Coqlex: Generating Formally Verified Lexers

Wendlasida Ouedraogo, Lutz Strassburger, Gabriel Scherer

Coqlex: Generating Formally Verified Lexers

Wendlasida Ouedraogo
Siemens Mobility
France

Lutz Straßburger
Inria Saclay
France

Gabriel Scherer
Inria Saclay
France

Abstract

A compiler consists of a sequence of phases going from lexical analysis to code generation. Ideally, the formal verification of a compiler should include the formal verification of each component of the tool-chain. The Compcert project, a formally verified C compiler, comes with associated tools and proofs that allow to formally verify most of those components. However, some components, in particular the lexer, remain unverified. In order to contribute to the end-to-end verification of compilers, we implemented a verified lexer generator whose usage is similar to OCamllex. Our software, called *Coqlex*, reads a lexer specification and generates a lexer equipped with Coq proofs of its correctness. It provides a formally verified implementation of most features that standard lexer generators (that are not formally verified) usually have. We also give a performance evaluation, comparing Coqlex to OCamllex and Verbatim++.

Keywords Lexer, Tokenizer, Regexp, Coq, Formal verification, OCamllex, Verbatim++, Compilers, lexical analyzers

ACM Reference format:
Wendlasida Ouedraogo, Lutz Straßburger, and Gabriel Scherer. 2022. Coqlex: Generating Formally Verified Lexers. In *Proceedings of , , (Preprint),* 12 pages.
DOI:

1 Introduction

A lexer is a tool that is in charge of the lexical analysis, one of the first phases of compilers and interpreters. Lexers take a sequence of characters (such as source code or command) as input and produce a sequence of tokens (parts of that input sequence of characters associated with meaning) that can be easily processed by parsers. During that process, lexers can ignore comments or white spaces, and also equip tokens with source position information (such as line numbers) to enable useful error messages during lexical analysis (lexing), parsing or later compilation stages.

Implementing a lexer from scratch can be difficult and time-consuming. This has lead researchers to build tools, libraries and generators to help implementing optimized lexers. Most of the existing implementations of those libraries and generators, such as OCamllex [19], do not come with a formal proof of correctness.

Preprint,
2022. . . . $0
DOI:

This is the starting point for our work on *Coqlex*, a formally verified lexer generator. Our goal was to provide a tool that is as versatile as OCamllex and that at the same time is formally verified, so that it can be integrated into formally verified compiler tool-chains, such as Compcert [12].

The main issues with the formal verification of tools such as lexers are related to (i) the execution time, (ii) the integration with existing parsers and (iii) the usability. This document presents techniques we used to tackle those challenges. Our contributions are as follows:

1. **The verification of lexical rule selection.** Most lexer generators produce lexers from lexer specification files. Those specifications defines a lexer using *lexing rules* that are pairs of input patterns, defined via regular expressions [9, 21] (regexp), and semantic actions that are in charge of production tokens. Depending on the selection policy and the text to analyse, the lexer selects a pair by analysing its regexps. When a pair is selected, the token to produce is handled by its semantic action. Coqlex implements two selection policies (the longest match and the shortest match associated to the priority rules) and provides the Coq proof of their correctness.
2. **The *Coqlex generator*.** Coqlex also provides a small preprocessor (the *Coqlex generator*) that lets users specify lexers in user-friendly syntax, inspired by the one of OCamllex. There is no simple specification for this input syntax itself. Instead, the *Coqlex generator* translates it to a human-readable Coq file with (heavier notation but) the same structure, and the correctness statement is given in terms of this translated source. This is similar to the "Coq production" mode of the Menhir parser generator [8].

After having finished this work, we discovered the independent research on the Verbatim++ [5, 6] formally verified lexer. For this reason, we will here also include a

3. **Comparison with other lexer generators.** More precisely, we compare Coqlex to OCamllex (because it is a standard tool) and to Verbatim++ (because it is the only other formally verified lexer generator we are aware of), with respect to execution time and usability.

In OCamllex, lexical rules are compiled into a non-deterministic automaton represented by a compact table of transitions, with backtracking and semantic actions. The generated lexer code simply follows the automaton transition by repeated table lookups. In contrast, Coqlex contains no such

compilation, it interprets the user-provided regular expressions against the input by using Brzozowski derivatives [3]. This allows to have a simpler formalization, and leads to surprisingly good performance: roughly 100x slower than OCamllex, but more than 10x faster than Verbatim++. This is reasonable for a pure program extracted directly from Coq, compared to an efficient implementation, and is more than fast enough in practice. For example, the lexer of the *Coqlex generator* is implemented in Coq, using the Coqlex data structures and functions. That generator is used without any noticeable slowness.

The Verbatim++ lexer (also verified in Coq) implements regexps using Brzozowski derivatives[3] and then compiles those regexps into deterministic finite automata[2] for fast regexp matching. Even though significant emphasis in the work [5, 6] is put on optimization, Verbatim++ remains substantially slower than Coqlex in our experiments (which used the benchmarks provided by Verbatim++).

Our work highlights the fact that a simple model and implementation can contribute sensitively to the reduction of execution time, leading to good performance. It also provides a complete lexer generator (a software) and library that allow to implement lexers easily in Coq and then extract them into OCaml code. Those lexers, written in Coq, come with proven lemmas that allow developers to prove specific properties on them. In addition, associated with menhir[16] verified parsers, Coqlex verified lexers allow to write fully formally verified front-ends for formally verified compilers such as Compcert[12].

This paper is organized as follows: In section 2 we discuss the representation of a lexer in Coq. In section 3 we present the *Coqlex generator* and discuss its specification and correctness in section 4. Section 5 presents implementation details of Coqlex. Section 6 compares the features and performance of Coqlex with OCamllex and Verbatim++. Finally, we discuss future work and conclude in section 7.

2 Representing a lexer in Coq

From a functional point of view, lexers are in charge of producing tokens (user-defined type that we will note `T`) from text (`string`). A natural type would be

```
lexer(T) := string -> list T
```

Instead of processing a list of tokens, most parsers like ocamlyacc[19] produce tokens one by one, and are called by the parser on demand. A function that performs *one step* of lexical analysis (lexing) consumes a string and returns a token and the remaining string. The type of such a function is

```
lex1(T) := string -> T * string
```

Lexing can fail for various reasons. In case of failure, lexers should provide useful error messages. For that reason, we defined a position data type and an error data type to encapsulate the lexing result. A function that performs one step of lexing becomes a function that takes an input string, a start position and in case of success, returns a token, the remaining string and the end position. Consequently, the type of one step of lexing becomes

```
lex1(T) :=  string ->
            position ->
            Result(T * string * position)
```

Most lexer generators generate lexers using a set of lexical rules that are regular expressions[9, 21] (regexp) associated with semantic actions that are in charge of producing the lexing result. The semantic action that will produce the returned lexing result is the first one associated to the regexp that matches the longest prefix of the input string (the lexeme): this is the longest match and the priority rules. Semantic actions have access to the lexing buffer (lexbuf), a data structure containing the lexeme, the start position (the position of the first letter of the lexeme), the end position (the position of the letter after the last letter of the lexeme) and the remaining string (the input string without the lexeme). The semantic action also specifies how the internal state of the lexer (at type `S`) should be updated. So, a natural type for semantic actions would be:

```
action(T) :=  lexbuf ->
              Result(T * string * position)
```

Those semantic actions can perform various operation, including recursive calls to the lexer that calls them. This could then lead to an infinite loop.[1] As Coq forbids the implementation of functions that loop[4], Coqlex had to find a solution to deal with those kinds of situations. We explored two possibilities:

1. Making restrictions on semantic actions that ensure termination. For example we could require that each semantic action discards at least one character from the input string.
2. Using the fuel technique: this technique consists into ensuring the termination of lexers using a natural number (`nat`) that decreases at every recursive call.

Requiring that each semantic action discards at least one input character is too strict in practice. Studying lexers in the wild, we have found many cases of lexers designed to "skip" an optional part of the input, that accept the empty string if nothing needs to be skipped. For example, the lexer of the OCaml compiler contains the following lexer:

```
rule skip_hash_bang = parse
  | "#!" [^ '\n']* '\n' { new_line lexbuf }
  | "" { () } (* accepts the empty string *)
```

We thus chose to express general, potentially non-terminating lexers using fuel. Consequently, the type of one step of lexing

[1] Section 6.1 provides a typical OCamllex example of a lexer that can loop due to recursive calls.

becomes

```
lex1(T) := nat ->
           lexbuf ->
           Result(T * string * position)
```

To make it simple to call a lexer from a semantic action, we replace the separate arguments `string` and `position` by the more informative `lexbuf` type already used by semantic actions.

```
lex1(T) := nat ->
           action(T)

action(T) := lexbuf ->
             Result(T * lexbuf)
```

3 Coqlex in practice

Coqlex comes with a Coq library that allow to write lexers using sets of lexical rules. It also provides a text processor that will convert a markup language (.vl syntax), that is similar to the OCamllex[19] specification language (.mll syntax), into its equivalent Coq code (.v file). Figure 1 presents the .vl version of the mini-cal (a micro language for arithmetic expressions : numbers, idents, + * - / and parentheses) lexer. This .vl definition has four parts:

1. The header section: The header section is arbitrary Coq text enclosed in curly braces. If present, the header text is copied as it is at the beginning of the output file. Typically, the header section contains the Coq `Require Import` directives, possibly some auxiliary functions and token definitions used for lexer definitions.
2. The regexp definition section: This section allows to give names to frequently-occurring regular expressions. This is done using the syntax `let` *ident* `=` *re* to associate the name *ident* to the regexp *re*. The syntax of regexp is defined in Figure 2.
3. The lexer definition section: This section allows to define lexers using sets of rules. A rule is defined using the syntax `|` *p* {*a*} (the '|' symbol is not mandatory for the first rule) to associate the pattern *p* to the Coq text representing a semantic action *a*. This pattern is either a regexp or a `string -> bool` function (defined using the syntax `$(`*f*`)` where *f* is the Coq code of this function). Typically, this kind of pattern is used to detect situations in which the lexing must stop (e.g when the input string is empty). When the pattern is a regexp, the semantic rule is said to be regexp based. Otherwise, the semantic rule is said to be function based.
4. The trailer section: This section is similar to the header section, except that its text is copied as it is at the end of the output file. Typically, this section contains Coq extraction directives.

```
(* header section *)
{
Require Import TokenDefinition.
}

(* regexp definitions *)
let ident = ['a'-'z']+
let numb = ['0'-'9']+

(* lexer definitions*)
rule minlexer = parse
| '\n' { sequence [new_line; minlexer] }
| ident {ret_l ID}
| numb { ret_l Number }
| '+' { ret PLUS }
| '-' { ret MINUS }
| '*' { ret TIMES }
| '(' { ret LPAREN }
| ')' { ret RPAREN }
| eof { ret Eof }
| _ { raise_l "unknown token :"}

(* trailer section *)
{}
```

Figure 1. mini-cal.vl file

re ::=	
'*c*'	Character constant
| "*string*"	String constant
| _	Char wildcard
| [$s_1 s_2 ... s_n$]	Union of character sets
| [^$s_1 s_2 ... s_n$]	Union of negation of character sets
| $re_1 \mid re_2$	Alternative
| $re_1\ re_2$	Concatenation
| $re_1 - re_2$	Difference
| *re*∗	Kleene star
| *re*+	Strict repetition
| *re*?	Option
s ::=	
'*c*'	Character constant
| 'c_1' - 'c_2'	Character range

Figure 2. Syntax of Coqlex regexps

Remarks:

- A .vl file allows to define multiple lexers. Those lexers are gathered in groups (made of mutually recursive

```
Require Import TokenDefinition.
Definition ident := Cat ((CharRange  "a"%char   "z"%char ))
        (Star ((CharRange  "a"%char   "z"%char ))).

Definition numb := Cat ((CharRange  "0"%char   "9"%char ))
        (Star ((CharRange  "0"%char   "9"%char ))).

Fixpoint minlexer {Storage: Set} fuel lexbuf storage
{struct fuel} := match fuel with
| 0 => (AnalysisNoFuel lexbuf, storage)
| S n => (match generalizing_elector
    (Action := semantic_action (Storage := Storage))
     LexerDefinition.longest_match_elector (
     [(Char  "010"%char ,  sequence [new_line; (minlexer n)]);
      (ident, ret_l ID);
      (numb,  ret_l Number );
      (Char  "+"%char ,  ret PLUS );
      (Char  "-"%char ,  ret MINUS );
      (Char  "*"%char ,  ret TIMES );
      (Char  "("%char ,  ret LPAREN );
      (Char  ")"%char ,  ret RPAREN );
      (RValues.regex_any,  raise_l "unknown token : ")],
      [(CoqlexUtils.EOF,  ret Eof )]) (remaining_str lexbuf) with
    | Some elt => exec_sem_action elt lexbuf storage
    | None => (AnalysisFailedEmptyToken lexbuf, storage)
    end)
end.
```

Figure 3. mini-cal.v

lexers) using the keyword *and*. To define non mutually recursive lexers, the user must use the keyword *then* instead.

- *Coqlex generator* users do not need to worry about the management of fuel when writing .vl files.
- For each lexer defined in the .vl file, the *Coqlex generator* produces a lexer function with same name.

Using the code in Figure 1, the *Coqlex generator* outputs the Coq code in Figure 3. Typically, the generator translates the regexp written in .vl syntax into the Coqlex regexp data type. The generated lexing function also calls Coqlex functions such as `generalizing_elector`, `longest_match_elector` and `exec_sem_action`. It also recurses over the fuel explicitly; we could instead generate a call to a fixpoint combinator, but this would be difficult to scale to mutually-recursive lexers.

Remark: Similarly to OCamllex, Coqlex also allows to choose the semantic action by matching the shortest prefix. In that case, the function `longest_match_elector` is replaced by the function `shortest_match_elector`.

4 *Coqlex generator* specification

Given a set of regexp-based rules l_r, a set of function based rules l_f, and a matching policy e, the generated Coq code implements a lexer — a function that takes a fuel n_f, lexbuf b, a storage s and returns a lexing result — that works as follows:

1. If n_f is equal to 0, then the result is an error. This error is a direct consequence of the fuel technique.
2. Otherwise, from the input string, l_r, l_f and e, the lexer chooses a rule whose semantic action will be in charge of returning the lexing result.
   a. if the selected rule is a function-based one, made up with a function f associated with a semantic action a, then there is no consumption. Consequently, the lexing result is the result of a called with b and s.
   b. if the selected rule is a regexp-based rule c – made up with a regexp r associated with a semantic action a – and if the length of the prefix matched by r using the policy e is a natural number n, then the lexing result is the result of a applied with the updated lexbuf b_u and the input storage s. The updated lexbuf is defined as follows:
      - the lexeme of b_u is the n first characters of the input string.
      - the remaining string of b_u is the input string without the lexeme.
      - the end position of $b_u$is the end position of b where the column number is incremented by n.
      - the start position of b_u is the end position of b.
   c. if no rule is selected, the lexer must return an error meaning that the input string contains elements that cannot be analysed by the lexer.

Except for the use of fuel, the functioning of the generated lexer defined above is standard.

A .vl file provides the description of lexers using lexical rules. That description is processed by the *Coqlex lexer generator* whose architecture is detailed in Figure 4. It has three components:

1. The lexer, that is in charge of generating a set of tokens from the text of the .vl file, is written in Coq using the Coqlex library and is formally verified.
2. The parser, that is in charge of generating an abstract representation from the set of token produced by the lexer, is implemented using menhir[16] with `--coq` switch to generate verified parsers.
3. The code printer, that is in charge of generating the .v file from the abstract representation produced by the parser, is written in OCaml and is not formally verified.

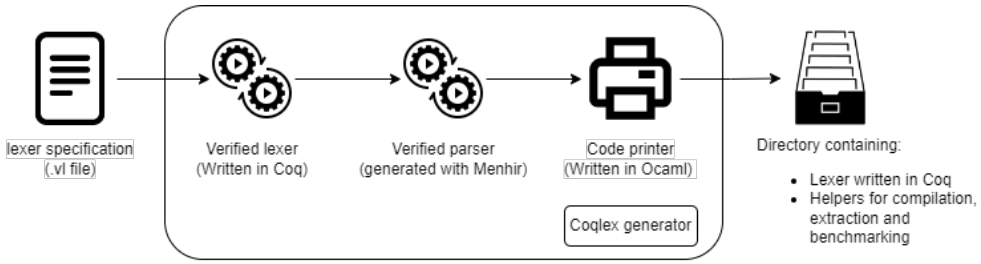


Figure 4. General structure of the *Coqlex lexer generator*.

The code printer does not include formal semantics equivalence between the representation of the .vl code and the generated .v code. This means that, *a priori*, a critical user should review the generated .v code. This does not take great efforts because the transformation does not include a complex compilation process: the .vl and .v files have similar structures and are human readable.

By comparison, Verbatim++ does not provide such generation tool, and OCamllex generates an OCaml code in which the patterns of the lexical rules are compiled into a non deterministic automaton[2] represented by a compact table of transitions, making the generated code non human readable.

In our case, the regexps, rule selection and associated policies used in the generated file are implemented and proved correct in Coq. Consequently, the attention of the critical user who wants to check the generated .v file must be focused on the following elements:

- The translation of regexps: The user must be assured of the correspondence between the regexps written in the input .vl files and those generated in the output.v files. This requires to read and understand the regexps constructors that will be defined in Section 5.
- The matching policy: The user has to make sure that matching policy corresponds to the one that is described in the .vl file. The keyword `parse` must correspond to `longest_match_elector` and `shortest` must correspond to `shortest_match_elector`.
- For every lexer, the user must be assured that the right regexps are associated to the right semantic actions and in the same order. In the Coq code, a difference is made between lexical rules made up with regexps associated with semantic actions (regexp-based rules) and those made up with `string -> bool` functions associated with semantic actions (function-based rules).

In a nutshell, a potential user has to (i) review the Coq implementation and verification of regexps, the rule selection together with the associated policies and helpers, that are written and proved in Coq, once; and (ii) either review the code printer of the *Coqlex generator* once, or review the elements listed above at every generation.

$$
\begin{array}{lll}
regex ::= & & \\
\quad \varnothing_r & \text{The empty regexp} & \\
 & L(\varnothing_r) = \varnothing & \\
\quad | \ \epsilon_r & \text{The empty string regexp} & \\
 & L(\epsilon_r) = \{\epsilon\} & \\
\quad | \ [[a]] & \text{The one-symbol regexp } (a \in \mathbb{A}) & \\
 & L([[a]]) = \{a\} & \\
\quad | \ e_1 + e_2 & \text{The alternative} & \\
 & L(e_1 + e_2) = L(e_1) \cup L(e_1) & \\
\quad | \ e_1 \cdot e_2 & \text{The concatenation} & \\
 & L(e_1 \cdot e_2) = \{s_1 +\!\!+ s_2 | s_1 \in L(e_1) \wedge s_2 \in L(e_2)\} & \\
\quad | \ e^* & \text{The Kleene star} & \\
 & L(e^*) = \{s^n | s \in L(e) \wedge n \in \mathbb{N}\} &
\end{array}
$$

Figure 5. Definition of regular expressions associated with the language they describe. Variables e, e_1 and e_2 are regular expression. The symbol $\{a\}$ denotes the set containing a unique string that is made up with a unique symbol which is a.

5 Coqlex implementation details

Most lexer generators such as OCamllex speed up lexical analysis by compiling lexical rules into finite automata during lexer generation. In Coqlex, lexical rules are interpreted on the fly, using Brzozowski derivatives[3] for regexps and simple functions for matching policies.

5.1 Brzozowski derivatives for regexps matching

Given an alphabet (set of symbols or characters) $\mathbb{A}$, the symbol ϵ that refers to the empty string, the operator $+\!\!+$ that refers to string concatenation, the notation s^n (with $n \in \mathbb{N}$) that refers to the concatenation of n copies of the string s and the notation $L(r)$ that refers to the language described by the regexp r, we can provide an inductive definition of regexp constructions as described in Figure 5.

Using all the notation above, we say that a regular expression r matches a string s if $s \in L(r)$. Similarly, when $s \notin L(r)$ we say that r does not match s.

Coqlex uses regexp constructions and matching algorithms based on the concept of Brzozowski derivatives. This concept

$$
\begin{aligned}
&\texttt{nullable}\ \varnothing_r = \texttt{false}\\
&\texttt{nullable}\ \epsilon_r = \texttt{true}\\
&\texttt{nullable}\ [[a]] = \texttt{false}\\
&\texttt{nullable}\ (e_1 + e_2) = \texttt{nullable}\ e_1 \vee \texttt{nullable}\ e_2\\
&\texttt{nullable}\ (e_1 \cdot e_2) = \texttt{nullable}\ e_1 \wedge \texttt{nullable}\ e_2\\
&\texttt{nullable}\ e^* = \texttt{true}
\end{aligned}
$$

Figure 6. Definition of the `nullable` function. The variable a stands for a symbol and variables e, e_1 and e_2 for regular expressions.

$$
\begin{aligned}
\varnothing_r / c &= \varnothing_r\\
\epsilon_r / c &= \varnothing_r\\
[\![\mathrm{a}]\!]\ /\ \mathrm{c} &= \begin{cases}\epsilon \text{ if } a = c\\ \varnothing_r \text{ otherwise}\end{cases}\\
(e_1 + e_2)/c &= (e_1/c) + (e_2/c)\\
(e_1 \cdot e_2)/c &= \begin{cases}(e_1/c \cdot e_2) + e_2/c \text{ if } \texttt{nullable}\ e_1 = \texttt{true}\\ (e_1/c \cdot e_2) \text{ otherwise}\end{cases}\\
e^*/c &= (e/c) \cdot e^*
\end{aligned}
$$

Figure 7. Definition of the derivative of a regexp. The variables a and c stand for symbols and variables e, e_1 and e_2 for regular expression.

$$
\begin{aligned}
r\|\epsilon &= r\\
r\|az &= (r/a)\|z
\end{aligned}
$$

Figure 8. Extension of the derivative of a regexp to strings. Variables r, ϵ, a and z denote, respectively, a regex, the empty string, a symbol and a string. The operator / refers to the derivative operation described in Figure 7. The notation az denotes the string composed of the symbol a as first element and the string z.

introduces two functions: the `nullable` function and the derivative of a regexp.

The `nullable` function takes a regexp r and returns the boolean `true` if r matches ϵ (the empty string) and `false` otherwise. Its inductive definition is given in Figure 6.

Using the notation az to denote the string built from the symbol a as first element and the string z, the derivative of a regular expression r by a symbol a is the regexp r/a that denotes the language $\{z | az \in L(r)\}$. Its inductive definition is given in Figure 7.

Brzozowski [3] extended the derivative operation to strings (denoted by $\|$) as described in Figure 8, and showed that for every regular expression r and every string s

$$s \in L(r) \iff \texttt{nullable}\ (r\|s) = \texttt{true}$$

Coqlex uses an existing Coq implementation[13] of Brzozowski derivatives for regexp matching. That implementation provides a Coq proof showing that this Brzozowski derivative implementation is a Kleene algebra[1, 10] and defines an equivalence relation ($\equiv$) for regexps whose formal definition is $e_0 \equiv e_1 \iff L(e_0) = L(e_1)$. It also provides additional regex constructors such as the conjunction and negation constructors that are not used in the regexp constructors that are provided by the *Coqlex generator* (see Figure 2). On the other hand, some of the constructions of regexps presented in Figure 2 are missing. For this reason, we modified the existing Coq implementation [13] of Brzozowski derivatives as follows:

1. We removed the conjunction and negation regexp constructors
2. We added four regexp constructors:
   - **the char wildcard:** The notation ω_r denotes a regexp that matches any 1-length-string. This regexp is defined by the following two properties: $\texttt{nullable}\ \omega_r = \texttt{false}$ and for all symbol s, $\omega_r/c = \epsilon$. Then, we proved that for all strings s, we have $s \in L(\omega_r)$ if and only if s consists of a single character.
   - **the character set:** The notation Σ_l^u (where l and u are symbols) denotes a regexp whose language is $L(\Sigma_l^u) = \{c | l \leq c \wedge c \leq u \wedge c \in \mathbb{A}\}$ (where $\leq$ is a reflexive, anti-symmetric and transitive order relation on symbols). This constructor is defined using the following two properties: $\texttt{nullable}\ \Sigma_l^u = \texttt{false}$ and for all symbol c
     $$\Sigma_l^u/c = \begin{cases}\epsilon_r \text{ if } l \leq c \wedge c \leq u\\ \varnothing_r \text{ otherwise}\end{cases}$$
     We proved that if $\neg(l \leq u)$, then $\Sigma_l^u \equiv \varnothing_r$ and that for every string s, $s \in L(\Sigma_l^u)$ if and only if s consists of only one symbol c such that $l \leq c \wedge c \leq u$.
   - **the negation of character set:** The notation $\overline{\Sigma_l^u}$ (where l and u are symbols) denotes a regexp whose language is $L(\overline{\Sigma_l^u}) = \{c | \neg(l \leq c \wedge c \leq u) \wedge c \in \mathbb{A}\}$. This constructor is defined using the following two properties: $\texttt{nullable}\ \overline{\Sigma_l^u} = \texttt{false}$ and for all symbol c
     $$\overline{\Sigma_l^u}/c = \begin{cases}\epsilon_r \text{ if } \neg(l \leq c \wedge c \leq u)\\ \varnothing_r \text{ otherwise}\end{cases}$$
     We proved that if $\neg(l \leq u)$, then $\overline{\Sigma_l^u} \equiv \omega_r$ and that for every string s, $s \in L(\overline{\Sigma_l^u})$ if and only if s consists of only one symbol c such that $\neg(l \leq c \wedge c \leq u)$.
   - **the difference:** The notation $e_1 - e_2$ (where e_1 and e_2 are regexps) denotes a regexp whose language is $L(e_1 - e_2) = \{s | s \in L(e_1) \wedge s \notin L(e_2)\}$. This construction is defined using the following

$$\frac{}{E_f([], s) = \bot}$$

$$\frac{f\ s = \texttt{true}}{E_f((f, a) :: t, s) = (f, a)} \qquad \frac{f\ s = \texttt{false}}{E_f((f, a) :: t, s) = E_f(t, s)}$$

Figure 9. The formal description of the selection of a function based-rule. This description uses the list notation: `[]` denotes the empty list and $h :: t$ denotes a list whose first element is h and whose tail is t. The symbol $\bot$ means that no rule is selected.

two properties: $\texttt{nullable}\ e_1 - e_2 = (\texttt{nullable}\ e_1) \wedge \neg(\texttt{nullable}\ e_2)$ and for all symbol c, $(e_1 - e_2)/c = e_1/c - e_2/c$. We proved that for all strings s, we have $s \in L(e_1 - e_2) \iff s \in L(e_1) \wedge s \notin L(e_2)$.

These constructors have also been added for performance reasons. In fact, the regexp $\Sigma_{c_n}^{c_{n+m}}$ could be written as $[[c_n]] + [[c_{n+1}]] + ... + [[c_{n+m}]]$. However, using the first representation ($\Sigma_{c_n}^{c_{n+m}}$), the derivation function will perform 2 comparisons, while the second one will perform $m + 1$ comparisons (see Figure 7).

5.2 Matching policies

Coqlex defines two types of rules: the function based and the regexp based ones. During the lexical analysis, the generated lexer has to select a rule. This selection starts by the choice of a function based rule (noted E_f). This function based rule selection, whose formal definition is given in Figure 9, consists of finding the first rule that is made of a function whose application with the input string returns true.

If no such function based rule is found, then lexer has to choose a regexp based rule.

Most lexers perform regexp based rule election using a longest match selection policy based on the longest match and priority rules. That selection policy allows to select the first lexical rule whose regexp matches the longest prefix of the input string.

The Coqlex definition of this policy uses two concepts:

prefix: A string p is said to be a prefix of a string s if and only if there exists a string s' such that $s = p +\!\!+ s'$. For example ϵ is a prefix of any string.

***l-score*:** Given a regexp r, a string s and a natural number n, we say that the l-score of r on s is n (we write this as $\mathbb{S}_l(r, s) = n$) if and only if the length of longest prefix of s that r can match is n. For example, the *l-score* of $[[a]]^*$ in ‘aabaaaa‘ is 2 as the longest prefix of ‘aabaaaa‘ that $[[a]]^*$ can match is ‘aa‘ whose length is 2. There exists cases where there is no score (e.g: $\mathbb{S}_l([[a]], \text{‘}bac\text{‘})$). In that case, we note $\mathbb{S}_l(r, s) = -\infty$.

The inductive definition of our implementation of *l-score* computation is given in Figure 10.

$$\frac{\texttt{nullable}\ r = \texttt{true}}{\mathbb{S}_l(r, \epsilon) = 0} \qquad \frac{\texttt{nullable}\ r = \texttt{false}}{\mathbb{S}_l(r, \epsilon) = -\infty}$$

$$\frac{\mathbb{S}_l(r/a, z) = n}{\mathbb{S}_l(r, az) = n + 1} \qquad \frac{\mathbb{S}_l(r/a, z) = -\infty \quad \texttt{nullable}\ r = \texttt{true}}{\mathbb{S}_l(r, az) = 0}$$

$$\frac{\mathbb{S}_l(r/a, z) = -\infty \quad \texttt{nullable}\ r = \texttt{false}}{\mathbb{S}_l(r, az) = -\infty}$$

Figure 10. The formal description of *l-score* computation.

To prove the correctness of $\mathbb{S}_l$, we used the Coq substring function of Coq string module[20] to define the prefix. This function takes two natural numbers n m and a string s and returns the substring of length m of s that starts at position n denoted by $\delta_n^m(s)$. Here, the position of the first character is 0. If n is greater than the length $|s|$ of s then ϵ is returned. If $m > (|s| - n)$, then $\delta_n^m(s) = \delta_n^{|s|-n}(s)$. Consequently, if $m \leq |s|$, then $\delta_0^m(s)$ is the prefix of length m of s. For all strings s and regexps r, we provided Coq proofs of the following theorems:

1. if there exists a natural number n such that $\mathbb{S}_l(r, s) = n$, then $n \leq |s|$. This helps to make sure that n can be used with δ to extract the prefix of length n.
2. if there exists a natural number n such that $\mathbb{S}_l(r, s) = n$, then $\delta_0^n \in L(r)$. This means that the input regexp matches the prefix of length n of s.
3. if there exists a natural number n such that $\mathbb{S}_l(r, s) = n$, then for all m such that $n < m \leq |s|$, $\delta_0^m(s) \notin L(r)$. This means that *l-score* is maximal. Therefore, there exists no prefix of length higher than n that r can match.
4. $\mathbb{S}_l(r, s) = -\infty$ if and only if for all natural number m, $\delta_0^m(s) \notin L(r)$.

Properties 1, 2 and 3 show that $\mathbb{S}_l$ is correct. This means that if a score is returned, this score is the length of the longest prefix of the input string that the input regexp can match. Property 4 shows the completeness and the soundness of $\mathbb{S}_l$. This means that if no score is found, then there is no score, and if there exists a score, $\mathbb{S}_l$ will return it.

Using $\mathbb{S}_l$, the longest match policy (noted E_l) consists in choosing the regexp based rule whose regexp has the highest *l-score*. The Coqlex formal definition of that policy is defined in Figure 11.

To prove the correctness of E_l, we proved with Coq that for every string s and list l_r of regexp based rules:

1. if $E_l(l_r, s) = \bot$ then for every regexp r and semantic action a such that $(r, a) \in l_r$, $\mathbb{S}_l(r, s) = -\infty$
2. if there exists a regexp r, a semantic action a and a natural number n such that $E_l(l_r, s) = (r, a, n)$ then for every regexp r', semantic action a' and natural number n' such that $(r', a') \in l_r$ and $\mathbb{S}_l(r', s) = n'$, $n' \leq n$

$$\frac{}{E_l([],s) = \bot} \qquad \frac{\mathbb{S}_l(r,s) = -\infty}{E_l((r,a) :: t, s) = E_l(t,s)}$$

$$\frac{\mathbb{S}_l(r,s) = n \qquad E_l(t,s) = (r_t, a_t, n_t) \qquad n_t > n}{E_l((r,a) :: t, s) = (r_t, a_t, n_t)}$$

$$\frac{\mathbb{S}_l(r,s) = n \qquad E_l(t,s) = \bot}{E_l((r,a) :: t, s) = (r,a,n)}$$

$$\frac{\mathbb{S}_l(r,s) = n \qquad E_l(t,s) = (r_t, a_t, n_t) \qquad n_t \leq n}{E_l((r,a) :: t, s) = (r,a,n)}$$

Figure 11. The formal description of the longest match selection policy. The symbol $\bot$ means that no rule is selected.

$$\frac{\texttt{nullable } r \texttt{ = true}}{\mathbb{S}_s(r,s) = 0} \qquad \frac{\texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r,\epsilon) = \infty}$$

$$\frac{\mathbb{S}_s(r/a, z) = n \qquad \texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r, az) = n + 1}$$

$$\frac{\mathbb{S}_s(r/a, z) = \infty \qquad \texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r, az) = \infty}$$

Figure 12. The formal description of *s-score* computation.

3. for every regexps r and r', semantic actions a and a' and natural number n, if $E_l(l_r, s) = (r, a, n)$ and $\mathbb{S}_l(r', s) = n$ then $E_l((r', a') :: l_r, s) = (r', a', n)$

Besides the longest match policy, Coqlex defines the shortest match policy that allows to select the first regexp based rules whose regexp matches the shortest prefix of the input string. The implementation technique of the shortest match policy is similar to the longest match policy. This implementation starts by the definition of the **s-score** (noted: $\mathbb{S}_s$) that allows to compute the length of the shortest prefix that a regexp can match. The formal definition of **s-score** is described in Figure 12.

Similarly to the $\mathbb{S}_l$, we proved the correctness and completeness of $\mathbb{S}_s$ through Coq proofs of the following theorems:

1. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then $n \leq |s|$. This helps to make sure that n can be used with δ to extract the prefix of length n.
2. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then $\delta_0^n \in L(r)$. This means that the input regexp matches the prefix of length n of s.
3. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then for all m such that $m < n$, $\delta_0^m(s) \notin L(r)$. This means that *s-score* is minimal. Therefore, there exists no prefix of length lower than n that r can match.
4. $\mathbb{S}_s(r,s) = \infty$ if and only if for all natural number m, $\delta_0^m(s) \notin L(r)$.

$$\frac{}{E_s([],s) = \bot} \qquad \frac{\mathbb{S}_s(r,s) = \infty}{E_s((r,a) :: t, s) = E_s(t,s)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = (r_t, a_t, n_t) \qquad n_t < n}{E_s((r,a) :: t, s) = (r_t, a_t, n_t)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = \bot}{E_s((r,a) :: t, s) = (r,a,n)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = (r_t, a_t, n_t) \qquad n_t \geq n}{E_s((r,a) :: t, s) = (r,a,n)}$$

Figure 13. The formal description of the shortest match selection policy.

Using $\mathbb{S}_s$, the shortest match policy consists into choosing the regexp based rule whose regexp has the lowest *s-score*. The Coqlex formal definition that policy is defined in Figure 13.

5.3 Coqlex rule selection

Using E_f, E_l and E_s, the formal definition of the rule selection E can be defined as follows:

$$E(E', l_r, l_f, s) = \begin{cases} E_f(l_f, s) \text{ if } E_f(l_f, s) \neq \bot \\ E'(l_r, s) \text{ otherwise} \end{cases}$$

where E' is either E_l or E_s. In the Coq code presented in Figure 3, E is represented by `generalizing_elector`, E_l is represented by `longest_match_elector`. The implementation of E_s in the Coqlex library is represented by `shortest_match_elector`.

5.4 Optimization

The naive implementation suggested by the formal definition of $\mathbb{S}_l$ and E_l has a time complexity that is at least quadratic in the size of the input string. In fact, the implementation of *l-score* requires reading all the characters of the input string for every regexp based lexical rule and thus for every token. However, this is not necessary in some cases (e.g for all string s with $\mathbb{S}_l(\varnothing_r, s) = -\infty$).

To increase the performance of $\mathbb{S}_l$, $\mathbb{S}_s$, E_l and E_s, we implemented a regexp simplification function which is based on the following properties:

The alternative: $r + \varnothing_r \equiv r$ and $\varnothing_r + r \equiv r$
The concatenation: $r \cdot \varnothing_r \equiv \varnothing_r$, $\varnothing_r \cdot r \equiv \varnothing_r$, $r \cdot \epsilon_r \equiv r$, $\epsilon_r \cdot r \equiv r$ and $r^* \cdot r^* \equiv r^*$
The Kleene star: $\varnothing_r^* \equiv \epsilon_r$, $(r^*)^* \equiv r^*$ and $\epsilon_r^* \equiv \epsilon_r$
The difference: $r - \varnothing_r \equiv r$ and $\varnothing_r - r \equiv \varnothing_r$

These simplifications aim to detect when a given regexp is equivalent to a regexp whose score is trivial (e.g $\varnothing_r$ or ϵ_r). We proved these properties in Coq and then used the smart constructor technique[7] to write an optimized version of the regexp derivative function. That function works similarly

to the original one, except that it returns a simplified version of the derivative. Then, we also rewrote the **s-score** and **l-score** functions to use the optimized version of the regexp derivative function and return the result for trivial cases. For example, we proved that for every r and s,

$$\mathbb{S}_s(r^*, s) = \mathbb{S}_s(\epsilon_r, s) = \mathbb{S}_l(\epsilon_r, s) = 0$$
$$\mathbb{S}_s(\varnothing_r, s) = \infty \qquad \mathbb{S}_l(\varnothing_r, s) = -\infty$$

We proved that the optimized **s-score** and **l-score** are equal to the original ones. We propagated this optimization to E_l and E_s and obtained performance in linear time in the size of the input string (see Section 6 below). In fact, during lexical analysis, the number of tokens is high, and the number of symbol reads required to produce each of them can be low. Before those optimizations, the *l-score* computation required to read the whole input string to produce a token, whereas in the optimized version we stop reading when the score is trivial, typically when the input string is equivalent to $\varnothing_r$ or ϵ_r. This allows to reduce drastically the number of character read and so the execution time.

6 Evaluation

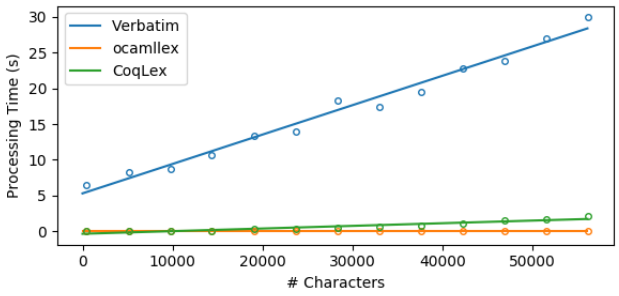
We are now going to compare Coqlex with OCamllex, the OCaml standard lexer generator, and Verbatim++, the only other research work on lexer verification that we are aware of.

First, we noticed three conceptual differences between Verbatim++, Coqlex and OCamllex:

1. Verbatim++ lexers are functions that take a input string and return a list of tokens. This means that either those lexers perform the full lexical analysis of the input string, or return nothing (failure). In Coqlex and OCamllex, lexers return a token (see Section 2). This means that even if the lexer cannot perform the full lexical analysis of the input string, Coqlex can provide a partial lexical analysis. It also allows to compute the next token only if it is needed.
2. Verbatim++ uses the notion of label, a data-type returned after the election. Semantic actions are functions that take that label and lexeme to return a token. Therefore, semantic actions do not have access on the remaining string and thus, cannot perform recursive calls. A consequence of this is that Verbatim++ lexers cannot ignore parts of the input string (such as comments and extra spaces).
3. In Coqlex, regexps are interpreted on the fly while Verbatim++ and OCamllex compile them into finite automata [2] for fast regexps matching.

Second, we evaluated the execution time of the generated lexical analysers in two phases. For the first phase, we evaluated their performance on the Verbatim++ JSON benchmark.

We started with analysing a JSON lexer implemented by the Verbatim++ developers using Verbatim++ Coq source code, then we used Coqlex and OCamllex generators to generate lexers with very close specifications. We compared the time performance and noticed a huge difference between the OCamllex and Coqlex generated lexers and the Verbatim++ lexer as presented in Figure 14.



	Verbatim++	Coqlex	OCamllex
Tokens per sec.	1.5×10^3	2.2×10^4	2.8×10^7
Time to process 50ko.	30 s	2 s	1.8×10^{-3} s

Figure 14. Comparison of execution time in seconds for Coqlex, OCamllex and Verbatim++ lexers on Verbatim++ JSON benchmark. The benchmark file contains 56154 characters and its lexical analysis should return 17424 tokens.

	Coqlex	OCamllex
Time to process 1.6Mo MiniML	0.45 s	0.04 s
Time to process 1.6Mo JSON	1.5 s	0.03 s

Figure 15. Comparison of execution time in seconds for Coqlex and OCamllex lexers on Miniml and JSON benchmark. The Miniml benchmark file contains 1599999 characters (for 28800 tokens) and the JSON benchmark file contains 1620948 characters (for 160489 tokens).

The analysis of this Figure shows linear performance for all three lexers. Generally, lexers implemented using Verbatim++ components are around 15 times slower than those generated using Coqlex and OCamllex. A similar study with XML files showed equivalent results.

For the second phase of the evaluation of the execution time, we evaluated the performance of Coqlex and OCamllex generated lexers by implementing the lexer of 2 languages: JSON[15, 18] and the first version of MiniML[17], a toy subset of OCaml. We could not perform an evaluation of those languages with Verbatim++ because their definitions imply recursive calls, a feature that is not handled by Verbatim++. The results of those evaluations are presented in Figure 15.

Generally, Coqlex executes faster when the number of characters divided by the number of token is higher. In fact, in our measurements, Coqlex has better performance for the MiniML analysis where the number of characters per token is ≈55.55 (see Figure 15). For a similar number of characters, this performance is three times slower for the JSON benchmark where the number of characters per token is five times lower (≈10.10).

We can observe that OCamllex generated lexers execute faster than Coqlex ones. However, Coqlex generated lexer performance is surprisingly good and does not pose limitations to its usefulness in real-world settings. In fact, Coqlex has been used to generate the lexer for an Ada-to-Ada optimizing compiler by Siemens mobility. This compiler is used in an industrial setting to process thousands of source code files with not a too noticeable difference with respect to OCamllex. Furthermore, the use of the *Coqlex generator*, whose lexer is implemented using the components of the Coqlex library, does not show noticeable slowness.

In regard to regexp specification, Coqlex allows to generate richer regular expressions than Verbatim++ and OCamllex. In fact, OCamllex allows to perform the regexp minus-operation only for charsets, while Coqlex allows to perform this operation on general regexps. In addition, Coqlex also allows to define function based rules other than `end-of-file`. Verbatim++ does not allow such operations. However, OCamllex allows to bind substrings matched by a regexp to identifiers, but neither Coqlex or Verbatim++ have this feature.

In regard to the syntax of the .vl files, the *Coqlex generator* is built to process a language that is very close to OCamllex. This means that there are only few differences between .vl files and their equivalent .mll files. For example, Figures 18 in the Appendix presents the OCamllex equivalent of the Coqlex lexer presented in Figure 1.

6.1 Looping lexers

Another advantage of Coqlex is that it provides protections against infinite loops. Let us consider the Coqlex lexer specified in Figures 16 and the OCamllex lexer specified in 17. Regarding those specifications, the lexers are supposed to work as follows:

- If the remaining string is ϵ then 1 is returned.
- Else if the longest prefix of the input string in `lexbuf` matches $[\![b]\!] \cdot [\![a]\!]^* \cdot [\![b]\!]$ then 0 is returned
- Else if it matches $[\![a]\!]^*$ it performs a recursive call on the remaining string of `lexbuf` (updated after the election). This is a common technique used to ignore elements such as comments during lexical analysis.

When such lexer is called with a string s that starts with a character that is different from ‘a‘ and ‘b‘, the election choose the semantic action that is associated to the regex $[\![a]\!]^*$ with a score of 0. This means that the lexeme is ϵ and the remaining string is s. As the semantic action associated to this regexp is a recursive call, it leads to an infinite loop. The lexer generated by OCamllex from the code in Figure 17 loops when the input string is ‘c‘, whereas the lexer generated by Coqlex from the code in Figure 16 returns an error. Verbatim++ does not handle this kind of problems because semantic actions do not allow recursive calls.

Furthermore, the simplicity of the Coqlex implementation allows to write proofs on Coqlex lexers. For example,

```
rule my_lexer = parse
    |  'b' 'a'* 'b' { ret 0 }
    |  'a'* { my_lexer }
    | EOF { ret 1 }
```

Figure 16. The Coqlex example of a lexer whose execution can loop

```
rule my_lexer = parse
  'b' 'a'* 'b'        { 0 }
  | 'a'*            { my_lexer lexbuf }
  | EOF             { 1 }
```

Figure 17. The OCamllex example of a lexer whose execution can loop

we have proven that the looping lexer defined in Figure 16 always returns an error related to the fuel when the first character of the input string is different from ‘a‘ and ‘b‘.

7 Conclusion

The formal correctness of lexing does not seem to be extensively studied in the literature. For instance, even for the formally proven compiler CompCert [12], lexing is one of the phases which are not formally verified. In existing approaches to verify lexers, like in CakeML [11], the lexer is implemented by hand (without using a generator) and proven equal to a simple and deterministic function. Most lexers are more complicated and it can be hard to find a simple and deterministic function that is equal to the lexer. Nipkow[14] proved the correctness of a regex-to-DFA translation and an accompanying lexer, but the implementation is not immediately suitable for programmatic lexing because that implementation does not correspond to an executable program. Only Coqlex and Verbatim++ suggest almost complete verified approach for lexer implementation and verification.

This work showed that implementing a lexer using simple data types and functions can have better performance than compiling lexical rules into deterministic finite automata (e.g Verbatim++ vs Coqlex performance). The low performance of Verbatim++ compared to OCamllex that also uses deterministic automata are due to the way the automata is stored. In OCamllex, the automata is computed one and stored using built-in data-types while in Verbatim++, that automata is computed at every lexical analysis.

In future work, the performance of our generator could be enhanced in several ways, for instance by speeding up

the election process by using finite automata as in OCamllex, allowing users to bind substrings matched by regexps, implementing a static analyzer to detect potential looping cases, implementing a .mll to .vl converter, etc.

Even if the execution time performance of Coqlex can be increased, it lays strong foundations for verified lexer generation. It is an alternative to non formally proven lexer generators, it allows to make one more step in proving end-to-end correctness of compilers, and it has already found applications in the real world.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. 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.

References

[1] Alasdair Armstrong, Georg Struth, and Tjark Weber. 2013. Kleene algebra. *Archive of Formal Proofs* 324 (2013).

[2] Michela Becchi and Patrick Crowley. 2013. A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Trans. Archit. Code Optim.* 10, 1, Article 4 (April 2013), 26 pages. DOI: http://dx.doi.org/10.1145/2445572.2445576

[3] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.

[4] Adam Chlipala. 2013. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press.

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

[6] 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*. 27–39.

[7] HaskellWiki. 2020. Smart constructors — HaskellWiki. https://wiki.haskell.org/index.php?title=Smart_constructors&oldid=63322. (2020). Accessed: 2020-10-14.

[8] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) parsers. In *ESOP 2012: Programming Languages and Systems, 21st European Symposium on Programming (LNCS)*. Springer, 397–416. DOI: http://dx.doi.org/10.1007/978-3-642-28869-2_20

[9] Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schille. 1996. Regular expressions for language engineering. *Natural Language Engineering* 2, 4 (1996), 305–328.

[10] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443.

[11] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. *SIGPLAN Not.* 49, 1 (Jan. 2014), 179–191. DOI: http://dx.doi.org/10.1145/2578855.2535841

[12] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[13] Takashi Miyamoto. 2011. Coq Regular Expression Git Page. https://github.com/coq-contribs/regexp. (2011). Accessed: 2020-10-14.

[14] Tobias Nipkow. 1998. Verified lexical analysis. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 1–15.

[15] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 263–273.

[16] François Pottier and Yann Régis-Gianas. 2016. The Menhir parser generator. (2016). http://gallium.inria.fr/fpottier/menhir.

[17] Christian Rinderknecht. 2018. A Mini-ML programming language. https://github.com/rinderknecht/Mini-ML/blob/master/Lang0/Lexer.mll. (2018).

[18] Aleksandra Sikora. 2017. Tutorial: parsing JSON with OCaml. (12 2017). https://medium.com/@aleksandrasays/tutorial-parsing-json-with-ocaml-579cc054924f

[19] Joshua B Smith. 2007. Ocamllex and Ocamlyacc. *Practical OCaml* (2007), 193–211.

[20] Laurent Thery. 2020. Coq String Module Documentation. https://coq.inria.fr/library/Coq.Strings.String.html. (2020). Accessed: 2020-10-14.

[21] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

A Appendix

A.1 Source code organization

The Coqlex source code is organized as follows:

The directory regexp_opt: contains the implementation of Coqlex extended version of regular expression based on Brzozowski derivatives.

RValues.v: contains the definition of usual regex (string, character, identifiers, numbers...)

RegexpSimpl.v contains the implementation of regexp simplification described in section 5.4.

MachLen.v: contains the implementation of the score used to perform the longest match rule. Similarly, ShortestLen.v contains the score function of the shortest match rule.

MachLenSimpl.v: contains the optimized version of the longest match rule score computation. Similarly, ShortestLenSimpl.v contains the optimized score function of the shortest match rule.

LexerDefinition.v: contains the Coqlex election system and data-type definition.

CoqlexUtils.v: contains the definition of usual semantic action.

CoqlexLexer.v: contains definition of the lexer of the *Coqlex generator*.

Extraction.v: contains the extraction directives of the .v files above.

Parser.vy: contains definition of the parser of the *Coqlex generator*.

coqlex.ml: contains *Coqlex generator* main function.

ParserUtils.ml and LexerUtils.ml: contains OCaml function that facilitates the use of the OCaml extracted code of Coqlex lexers.

The directory example: contains examples of Coqlex lexer specified by .vl files, their OCamllex equivalent and benchmark data.

The directory Comparison: contains JSON benchmark data, Verbatim++, OCamllex and Coqlex lexers and a python code that allowed to plot the Figure 14.

A.2 OCamllex version of mini-cal

```
(* header section *)
{
open Lexing
open TokenDefinition.
}

(* regexp definitions *)
let ident = ['a'-'z']+
let numb = ['0'-'9']+

(* lexer definitions*)
rule minlexer = parse
| '\n' { new_line lexbuf; minlexer lexbuf }
| ident {ID (Lexing.lexeme lexbuf)}
| numb { Number (Lexing.lexeme lexbuf)}
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { Eof }
| _ { failwith ("unknown token : " ^ (Lexing.lexeme lexbuf))}

(* trailer section *)
{}
```

Figure 18. mini-cal.mll file