Verbatim: A Verified Lexer Generator

Derek Egolf
Tufts University
Medford, MA
derek.egolf@tufts.edu

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

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

***Abstract*—Lexers and parsers are often used as front ends to connect input from the outside world with the internals of a larger software system. These front ends are natural targets for attackers who wish to compromise the larger system. A formally verified tool that performs mechanized lexical analysis would render attacks on these front ends less effective.**

In this paper we present Verbatim, an executable lexer that is implemented and verified with the Coq Proof Assistant. We prove that Verbatim is correct with respect to a standard lexer specification. We also analyze its theoretical complexity and give results of an empirical performance evaluation. All correctness proofs have been mechanized in Coq.

***Keywords*-lexical analysis; interactive theorem proving**

I. Introduction

Lexers and parsers are often used as front ends to connect input from the external world with an otherwise trusted computing base. These front ends are natural targets for attackers who wish to access the internals of such computing bases. There is a plentiful pool of such attacks in the real world, many of which have exploited avoidable implementation bugs [1]–[7]. A formally verified tool that performs mechanized lexical analysis would render many of those attacks far less effective.

In this paper we present Verbatim, an executable lexer that provably conforms to a standard lexer specification. Verbatim takes as input a list of lexical rules and a string, and it uses a technique for regular expression matching based on the concept of Brzozowski derivatives [8] to tokenize the string. The resulting list of tokens satisfies a common specification known as the "maximal munch" principle: each token is the longest prefix of the remaining input string that matches a lexical rule.

The Coq development that accompanies this paper is publicly available online [9]. The development consists of roughly 700 lines of specification and 1200 lines of proof.

This paper is organized as follows. In Section 2 we provide background information on lexical analysis, the maximal munch principle, and Brzozowski derivatives. We present the Verbatim implementation in Section 3. In Section 4, we discuss how we constructed a provably terminating lexer implementation in Coq. Section 5 presents our correctness theorems and proof sketches. In Section 6, we discuss our use of proof automation in the Coq development. We analyze Verbatim's runtime complexity and present results of an empirical performance evaluation in Section 7. In Section 8, we give a brief overview of related work. Finally, in Section 9 we discuss our plans for future development.

II. Background and Specification

Lexical analysis (lexing) is the process of partitioning an input string into a list of words and assigning a label to each of these words. A label-word pair is usually called a token. A lexer performs this process with the help of lexical rules that specify how to partition the input. Given a set of lexical rules, we must specify how the lexer is to apply these rules. In particular, we must specify how the lexer partitions the input and how the lexer labels a word that matches multiple rules. In this section, we formalize regexes, regex matching, the maximal munch principle, and Brzozowski derivatives.

A. Regular Expressions

Regexes inductively denote regular languages. These expressions are natural interfaces for lexing as they are both human and machine readable. If a string z is in the language represented by regex e, we say that z matches e and write $z \simeq e$. We use the canonical, inductive definitions of regexes (Figure 1) and regex matching (Figure 2) presented in Software Foundations [10], a popular textbook on interactive theorem proving in Coq.

We differentiate the empty string λ from the empty regex ε, which denotes the language $\{\lambda\}$. Additionally, a is the string consisting solely of symbol a, while $[\![a]\!]$ is the regex that denotes $\{a\}$, the language containing only that string.

We represent a lexical rule as a label-regex pair. We say that a string z matches a rule (l, e) iff $z \simeq e$ and we write $z \simeq (l, e)$ to represent such a match.

B. Maximal Prefixes

If string $z = p +\!\!+\, s$, we say that p is a prefix of z and we write `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 $\texttt{MaxPref}_R\ p\ z$ (we formalize this definition in Figure 3).

Symbol	$a, b \in \Sigma$
String	$z ::= \lambda \mid az$
Regex	$e ::= \varnothing \mid \varepsilon \mid [\![a]\!] \mid e + e \mid e \cdot e \mid e^*$
Rule	$r ::= (z, e)$
Token	$t ::= (z, z)$

Figure 1: Definition of strings, regular expressions, lexical rules, and tokens over an alphabet Σ. For brevity, we write non-empty strings without a terminal λ. For example, we write a instead of $a\lambda$.

(MEMPTY)
$$\lambda \simeq \varepsilon$$

(MCHAR)
$$a \simeq [\![a]\!]$$

(MAPP)
$$\frac{z_1 \simeq e_1 \qquad z_2 \simeq e_2}{z_1 \mathbin{+\!\!+} z_2 \simeq e_1 \cdot e_2}$$

(MUNIONL)
$$\frac{z \simeq e_1}{z \simeq e_1 + e_2}$$

(MUNIONR)
$$\frac{z \simeq e_2}{z \simeq e_1 + e_2}$$

(MSTAR0)
$$\lambda \simeq e^*$$

(MSTARAPP)
$$\frac{z_1 \simeq e \qquad z_2 \simeq e^*}{z_1 \mathbin{+\!\!+} z_2 \simeq e^*}$$

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

C. Maximal Munch Principle

Regardless of the exact process used to lex an input string, that process ought to be unambiguous. One popular specification for unambiguous lexing is the maximal munch principle [11]. Given an input string z and a list of rules R, the maximal munch principle says that the first word of z is the maximal prefix of z with respect to R. If multiple rules in R match the maximal prefix, we label this first word according to the matching rule with the least index in R (see FIRSTTOKEN in Figure 4). If we partition z as $z = p \mathbin{+\!\!+} s$, where p is the first word of z, then we can specify the remaining tokens inductively as the tokens of s. If no maximal munch exists or the remaining suffix is empty, we specify the remaining suffix as unprocessed. (see TOKENSNIL and TOKENSCONS in Figure 4).

D. Brzozowski Derivatives

Verbatim's regex matcher uses a matching algorithm based on the concept of Brzozowski derivatives. Brzozowski derivatives are a popular approach to regex matching in the functional programming community [12], and we used them as the basis for our lexer because they allow for incremental matching on a sequence of characters.

Intuitively, the derivative of a language L with respect to symbol a chops off the a from those strings in L that begin with a and includes only the resulting suffixes. More

(PREFIX)
$$\frac{p \mathbin{+\!\!+} s = z}{\texttt{Prefix}\ p\ z}$$

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

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

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

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

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

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, while in TOKENSNIL all of z is unprocessed.

formally:

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

Brzozowski showed that this operation preserves regularity, so 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

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

Because regular expressions inductively denote regular languages, we 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 recursive algorithm 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 `nullable` r_1 evaluates to `true` if $\lambda \simeq r_1$ and `false` otherwise. We also compute `nullable` recursively:

$$\begin{aligned}
\texttt{nullable } \varnothing &:= \texttt{false} \\
\texttt{nullable } \varepsilon &:= \texttt{true} \\
\texttt{nullable } [\![b]\!] &:= \texttt{false} \\
\texttt{nullable } (r_1 + r_2) &:= \texttt{nullable } r_1 \vee \texttt{nullable } r_2 \\
\texttt{nullable } (r_1 \cdot r_2) &:= \texttt{nullable } r_1 \wedge \texttt{nullable } r_2 \\
\texttt{nullable } r^* &:= \texttt{true}
\end{aligned}$$

III. Implementation

The Verbatim implementation has three main components: a regex matcher, a maximal prefix finder, and a top-level `lex` function. Breaking the implementation into these components allowed us to prove the correctness of the development in a modular fashion.

The Verbatim matcher takes a regex and a string as input, and recursively takes derivatives of the regex for each character in the string. If the resulting regex is nullable, the matcher returns `true`, and otherwise returns `false`. We say that this matcher does *incremental matching*; while consuming a string, it checks at each character whether or not the current regex is nullable. If the current regex is nullable, the prefix consumed thus far matches the original regex.

We use this matcher to construct a function, `maxpref_one`, that finds the maximal prefix for a **single** lexical rule.

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

In addition to returning the maximal prefix, the function also returns the complementary suffix of this prefix.

Given a matcher and a string z, we could find the maximal prefix by applying the matcher to each prefix of z. The time complexity of this operation would be quadratic. Instead, `maxpref_one` exploits incremental matching and makes just one pass over the string. The last character in the string that produces a nullable regex corresponds to the last character of the maximal prefix. If no character in the string produces a nullable regex, there is no maximal prefix and the finder returns `None`. For instance,

$$\texttt{maxpref\_one bb } (\texttt{l}_1, [\![\texttt{a}]\!]) = \texttt{None}$$

because no prefix of bb matches $[\![\texttt{a}]\!]$. Specifically, $\partial_b [\![\texttt{a}]\!] = \varnothing$ and $\partial_b \varnothing = \varnothing$. Neither $\varnothing$ nor $[\![\texttt{a}]\!]$ is nullable. Because none of the encountered regexes are nullable, Verbatim recognizes that there is no maximal prefix.

We create a maximal prefix finder for a list of regexes by wrapping the singleton finder in a function called `max_pref`.

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

This function takes a string and a **list** of rules and, if possible, returns the longest prefix matching any rule, the complementary suffix, and the label associated with the earliest matching rule. Otherwise, the function returns `None` if no maximal prefix can be found for any rule.

Next, we have our workhorse function

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

which takes as input a string and the rules with which to lex that string, and returns as output a list of tokens and the unprocessed suffix of the input string. As seen in Figure 5, the `lex'` function repeatedly calls `max_pref`; each call produces a single token and a remaining suffix that serves as input to a recursive `lex'` call. The function terminates when `max_pref` returns an empty prefix or `None`.

Finally, our top-level function, `lex`, calls `lex'` with appropriate initial values.

A challenging aspect of implementing Verbatim was defining `lex'` in a provably terminating way. In the next section, we sketch the function's termination proof and discuss our Coq mechanization of this proof.

IV. Termination

To avoid logical inconsistencies, all Coq functions must terminate provably. When a function is *primitively recursive* on one of its arguments, Coq is able to infer that the function terminates. Here, "primitively recursive" means that the function calls itself on a syntactic subterm of one of its arguments.

Intuitively, the `lex'` function must terminate because it makes recursive calls on proper suffixes of the input string. However, the function obtains a suffix via a call to `max_pref`, and Coq's termination checker is unable to detect the fact that `max_pref` returns a proper suffix of its input. It was straightforward to prove that `max_pref` returns a proper suffix of its input, but it was challenging to leverage that proof to obtain a provably terminating `lex'` definition.

As depicted in Figure 6, we solved this problem with a recursion technique in which `lex'` takes as an additional

```
1   Fixpoint lex' (in_str : String) (rules : list Rule)
2
3     : (list Token) * String :=
4     match max_pref in_str rules
5
6     with
7     | (_, None) => ([], in_str)
8     | (_, Some ([], _)) => ([], in_str)
9     | (label, Some (prf_hd :: prf_tl, suffix)) =>
10
11      match lex' rules suffix
12      with
13      | (tkns, rest) =>
14        ((( label, prf_hd :: prf_tl) :: tkns), rest)
15      end
16    end.
```

Figure 5: This definition of $\texttt{lex}'$ captures the function's semantics, but it does not compile, because Coq is unable to detect the fact that `max_pref` (line 4) returns a `suffix` of its `in_str` argument that is structurally smaller than `in_str`.

```
1   Fixpoint lex' (in_str : String) (rules : list Rule)
2     (Ha : Acc lt (length in_str)) {struct Ha}
3     : (list Token) * String :=
4     match max_pref in_str rules as mpref'
5       return max_pref in_str rules = mpref' -> _
6     with
7     | (_, None) => fun _ => ([], in_str)
8     | (_, Some ([], _)) => fun _ => ([], in_str)
9     | (label, Some (prf_hd :: prf_tl, suffix)) =>
10      fun Heq =>
11        match (lex' rules suffix
12          (acc_rec_call _ _ _ _ _ _ Ha Heq)) with
13          | (tkns, rest) =>
14            ((( label, prf_hd :: prf_tl) :: tkns), rest)
15        end
16    end eq_refl.
```

Figure 6: The actual definition of $\texttt{lex}'$ includes an additional parameter, `Ha` (line 2), which is a proof that the length of the input string is accessible in the standard "less than" relation on natural numbers. In the $\texttt{lex}'$ recursive call (line 11), the `acc_rec_call` function (line 12) constructs a proof that the length of the `suffix` argument is accessible in the same relation. This proof term is structurally smaller than `Ha`; therefore, $\texttt{lex}'$ is structurally recursive on its accessibility proof parameter.

parameter a proof that the string's length is *accessible* in a well-founded relation. A well-founded relation is one that contains no infinite descending sequences. For example, the standard $<$ relation on natural numbers is well-founded because a descending sequence from any natural number must eventually end at zero. Informally, an element x is accessible in a well-founded relation R if every element $y < x$ is also accessible. Note that the least element in R is "trivially" accessible according to this definition; for example, zero is accessible in $<$ because no natural number is less than zero.

The concept of accessibility enables us to define $\texttt{lex}'$ in a provably terminating way. Suppose that string z' is a proper suffix of z, and $\texttt{Acc}_<(z)$ is a proof that the length of z is accessible in the $<$ relation. In this case, one can obtain a corresponding proof term for z', $\texttt{Acc}_<(z')$, that is a syntactic subterm of $\texttt{Acc}_<(z)$. We take advantage of this fact by adding an accessibility proof term as a parameter to $\texttt{lex}'$; the function becomes structurally recursive on this proof term.

V. Correctness

We prove the following properties of Verbatim and its specification:

1) **Soundness**: If Verbatim produces a tokenization for its input, then that tokenization is correct according to our 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.

A. Soundness

Our soundness theorem says that the output of the top-level `lex` function partitions and labels the input correctly.

$$\begin{aligned}
&\forall t\ u\ z\ R,\\
&\texttt{lex}\ R\ z = (t,\ u)\\
&\land\ \texttt{rules\_are\_unambiguous}\ R\\
&\to \texttt{Tokens}_R\ (t,\ u)\ z
\end{aligned}$$

Here z is the string to be lexed, R is a list of lexical rules, t is the list of tokens that `lex` produces, and u is the unprocessed suffix of z.

The predicate `rules_are_unambiguous` holds when the rules map each label to exactly one regex. We introduced this constraint to make the proof more tractable. Without the constraint, it becomes much more difficult to show that the lexer correctly handles certain syntactically valid but unintuitive lists of rules. For instance, consider the following rules:

$$R = [(\texttt{l}_1,\ [\![\texttt{a}]\!]^*);\ (\texttt{l}_2,\ [\![\texttt{b}]\!]^*);\ (\texttt{l}_1,\ [\![\texttt{b}]\!])]$$

The input string `b` matches both the second and third rules. This string ought to be labeled $\texttt{l}_2$ because $(\texttt{l}_2,\ [\![\texttt{b}]\!]^*)$ appears earlier and our disambiguation strategy prefers earlier rules. But the presence of a later matching rule, $(\texttt{l}_1,\ [\![\texttt{b}]\!])$, which has the same label as an earlier non-matching rule, $(\texttt{l}_1,\ [\![\texttt{a}]\!]^*)$, complicates the proof. We would have to show that in cases like this one, Verbatim chooses the label of the earliest matching rule—not the earliest label associated with *any* matching rule. This kind of rule list is unlikely to appear in practice, so we opted to disallow it. This requirement does not limit the expressiveness of the lexer—if the user would like multiple regexes to have the same label, they can simply union the regexes.

The proof of this theorem goes by strong induction on the length of the input string. The primary difficulty is showing that the first token produced by $\texttt{lex}'$ really is the first token of the input, as defined by `FirstToken`. Although it is relatively easy to show that the word of the first token is a maximal prefix, it is harder to show that $\texttt{lex}'$ gives the word a correct label. This difficulty lies in the fact that we must label the word according to the least-indexed (earliest) matching rule. In Section 6, we discuss our proof of "sound labeling" further.

B. *Uniqueness*

The Uniqueness lemma says that according to our specification, there is only one way to correctly lex a string with a given list of rules. This result is important in and of itself because unambiguity is a cornerstone of lexing, but it is also important because, when paired with Soundness, it proves the Completeness of our lexer. The lemma is as follows:

$$\begin{aligned}&\forall t\ u\ t'\ u'\ z\ R,\\&\texttt{Tokens}_R\ (t,\ u)\ z\\&\wedge\ \texttt{Tokens}_R\ (t',\ u')\ z\\&\rightarrow (t,\ u) = (t',\ u')\end{aligned}$$

Because $\texttt{Tokens}_R$ is defined inductively on the first token, this lemma follows from the fact that the first token is unique. Intuitively, we know that the first token is unique because there is only one maximal prefix and because tokens are labeled according to the earliest matching rule.

In the proof development, we suppose that the `FirstToken` property holds for tokens $(l,\ w)$ and $(l',\ w')$. We then show that w and w' are of the same length and are prefixes of the same string and hence $w = w'$. We then establish that the rules associated with l and l' are either the same or that one is earlier than the other. In the first case $l = l'$ and we're done. In the latter case we derive a contradiction: according to the `FirstToken` definition, the higher-indexed rule cannot match the maximal prefix, but both rules match w. Hence $(l,\ w) = (l',\ w')$ and the first token is unique.

C. *Completeness*

This theorem says that the output of `lex` is uniquely correct.

$$\begin{aligned}&\forall t\ u\ z\ R,\\&\texttt{Tokens}_R\ (t,\ u)\ z\\&\wedge\ \texttt{rules\_are\_unambiguous}\ R\\&\rightarrow \texttt{lex}\ R\ z = (t,\ u)\end{aligned}$$

This theorem follows directly from Soundness and Uniqueness. Suppose

1) $\texttt{Tokens}_R\ (t,\ u)\ z$
2) $\texttt{rules\_are\_unambiguous}\ R$
3) $\texttt{lex}\ R\ z = (t',\ u')$, for some $t',\ u'$

Through Soundness, we can conclude from (2) and (3) that

$$\texttt{Tokens}_R\ (t',\ u')\ z$$

Then by Uniqueness and (1), we can conclude that

$$(t',\ u') = (t,\ u)$$

Then by substitution into (3) we have

$$\texttt{lex}\ R\ z = (t,\ u)$$

Hence Completeness follows from Soundness and Uniqueness.

VI. Proof Automation

In this section, we describe how we made judicious use of Coq's proof search facilities to prove a difficult lemma that requires intensive case analysis.

Coq is a tool for proving theorems interactively. The interactivity manifests as a window containing a set of hypotheses and a goal, which represent the premises and conclusion of a theorem, respectively. The user manipulates the state of the goal and the hypotheses by applying "tactics" until the goal is trivially true or until one of the hypotheses is false. In either case, Coq has constructed a proof of the theorem.

Proofs often involve many, potentially nested, subproofs. For instance, if a hypothesis contains an if-then-else expression, we might use the *destruct* tactic to do case analysis on the branches of that expression. Each branch produces a new hypothesis in place of the prior one as well as a new subgoal. We would then have to prove each of the new subgoals.

Coq provides support for semi-automated proof search; users can write custom tactics that analyze the current hypotheses and goal in order to determine how to manipulate the proof state. We take advantage of this facility in our proof of sound labeling (first mentioned in Section V-A).

The lemma is as follows: On a list of rules R and an input string z, suppose that the function `max_pref` returns a label l, prefix p, and suffix s. Additionally, suppose that p is the maximal prefix of z with respect to some rule $(l,\ e)$ in R. We wish to show that for all $(l',\ e')$ in R, if $(l',\ e')$ comes before $(l,\ e)$, then p is not the maximal prefix of z with respect to $(l',\ e')$.

This lemma may have been the most challenging in the entire development. The proof is as follows. Because $(l',\ e')$ appears before $(l,\ e)$ in R, we know that R has the following general form:

$$R = R_1 +\!\!+ [(l',\ e')] +\!\!+ R_2 +\!\!+ [(l,\ e)] +\!\!+ R_3$$

where R_1 consists of the elements before $(l',\ e')$, R_2 consists of the elements between $(l',\ e')$ and $(l,\ e)$, and R_3 consists of the elements after $(l,\ e)$.

When `max_pref` is applied to the partition above, the function must decide which of the five sublists contains the rule that produces the maximal prefix. Applying `max_pref` to the partition produces an expression that contains many if-then-else clauses. Each condition of these clauses imposes a constraint on the length of a possible maximal prefix. We use a custom tactic to do case analysis on all of these if-then-else clauses. Because there are many nested clauses, the analysis produces 625 subgoals, each of which comes with a different set of constraints on the length of the maximal prefix.

Recall, though, that we assumed the maximal prefix is p. We can therefore check p against the constraints that the case analysis generates. In all cases, the constraints force one of two conclusions: either (1) p is not the maximal prefix of z with respect to $(l',\ e')$, or (2) at least one constraint is inconsistent with our assumption that p is the maximal prefix. In case (1), we have proven the conclusion of our original lemma. In case (2), there is a contradiction and thus the implication is vacuously true.

Because of the large number of subproofs required for this lemma, we rely heavily on automation to make the problem tractable. We are able to solve all 625 subproofs using just nine distinct tactics.

For example, many subgoals (465/625) have the following form:

$$\frac{\begin{array}{c}(l_1, e_1) = (l_2, e_2)\\ (l_2, e_2) = (l_3, e_3) \qquad \ldots \qquad (l_{n-1}, e_{n-1}) = (l_n, e_n)\\ \texttt{max\_pref}\ z\ R = (l_1, o) \qquad \texttt{max\_pref}\ z\ R \neq (l_n, o)\end{array}}{\bot}$$

Since the conclusion is $\bot$ (false), we must derive a contradiction in the hypotheses. We solve this category of subgoals in part with a custom tactic `inj_all`, which takes hypotheses like $(l_1, e_1) = (l_2, e_2)$ and produces the hypotheses $l_1 = l_2$ and $e_1 = e_2$. If we apply this tactic repeatedly, we arrive at the fact that $l_1 = l_n$. After substitution, the proof state is

$$\frac{\texttt{max\_pref}\ z\ R = (l_1, o) \qquad \texttt{max\_pref}\ z\ R \neq (l_1, o)}{\bot}$$

At this point, we can finish the subgoal by deriving a contradiction from the hypotheses.

Another sizable portion of subgoals (92/625) have this form:

$$\frac{\begin{array}{c}\texttt{len}\ p_0 < \texttt{len}\ p_1\\ (x, \texttt{max\_pref\_one}\ z\ r_0) = (x', \texttt{Some}(p_0, s_0))\\ (y, \texttt{max\_pref\_one}\ z\ r_1) = (y', \texttt{Some}(p_1, s_1))\\ \texttt{max\_pref\_one}\ z\ r_0 = \texttt{Some}(p, s)\\ \texttt{max\_pref\_one}\ z\ r_1 = \texttt{Some}(p, s)\end{array}}{\bot}$$

After substituting and applying `inj_all`, we reach the following state:

$$\frac{\texttt{len}\ p_0 < \texttt{len}\ p_1 \qquad p = p_0 \qquad p = p_1}{\bot}$$

From this state, we can derive the hypothesis $\texttt{len}\ p < \texttt{len}\ p$, which is a contradiction.

The final category of subgoals (68/625) have the following form:

$$\frac{\begin{array}{c}(x, \texttt{max\_pref\_one}\ z\ r) = (x', \texttt{None})\\ \texttt{max\_pref\_one}\ z\ r = \texttt{Some}\ y\end{array}}{\bot}$$

By substituting and applying `inj_all`, we produce the contradictory hypothesis $\texttt{Some}\ y = \texttt{None}$.

None of these subgoals were particularly difficult conceptually. The problem was that there were hundreds of trivial subgoals. Once we identified the categories of subgoals described above, we were able to solve them using proof automation.

VII. Performance Evaluation

For a specific list of lexical rules, Verbatim has quadratic theoretical time complexity with respect to the length of the input string. Even if the maximal prefix is short, Verbatim must scan the entire input in order to rule out longer possibilities.

For example, consider Verbatim's behavior on the singleton list of rules $R = [(\texttt{A}, [\![\texttt{a}]\!])]$ and the input string aaa. To find the first token, it will perform the following calculations:

1) $\partial_{\texttt{a}}[\![\texttt{a}]\!] = \varepsilon$, which is nullable. Therefore, a might be the maximal prefix.
2) $\partial_{\texttt{a}}\varepsilon = \varnothing$, which is not nullable. Therefore, aa cannot be the maximal prefix.
3) $\partial_{\texttt{a}} = \varnothing$, which is not nullable. Therefore, aaa cannot be the maximal prefix.

Therefore, a is the first maximal prefix and $(\texttt{A}, \texttt{a})$ is the first token. Verbatim will then find the second token, repeating steps 1 and 2. To find the third token, it will repeat step 1. In this case there were 3 + 2 + 1, operations, but if the input string were $\texttt{a}^n$, Verbatim would perform $n + (n-1) + ... + 2 + 1$ operations.

We know then that the runtime is $O(tn)$ where t is the number of tokens produced. In the worst case, each token is a single character, so $t = n$. Therefore, the overall theoretical complexity is $O(n^2)$.

To confirm this complexity empirically, we extracted the Verbatim source code to OCaml, instantiated a JSON lexer,[1] and evaluated its performance. The data set used in the evaluation was a collection of gross domestic product (GDP)

[1]The lexer currently does not support escape sequences, such as those for Unicode characters. Escape sequences do not appear in our evaluation data set.

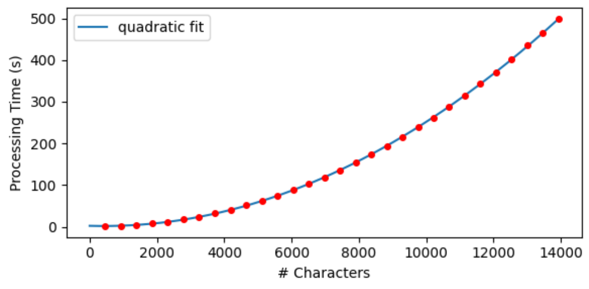

Figure 7: Verbatim execution time on JSON inputs. Each point represents the average execution time over five trials; there was little variance across trials for a given input.

statistics [13] formatted as a JSON list. This list contains 60 entries. We took 30 prefixes of this dataset, each prefix containing two more GDP entries than the previous one. We ran Verbatim on each of these prefixes five times and recorded the lexer's execution time for each trial.

We ran the evaluation on a laptop with 15.3 GB of RAM, 8 1.8 GHz cores, and Ubuntu 18.04. We used the 4.11.0+flambda version of OCaml and compiled with the -O3 flag.

The results of our empirical tests, depicted in Figure 7, confirm that Verbatim has $O(n^2)$ time complexity. A quadratic regression returned a correlation coefficient of 0.99996. Because a coefficient close to 1 indicates a good fit, a quadratic function accurately models the empirical results.

VIII. Related Work

Most related work falls into two categories: automata theory and lexical analysis. Much of the work on verified lexical analysis and regex matching resides outside of the Coq literature. Within automata theory, work that uses Brzozowski derivatives to convert regexes to DFAs is of particular interest, as adding this component is one of our possible next steps.

In terms of automata theory, Almeida et al. [14] present a derivative-based function for computing the *support* of a regex: a set of regexes that can be used to produce a non-deterministic finite automaton (NFA) for the original one. The authors implement this function in Coq and prove its correctness. They are not concerned with lexical analysis in particular, and they have not produced an NFA that could be used in a matcher or a lexer.

Coquand and Siles [15] discuss Brzozowski derivatives in the context of proving regex equivalence. The authors give a detailed exposition of the finiteness of Brzozowski's construction in Coq. They show that a regex has only a finite number of derivatives up to a notion of similarity. Because derivatives correspond to states in Brzozowski's DFA construction, their finiteness result could help prove the termination of this construction.

As for lexical analysis, the RockSalt security policy checker for native code [16] includes a verified regex-to-DFA construction based on Brzozowski derivatives. The authors use this construction to produce a recognizer rather than a lexer. The x86 grammar they are interested in is unambiguous, so they do not need to employ a disambiguation strategy such as the maximal munch principle.

Hardin [17] describes a large project that includes a regex-to-DFA construction based on Brzozowski derivatives, implemented in HOL4. The algorithm is proved correct and used as part of a lexer. The paper does not describe the lexer in great detail, so it is difficult to compare the work to Verbatim. However, the termination and correctness arguments are likely to be quite different from ours, because HOL4 and Coq are based on different underlying logics.

Lopes et al. [18] discuss a Brzozowski derivative-based matcher. The authors implemented a function that takes as input a regular expression e and a string s, and outputs a proof that s matches e (in the case that they do match). Their tool does not produce labeled tokens and does not employ a disambiguation strategy.

Ausaf et al. [19] discuss a lexer that is based on Brzozowski derivatives and implemented with Isabelle/HOL. This tool is similar in scope to Verbatim. Like Verbatim, the tool matches regexes without using intermediate automata. Whereas Verbatim uses a list of lexical rules as input to the top-level lexer, this tool uses a disjunction of regular expressions. Despite this difference in representation, both tools use the maximal munch principle and prefer "earlier" rules for disambiguation. In the interest of error reporting, Verbatim is capable of partial lexing. The tool discussed in this paper returns `None` if it is unable to lex the string completely. Both tools produce labeled tokens. The authors of this paper do not discuss the theoretical runtime complexity of their tool, nor do they discuss any empirical results.

Nipkow [20] formalized the conversion of regular expressions to deterministic finite automata (DFAs) and verified a lexer in Isabelle/HOL, but he did not produce an executable program suitable for lexing.

IX. Future Directions

Although we have an executable lexer that satisfies a widely-used correctness specification, its underlying algorithm is somewhat naive. Matching directly on regexes using Brzozowski derivatives can be expensive. It would be more efficient to match using a DFA. The cost of building the DFA would be comparable to computing the Brzozowski derivatives during runtime, but a DFA can be computed before lexing. Throughout this work, we strove to keep the lexer agnostic to the implementation of the matcher, so swapping in a DFA-based matcher should be relatively straightforward.

By swapping in a DFA-based matcher, we could also achieve linear worst-case complexity through memoization

[21]. As it stands, we are scanning almost the entire input for every token. In the worst case, there will be many cases where we start at some state (or regex) q and consume the same suffix s, never touching an accepting state (or nullable regex) along the way. In this case, we know that the state-suffix pair (q, s) will never produce a longer prefix. It is possible to keep track of these non-productive pairs and short circuit the prefix search when the lexer reaches one of them. Reps shows that this memoization technique enables lexers to achieve linear worst-case complexity.

Finally, we are interested in combining our lexer with a verified parser, such as the LL(1) parser that Lasser et al. [22] present. The resulting pipeline would vet program inputs in a fully verified manner, helping to defend systems against malicious input.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1934553. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. Sam Lasser's research is supported by a Draper Fellowship.

References

[1] "Cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory," https://bugs.chromium.org/p/project-zero/issues/detail?id=1139, 2017.

[2] "CVE-2017-5638," National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2017-5638, 2017.

[3] D. Goodin, "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/, 2017.

[4] "CVE-2016-0101," National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2016-0101, 2016.

[5] "CVE-2020-8597," National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2020-8597, 2020.

[6] D. Goodin, "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/, 2020.

[7] M. Kumar, "Critical PPP Daemon Flaw Opens Most Linux Systems to Remote Hackers," The Hacker News, 2020.

[8] J. A. Brzozowski, "Derivatives of Regular Expressions," J. ACM, vol. 11, no. 4, pp. 481–494, 1964. [Online]. Available: https://doi.org/10.1145/321239.321249

[9] D. Egolf, "Verbatim source code," https://github.com/egolf-cs/vlg, 2021.

[10] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, Logical Foundations, ser. Software Foundations series, volume 1. Electronic textbook, May 2018, version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[11] R. G. G. Cattell, "Formalization and Automatic Derivation of Code Generators," Ph.D. dissertation, Pittsburgh, PA, USA, 1978.

[12] S. Owens, J. H. Reppy, and A. Turon, "Regular-expression derivatives re-examined," J. Funct. Program., vol. 19, no. 2, pp. 173–190, 2009. [Online]. Available: https://doi.org/10.1017/S0956796808007090

[13] World Bank, "United States annual GDP data [data file]," Retrieved from http://api.worldbank.org/v2/countries/USA/indicators / NY.GDP.MKTP.CD?per_page = 5000&format = json, 2020.

[14] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa, "Partial Derivative Automata Formalized in Coq," in Implementation and Application of Automata, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 59–68.

[15] T. Coquand and V. Siles, "A Decision Procedure for Regular Expression Equivalence in Type Theory," in Certified Programs and Proofs, ser. Lecture Notes in Computer Science, J.-P. Jouannaud and Z. Shao, Eds., vol. 7086. Springer, Berlin, Heidelberg, 2011, pp. 119–134.

[16] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan, "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, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 395–404. [Online]. Available: https://doi.org/10.1145/2254064.2254111

[17] D. S. Hardin, "Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems," in Proceedings of the 2020 IEEE Systems Conference (to appear), 2020.

[18] R. Lopes, R. Ribeiro, and C. Camarão, "Certified Derivative-Based Parsing of Regular Expressions," in Programming Languages, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 95–109.

[19] F. Ausaf, R. Dyckhoff, and C. Urban, "POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl)," in Interactive Theorem Proving, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 69–86.

[20] T. Nipkow, "Verified Lexical Analysis," in Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings, ser. Lecture Notes in Computer Science, J. Grundy and M. C. Newey, Eds., vol. 1479. Springer, 1998, pp. 1–15. [Online]. Available: https://doi.org/10.1007/BFb0055126

[21] T. Reps, "“Maximal-Munch” Tokenization in Linear Time," ACM Trans. Program. Lang. Syst., vol. 20, no. 2, pp. 259–273, Mar. 1998. [Online]. Available: https://doi.org/10.1145/276393.276394

[22] S. Lasser, C. Casinghino, K. Fisher, and C. Roux, "A Verified LL(1) Parser Generator," in 10th International Conference on Interactive Theorem Proving (ITP 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.