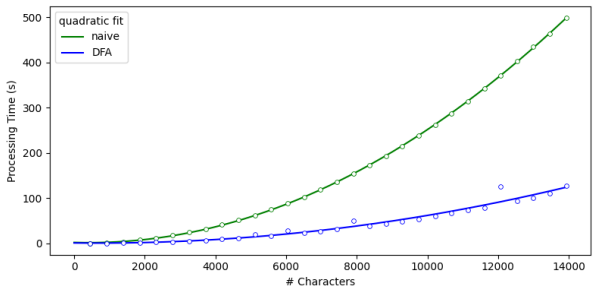


Figure 5. Peformance comparison between the naive and DFA-based lexers. The benchmark contains 30 JSON inputs of varying size. Smaller inputs are prefixes of larger inputs. Each point represents the lexer's average execution time across five trials for a given input.

Lemma 3.8. Let e be a regular expression over alphabet Σ. Additionally, let

$$T = \texttt{fill\_T}\ \boxtimes\ e\ \Sigma\ d_e$$

Then for all e', $F_T(e') = \texttt{nullable}\ e'$.

Proof. The function $F_T(e')$ is equivalent, by definition, to `nullable` e' when e' is not a row label in T. Otherwise, e' is a row label in T and therefore $e' \in E$ if and only if e' is nullable. So in this case $F_T(e')$ is still equivalent to `nullable` e'. □

Finally, we use these equivalences to show that this automaton-based matcher is equivalent to the naive matcher, which has already been proven correct.

Theorem 3.9. Let e be a regular expression over alphabet Σ. Additionally, let $T = \texttt{fill\_T}\ \boxtimes\ e\ \Sigma\ d_e$ and let DFA $D = (\texttt{regex}, \Sigma, \delta_T, e, F_T)$.

Then string z is in the language of e iff D accepts z.

Proof. It follows immediately from Lemmas 3.7 and 3.8 that D behaves exactly like the naive matcher. The naive matcher is correct, so D is as well. □

3.5 Performance

Because Verbatim no longer computes derivatives at lex time, we expected it to be faster under this optimization. However, its asymptotic behavior should not change because it still needs to traverse the entire remaining input string to produce each token.

We confirmed this hypothesis empirically by benchmarking the optimized lexer on the data set from the original Verbatim performance evaluation, using the same methodology. We obtained an executable lexer by using Coq's extraction mechanism to extract Verbatim to OCaml source code. The benchmark consists of United States GDP data from the past several decades, stored in JSON format [32].

As one can see from the plot of the results (Figure 5), the lexer's asymptotic behavior does not change. Performance does improve by a factor of four on the largest data point.

4 Optimization #2: Memoization

Reps showed that maximal munch-based lexical analysis can always be done in linear time by caching the results of frequent calls [25]. In the process of lexing a string z with respect to a rule (l, e) Verbatim makes many calls to `maxpref_one`. The input string to `maxpref_one` is always a suffix of z and the input rule is always (l, e'), where e' is an nth derivative of e for some n. Many of the calls to `maxpref_one` are made with the same arguments. Consider the following example:

Example 4.1. Suppose we want to lex the string `aaaa` according to $R = [(\texttt{A}, [\![\texttt{a}]\!])]$. While determining the first two tokens, the call stack would include the following (for brevity, we omit the label `A` from the rules in calls to `maxpref_one`):

1.0. `lex` $[(\texttt{A}, [\![\texttt{a}]\!])]$ `aaaa`	
1.1. `maxpref_one` $[\![\texttt{a}]\!]$ `aaaa`	2.1. `lex` $[(\texttt{A}, [\![\texttt{a}]\!])]$ `aaa`
1.2. `maxpref_one` ε `aaa`	2.2. `maxpref_one` $[\![\texttt{a}]\!]$ `aaa`
1.3. `maxpref_one` $\varnothing$ `aa`	2.3. `maxpref_one` ε `aa`
1.4. `maxpref_one` $\varnothing$ `a`	2.4. `maxpref_one` $\varnothing$ `a`
1.5. `maxpref_one` $\varnothing$ λ	2.5. `maxpref_one` $\varnothing$ λ

Notice that items 2.4-2.5 are the same as items 1.4-1.5. A similar phenomenon will occur when the remaining tokens are computed. By memoizing the results of items 1.4-1.5, we can avoid many future recursive calls.

4.1 Implementation

The memo is a 2D lookup table M, where row labels are regular expressions and column labels are strings. Each entry is of type `option (option (String * String))`. The outer option indicates whether the entry is filled (`Some`) or empty (`None`). The inner option corresponds to the output type of `maxpref_one`. Therefore, if $M[r, s]$ = `Some` o and `maxpref_one` r $s = o$, we can access $M[r, s]$ instead of computing `maxpref_one` r s.

We replace `maxpref_one` with the following function:

```
M_maxpref_one : Rule -> String -> Memo
                -> Memo * option (String * String)
```

This function behaves like `maxpref_one`, except in the following ways:

1. Before making a recursive call with arguments r and s, the memoized version checks whether $M[r, s]$ is empty. If it is, the recursive call is made. If not, the function returns the table entry at $M[r, s]$.
2. After making a recursive call with arguments r and s, the memoized version updates $M[r, s]$ so that it contains the result of the recursive call.

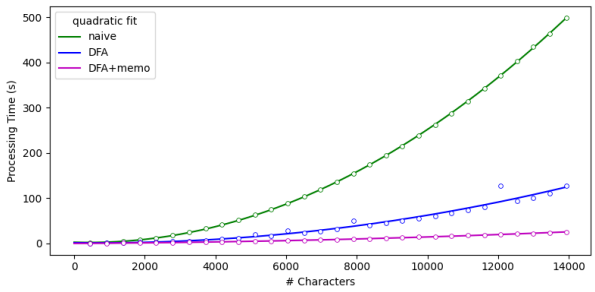


Figure 6. Performance comparison for three versions of Verbatim: the original version, the DFA-based version, and the memoized DFA-based version.

It would be insufficient to only memoize `maxpref_one`. Recall from Example 4.1 that the repeated calls occur across calls to `lex`. To take full advantage of memoization, we must also create memoized functions `M_lex` and `M_max_pref` so that the memos are passed across token computations. We create these functions by following the pattern that we used for `maxpref_one`.

4.2 Sketch of Mechanized Correctness Proof

To prove correctness, we define a property of memos called *lexiness*. We say that a memo is lexy if for all r, s, $M[r, s]$ = `None` or $M[r, s]$ = `Some` o and o = `maxpref_one` r s. We then show that lexiness is closed under `M_maxpref_one`, `M_max_pref`, and `M_lex`.

The initial memos are empty and thus trivially lexy. We know that `M_maxpref_one` operates solely on lexy memos, so the only thing left to show is that `maxpref_one` is equivalent to `M_maxpref_one` when the input memo is lexy. The result of `M_maxpref_one` is either a recursive call on a proper suffix of its input or it is the result of accessing a lexy memo. The inductive hypothesis says that the recursive call is equivalent to `maxpref_one`. By definition, the entry of the lexy memo is equivalent to `maxpref_one`. So in either case, `maxpref_one` and `M_maxpref_one` are equivalent.

The closure properties for the higher-level functions follow directly from that for `M_maxpref_one`.

4.3 Performance

With memoization, Verbatim's theoretical runtime should change. Per Reps, we might even expect the runtime to be linear. Once again, we test this hypothesis empirically.

Figure 6 shows that the "DFA + memo" version of Verbatim performs five times faster than the "DFA without memo" version, and 20 times faster than the naive implementation on the largest data point. Although the plot appears to be flatter, zooming in (Figure 7) shows that the performance is still not linear. One possible explanation is related to the fact that the Coq core library does not provide data structures that support constant-time lookup. Currently, the memo is

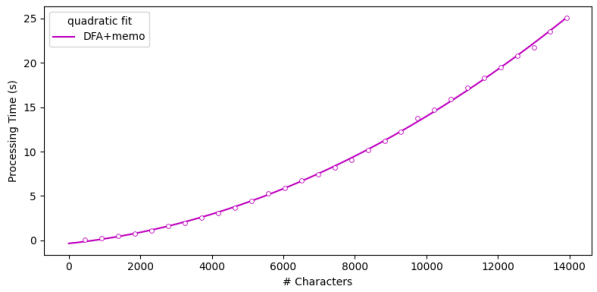


Figure 7. Performance of the memoized DFA-based lexer.

instantiated as a finite map whose keys have a total ordering. This structure supports lookups in $O(\log m)$ time, where m is the number of items in the memo. (Note that the maximum value of m grows with input size, because larger input strings have more suffixes, which correspond to more columns in the table.) Additionally, part of the memo key is a string of size $O(n)$, where n is the length of the input string. These large keys add overhead to each lookup.

5 Optimization #3: Fast Memo Lookups

In this section, we present a further optimization that greatly reduces the overhead associated with memo lookups. The optimization involves using the *lengths* of input string suffixes as memo keys, instead of the suffixes themselves. This change enables us to represent a memo with a *binary trie*—a data structure that supports more efficient lookups for our use case.

5.1 Binary Numbers as Keys

The memo stands in for the function `maxpref_one`. Because the second argument to `maxpref_one` is a string, it was natural for the second component of the memo key to be a string as well. A key realization due to Reps is that this argument is always the ith suffix of the original string, for some i.

With this correspondence in mind, we modify several components from the previous sections. Our quick memo, Q, is still a 2D lookup table with rows labeled by regular expressions, but now the columns are labeled by binary natural numbers. (Note that if the numbers were unary—Coq's default representation of naturals—the lookups would not be any faster.) Another important change to Q is that every non-empty entry contains the length of the maximal prefix, in addition to the prefix itself and its complementary suffix. Caching the length spares us from computing it after performing a lookup, which would be an expensive operation. Indeed, avoiding length computations will be a common theme in this section.

We also modify the memoized version of `maxpref_one` and name it as follows:

```
Q_maxpref_one :
 Rule -> BinNat -> String -> Memo
-> Memo * option (String * String * BinNat)
```

This function works much like `M_maxpref_one`. The primary difference is the `BinNat` elements in the signature and how they are handled. The `BinNat` input represents the length of the string input; we prove that this invariant holds in all cases where `Q_maxpref_one` is called. The `BinNat` in the output corresponds to the length of the maximal prefix returned; we prove that this invariant holds as well. In this way, we thread the lengths of input/output strings through the program. When `Q_maxpref_one` makes a recursive call it only needs to decrement the `BinNat`. These changes enable us to sidestep on-the-fly computations of string length.

We make similar changes to the signatures of the higher-level functions to obtain `Q_max_pref` and `Q_lex`, which maintain the same length invariant between strings and `BinNats`. In addition to the length invariant, we also maintain the invariant that at any point in the computation, the current string is a suffix of the original string. These two invariants together say that the current string is the ith suffix of the original string, where i is the length of the current string.

Finally, we redefine the *lexiness* property for quick memos. We say that a quick memo Q is lexy with respect to the original string z if for all i, r one of the following is true:

1. $Q[r, i] = \texttt{None}$
2. All of the following are true:
   a. $Q[r, i] = \texttt{Some}\ o$
   b. $o = \texttt{maxpref\_one}\ r\ s$
   c. s is the ith suffix of z

We then show that lexiness is closed under

- `Q_maxpref_one`
- `Q_max_pref`
- `Q_lex`

As in the previous section, these closure properties are sufficient to show that `Q_maxpref_one` and `maxpref_one` are equivalent. In turn, this fact is enough to show that the previous higher-level functions are equivalent to their quick memo counterparts.

5.2 Binary Tries as Memos

We now consider the problem of large memos. If we continue using Coq's finite map library, lookups will become more expensive as the memo grows larger. As mentioned before, lookup time would be $O(\log m)$ where m is the number of entries in the memo. This fact is problematic, as m can certainly be larger than the length of the original input string (because the string's length is equal to the number of possible suffixes, and the same suffix can appear as a key with multiple regexes). The fact that we are now using binary numbers

as keys instead of strings does not resolve this performance concern, but it does lend itself to the use of a different data structure: binary tries.

A binary trie is defined inductively in terms of an underlying element type:

```
Inductive Trie {Elt : Type} : Type :=
| Leaf
| Branch (x : option Elt) (t0 t1 : Trie).
```

It has associated getter and setter functions:

```
set_Trie : Trie -> list bool -> Elt -> Trie
get_Trie : Trie -> list bool -> option Elt
```

These functions use a list of booleans as the key. If the list is empty, then the current node is modified/returned. Otherwise, we search the left or right subtrie for the tail of the list, depending on whether the head is 0 or 1. If the necessary subtrie is not found, then we create it in the case of the setter and we return `None` in the case of the getter.

Binary trie lookups are fast: just $O(b)$ in the size of the boolean list. Equivalently, if b is the binary representation of a number n, then lookups are $O(\log n)$. In addition, each lookup is independent of unvisited nodes. This data structure therefore has the property we want: lookup time is invariant in the size of the memo. Binary tries also have the desirable property that they do not require comparisons between keys; lookups are performed simply by reading the key bit-by-bit.

The only proof obligations for this optimization involve showing that a binary trie has the traditional properties of lookup structures:

1. `get_Trie (set_Trie t b x) b = Some x`
2. `get_Trie (set_Trie t b x) b' = get_Trie t b'` (where `b` ≠ `b'`)

We prove that our binary trie implementation satisfies these properties.

5.3 Performance

Using this data structure in the development is straightforward. Abstractly, the memo and quick memo are both 2D lookup tables. Concretely, though, the slower memo is implemented as a finite map with tuple keys. We alter this design only slightly for the concrete quick memo. The quick memo is a binary trie where the element type is a finite map. The inner finite maps use regexes as keys. Admittedly, lookups in these maps are still expensive, but the sizes and number of regexes in the memo are independent of the size of the input string. We do not consider the runtime with respect to the number of lexical rules or the size of their underlying regular expressions; we treat the lexical rules as fixed in our runtime analysis. In this case, lookups in this memo are $O(\log n)$, where n is the length of the original input string. This optimization is a significant improvement over the finite map approach and is the best we can do without fixed-sized integers.

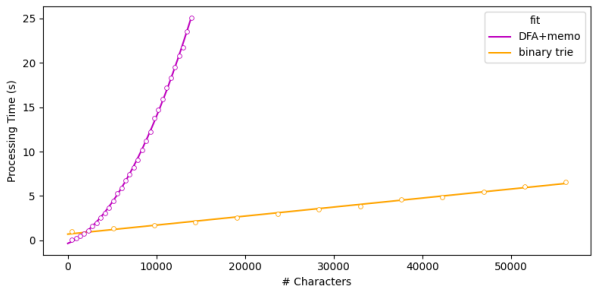


Figure 8. Performance of the binary trie version of Verbatim compared with the finite map version. The "DFA + memo" curve corresponds to the "DFA + memo" curves in Figures 6 and 7.

When analyzing the lexer's theoretical runtime complexity, it is helpful to pretend that the memo is populated eagerly (i.e., in a separate step from calls to the maximal prefix finder). In reality, the actions of adding memo entries and finding maximal prefixes are interleaved. The analysis is as follows:

Suppose we have a fully populated quick memo, Q. Then every call to the maximal prefix finder will be a memo lookup. In the worst-case scenario, there are n maximal prefixes, each of length 1. We will therefore perform n lookups and the ith lookup will cost $O(\log(n-i))$ as that is the length of the remaining suffix at that point. The overall cost will therefore be

$$O\left(\sum_{i=0}^{n-1} \log(n-i)\right) = O(\log n!) = O(n \log n)$$

Furthermore, the cost to populate Q eagerly is $O(n \cdot g(n))$, where g is the cost of a lookup, because there are $O(n)$ columns in Q and the number of rows is independent from the length of the string. As we mentioned, our lookups are $O(\log n)$, so the cost to populate Q is $O(n \log n)$.

Therefore, the overall runtime complexity of the lexer is still superlinear: $O(n \log n)$.

Empirically, this optimization produces a drastic improvement in performance. We evaluated the binary trie-based lexer on a subset of points from the original benchmark, plus a set of larger inputs to provide strong evidence of asymptotically superior performance. Figure 8 displays the results of benchmarking the version of Verbatim with a trie-based memo. The R^2 value of the linear regression for this plot is 0.94, suggesting that a linear equation effectively models Verbatim's performance in this experiment.

6 Semantic Actions

The versions of Verbatim described thus far produce tokens that are label-string pairs. The label is simply a tag indicating which lexical rule produced the token, and the string is a literal value taken from the input string. Henceforth, we

will refer to this type of token as a *literal token*. This simple token representation places an unnecessary burden on downstream parsers, which must convert the string values into values of other types to produce an expressive abstract syntax tree (AST). In this section, we describe the concept of a *semantic token*, and we extend Verbatim to output such tokens. We then discuss the correctness properties that this extension requires and that our implementation satisifies. Finally, we convert an existing "literal" JSON lexer (a lexer that produces literal tokens) to a semantic lexer, and we compare the performance of the two versions.

6.1 Semantic Tokens

Suppose Verbatim tokenizes the input string '2 + 2' as follows:

```
[ (NUM, '2'); (PLUS, '+'); (NUM, '2') ]
```

Now consider a parser tasked with converting these tokens into an AST for an arithmetic expression. Presumably, the parser does not need to know that '+' is the literal value in the PLUS token—the PLUS label conveys the fact that the expression is an addition. However, it would be preferable for the NUM leaves of the tree to contain the numeric value 2 instead of the *string* value '2' so that a downstream application can evaluate the arithmetic expression. Therefore, the following token representation is preferable:

```
[ (NUM, 2 : nat); (PLUS, () : unit); (NUM, 2 : nat) ]
```

Verbatim cannot determine such semantics a priori. Rather, the user must provide them as part of the lexical specification. Therefore, this extension to Verbatim requires the user to include a new component of the specification: a `sem_ty` (semantic type) function from label *values* to semantic *types*. For example, if the two labels in a given specification are NUM and PLUS, then the function might map NUM to `nat` and PLUS to `unit`. Note that Verbatim cannot infer this mapping automatically; for example, the user might want to map NUM to `int` instead.

The notion of a semantic type for a label enables us to define *semantic tokens*. A semantic token is a dependently typed pair

```
{x : label & sem_ty x}
```

where the first element is a label, and the second element is a semantic value with a type determined by the label.

6.2 Semantic Actions for Verbatim

In addition to defining the `sem_ty` function, the user defines a semantic action function `sem_action` that maps literal tokens to `option` semantic tokens. When provided with a literal token `t = (x, s)`, the function attempts to produce a semantic token `t' : {x & sem_ty x}` by converting `s` into a value of the appropriate semantic type. If `s` is not well-formed, the action returns None. For example, a semantic action that converts strings to integers would return None if given a string that does not represent a valid integer.

A desirable property of this scheme might be that a lexical rule's regex and the semantic actions "agree" as follows: whenever a string matches the regex, applying the action to the string will succeed (not return None). For example, if a string matches a regex that represents valid integers, then converting the string to an integer should succeed. While some users might be interested in proving this property for their lexical specifications, we do not require such a proof because it would impose a heavy burden on users who are less familiar with Coq. Instead, we opt to fail safely; a None result is enough to stop a user from passing ill-formed tokens to a parser.

Another property, which we do require the user to prove, is that labels *carry*: each token that a semantic action produces must have the same label as the original literal token. In our experience, this property is easy to prove for typical semantic actions.

Once the user provides a labels-to-type mapping, semantic actions, and a proof that labels carry, Verbatim is able to return semantic tokens instead of literal tokens. An auxiliary function `sem_lex'` computes a list of semantic tokens, possibly containing None if a literal was ill-formed. The function `sem_lex` calls `sem_lex'` and returns None if any of the tokens are None; otherwise, it simply returns the list of semantic tokens.

The function `sem_lex'` operates by first calling the literal lexer, `lex`. It then applies a semantic action to each literal token to obtain the list of semantic tokens.

6.3 Correctness

We say that a list of semantic tokens tokenizes an input string z if the following conditions are satisfied:

1. A list of literal tokens t tokenizes z (as defined previously).
2. Let s_n and l_n be the nth semantic and literal tokens, respectively. Then for all n, s_n = `sem_action` l_n, where `sem_action` is the user-defined semantic action function.

We also say that z does not have a semantic tokenization if `sem_action` l_n = None for some n.

In the development, we prove soundness and completeness for the semantic lexer. Additionally, we prove that our definition of semantic tokenization is unambiguous. Because `sem_lex'` is a simple map operation over the literal tokens, these proofs depend largely on the corresponding theorems for the literal lexer.

6.4 A Semantic JSON Lexer and Its Performance

To confirm that semantic actions can add relatively little overhead, we created a semantic version of the JSON lexer used in previous benchmarks and measured its performance.

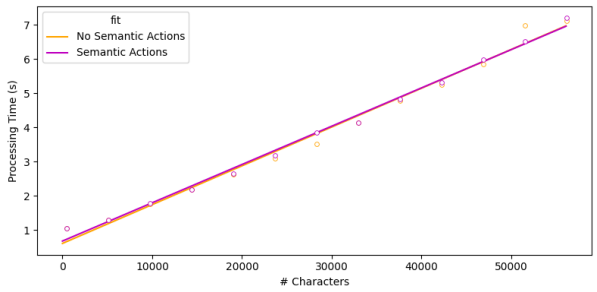


Figure 9. Performance comparison of the literal and semantic versions of Verbatim.

It is hard to make general runtime guarantees about semantic lexers because in principle, users could define semantic actions of arbitrary time complexity. The semantic actions that we use to tokenize JSON are, in the worst case, linear in the size of the token. Therefore, we should not expect these actions to add substantial overhead.

We define the `sem_ty` function for JSON as follows:

```
Definition sem_ty (l : Label) : Type :=
  match l with
  | INT => Z
  | FLOAT => Z
  | STRING => string
  | _ => unit
  end.
```

Coq's integer type is `Z`.[1] The labels mapped to `unit` include braces, brackets, white space, and keywords. The semantics of these tokens can be determined by their label.

We then define the `sem_action` function for JSON. For the most part, the actions are defined in terms of built-in Coq functions for converting strings to other types. A few edge cases required careful attention: for example, correctly handling −0 and leading zeros. An implementation that handles floats would require even finer attention to detail if it did not rely on external libraries.

Figure 9 shows that there is no significant performance difference between the literal JSON lexer and the version augmented with semantic actions.

7 Related Work

In addition to Verbatim [7], on which the current work builds, there have been several other successful efforts to produce verified lexers. Ausaf et al. [1] present a derivative-based lexer that is similar to Verbatim in its algorithmic approach and correctness specification. The tool is implemented and verified with Isabelle/HOL. The authors do not discuss the tool's theoretical complexity or empirical performance. Hardin [11] uses the HOL4 theorem prover to verify an implementation of Brzozowski's regex-to-DFA algorithm, which is incorporated into a lexer. Lopes et al. [16] present a regex matcher based on Brzozowski derivatives. The matcher takes a regular expression e and a string s as input; if s matches e, the tool produces a proof of the match. The matcher does not employ a disambiguation policy or produce labeled tokens. Nipkow [18] uses Isabelle/HOL to prove the correctness of a regex-to-DFA translation and an accompanying lexer, but the implementation does not correspond to an executable program and thus is not immediately suitable for programmatic lexing.

As for slightly different applications of Brzozowski derivatives, the RockSalt security policy checker [17] features a verified regex-to-DFA conversion algorithm. The resulting DFA is used for a recognition task rather than for lexing in the usual sense. Derivatives have also been used for checking regular expression equivalence in a verified manner [5, 13].

Our memoization optimizations (Sections 4 and 5) were inspired by Reps's technique for linear-time lexing [25].

There is an extensive literature on verified parsing and its applications [2, 24, 26–31]. In particular, our dependently typed representations of semantic tokens and semantic actions (Section 6) were inspired by similar schemes that several verified parsers employ [12, 15]. Indeed, one of our goals in choosing this representation was to make the lexer interoperable with existing verified parsers, and thereby enable users to create fully verified front ends for their applications.

8 Future Work

We envision three areas of future work: further evaluations, additional optimizations, and integration with a parser as part of a usable, verified toolchain.

8.1 Further Evaluation

This work would benefit from at least three types of experiments. The first type is evaluation of memory usage. We have drastically improved Verbatim's execution speed, but much of this improvement came from various forms of caching. It would be useful to see how these improvements affected the tool's memory efficiency, and to determine whether the tool could be improved on this front.

We would also like to compare Verbatim's performance to that of unverified state-of-the-art lexers. Understanding and bridging the differences in performance between verified tools and their unverified counterparts is an important step towards the adoption of verified software in real-world systems.

Finally, we would like to evaluate Verbatim on more benchmarks across more domains. Verbatim is a general-purpose tool; for every list of well-formed lexical rules, it produces a correct maximal munch lexer. We would like to further

[1] Our JSON lexer does not currently handle full-blown floats. Correctly parsing floats is a difficult problem with its own literature and is not a contribution of this work. Our benchmark data set does not contain floats.

explore Verbatim's performance characteristics across the range of domains that it supports.

8.2 Further Optimization

There is at least one other optimization that would improve Verbatim's runtime with respect to the size of the input string. There is also an opportunity to improve the lexer's runtime with respect to the lexical rules.

As mentioned in Section 5, the current version of Verbatim has theoretical complexity $O(n \log n)$ in the size of the input string. Reps explains how maximal munch lexing can be done in linear time [25]. The factor of $\log n$ comes from lookups in the quick memo Q. In particular, the smallest representation of the length of an input string prefix has size $\log n$. Regardless of which data structure we use, this $\log n$ factor will remain as long as we have to look up keys of this size. It turns out that we can avoid looking up such keys from scratch; every time we index into Q with key i, the next lookup is guaranteed to use key $i \pm 1$. So, we can avoid looking up these keys from scratch if we maintain a cursor in Q that moves to the next or previous column depending on the nature of the following lookup. Implementing and verifying the correctness of this optimization would result in true linear runtime.

This paper captures our focus on optimizing Verbatim with respect to the size of the input string. Indeed, this is a reasonable preoccupation given that the lexical rules are usually static for a given application. That said, there is at least one optimization that can be made to our DFA representation. We currently construct a DFA for each lexical rule and find a maximal prefix for each DFA. Prior work on Brzozowski derivatives [22] suggests a way to construct just one DFA. This change would result in constant runtime with respect to the number of lexical rules.

8.3 Integration and Usability

As mentioned, our use of semantic actions and tokens was inspired by existing verified parsers. In a related line of research, we are working on verifying semantically aware parsers that are compatible with broad classes of context-free grammars. Verbatim outputs the type of tokens that such parsers consume; we look forward to integrating Verbatim with downstream verified parsers.

Currently, the user must provide Verbatim with lexical rules in the form of Coq code. Verbatim would be more accessible if the user could provide lexical rules as plaintext input in a familiar syntax. One exciting direction we see for this project involves bootstrapping Verbatim, along with a verified parser; in other words, we could build a verified front end for Verbatim's input format. In this way, we could leverage our tools to improve their own usability.

9 Conclusion

Previous work on Verbatim focused on formalizing, implementing, and verifying a tool for performing lexical analysis. This paper expands on that work in two ways. First, we have gone to great lengths to optimize Verbatim. Lexical analysis is expected to be fast; the work described in this paper is a necessary step in making verified software more competitive with unverified state-of-the-art tools. Additionally, parsers often expect to consume tokens with an expressive type; to increase Verbatim's utility, we have given the user finer-grained control over the tool's output type by way of semantic actions. We believe this work highlights the effectiveness of an incremental verification strategy, in which an initial prototype and subsequent optimizations are verified in separate phases. We hope the work serves as a small step towards the widespread use of verified, usable, and practical language-processing pipelines.

References

[1] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Interactive Theorem Proving*. Springer International Publishing, Cham, 69–86.

[2] Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda. https://doi.org/10.2168/LMCS-12(2:6)2016

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

[4] R. G. G. Cattell. 1978. Formalization and Automatic Derivation of Code Generators.

[5] Thierry Coquand and Vincent Siles. 2011. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, 119–134. https://doi.org/10.1007/978-3-642-25379-9$_1$1

[6] Derek Egolf. 2021. Verbatim source code. https://github.com/egolf-cs/Verbatim.

[7] Derek Egolf, Sam Lasser, and Kathleen Fisher. 2021. Verbatim: A Verified Lexer Generator. In *LangSec Workshop*. https://langsec.org/spw21/papers.html#verbatim

[8] Dan Goodin. 2017. Failure to patch two-month-old bug led to massive Equifax breach. https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/.

[9] Dan Goodin. 2020. Windows has a new wormable vulnerability, and there's no patch in sight. https://arstechnica.com/information-technology/2020/03/windows-has-a-new-wormable-vulnerability-and-theres-no-patch-in-sight/.

[10] Google Project Zero 2017. cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory. https://bugs.chromium.org/p/project-zero/issues/detail?id=1139.

[11] David S Hardin. 2020. Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems. In *Proceedings of the 2020 IEEE Systems Conference*.

[12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Proceedings of the 21st European Symposium on Programming (ESOP 2012)*. Springer, 397–416. https://doi.org/10.1007/978-3-642-28869-2$_2$0

[13] Alexander Krauss and Tobias Nipkow. 2012. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *J. Autom. Reason.* 49, 1

(2012), 95–106. https://doi.org/10.1007/s10817-011-9223-4

[14] Mohit Kumar. 2020. Critical PPP Daemon Flaw Opens Most Linux Systems to Remote Hackers. https://thehackernews.com/2020/03/ppp-daemon-vulnerability.html. *The Hacker News* (2020).

[15] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2019. A Verified LL(1) Parser Generator. In *Proceedings of the 10th International Conference on Interactive Theorem Proving (ITP '19)*. 24:1–24:18. https://doi.org/10.4230/LIPIcs.ITP.2019.24

[16] Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. 2016. Certified Derivative-Based Parsing of Regular Expressions. In *Programming Languages*. Springer International Publishing, Cham, 95–109.

[17] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 395–404. https://doi.org/10.1145/2254064.2254111

[18] Tobias Nipkow. 1998. Verified Lexical Analysis. In *Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science, Vol. 1479)*, Jim Grundy and Malcolm C. Newey (Eds.). Springer, 1–15. https://doi.org/10.1007/BFb0055126

[19] NIST 2016. CVE-2016-0101. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2016-0101.

[20] NIST 2017. CVE-2017-5638. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2017-5638.

[21] NIST 2020. CVE-2020-8597. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2020-8597.

[22] Scott Owens, John H. Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

[23] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[24] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[25] Thomas Reps. 1998. "Maximal-Munch" Tokenization in Linear Time. *ACM Trans. Program. Lang. Syst.* 20, 2 (March 1998), 259–273. https://doi.org/10.1145/276393.276394

[26] Tom Ridge. 2011. Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, 103–118. https://doi.org/10.1007/978-3-642-25379-9$_1$0

[27] Sorawit Suriyakarn, Clément Pit-Claudel, Benjamin Delaware, and Adam Chlipala. 2018. Narcissus: Deriving Correct-By-Construction Decoders and Encoders from Binary Formats. *CoRR* abs/1803.04870 (2018). arXiv:1803.04870 http://arxiv.org/abs/1803.04870

[28] Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *J. Autom. Reason.* 60, 3 (2018), 257–277. https://doi.org/10.1007/s10817-017-9429-1

[29] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 413–429. https://doi.org/10.1007/978-3-319-96142-2$_2$5

[30] Marcell van Geest and Wouter Swierstra. 2017. Generic packet descriptions: verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, Sam Lindley and Brent A. Yorgey (Eds.). ACM, 30–40. https://doi.org/10.1145/3122975.3122979

[31] Ryan Wisnesky, G. Malecha, and J. G. Morrisett. 2010. Certified Web Services in Ynot.

[32] World Bank. 2020. United States annual GDP data [data file]. Retrieved from http://api.worldbank.org/v2/countries/USA/indicators/NY.GDP.MKTP.CD?per$_p$age=5000&format=json.