Coqlex: Generating formally verified lexers

W. Ouedraogo G. Scherer L. Straßburger

Siemens Mobility - INRIA Saclay - LIX

ouedraogo@lix.polytechnique.fr

Scalp Working Group Days - 16/02/2023

Presentation Overview

Compilation

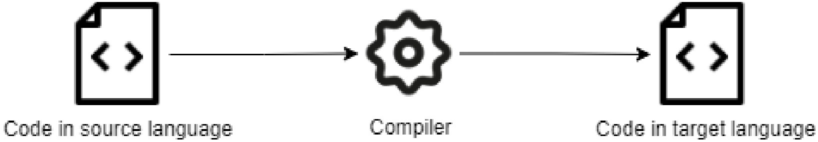


Figure: Compiler design 1/4.

Compilation

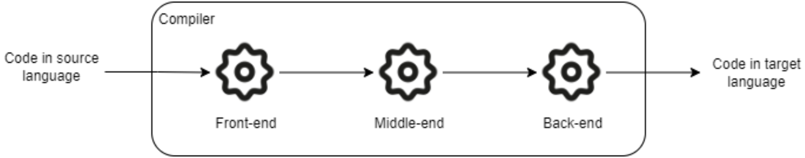


Figure: Compiler design 2/4.

Compilation

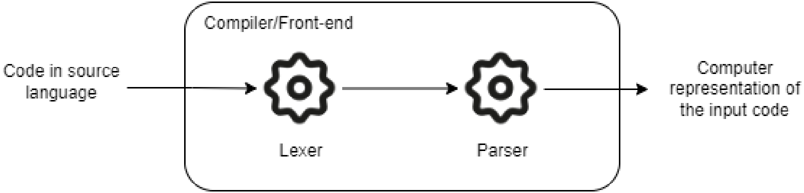


Figure: Compiler design 3/4.

Front-end overview

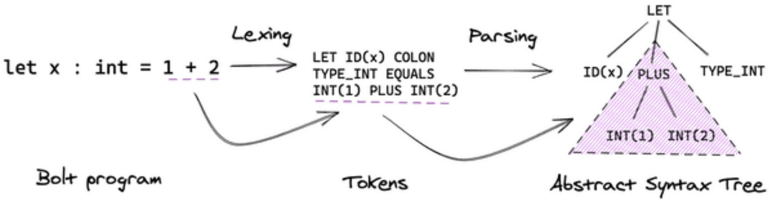


Figure: Compiler design 4/4.

Lexers in practice

From a given string, a lexer generates a stream/list of tokens: part of the input string (lexeme) associated with meaning.

Common additional features

1. Ignore white spaces and comments
2. Detect/reject keywords
3. Track line/column numbers

Lexers and parser are usually generated using generators

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

Lexical buffer data structure:

- tracks positions
- used as lexer input
- is updated by lexers

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
(*OCamllex syntax*)
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'*        { my_lexer lexbuf }
  | 'c' { 20 }
  | 'c'+ { 21 }
```

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'*        { my_lexer lexbuf }
  | 'c' { 20 }
  | 'c'+ { 21 }
```

The priority and longuest match rule: the semantic action of the first lexical rule whose regex matches the longest prefix of the input string.

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'*   { my_lexer lexbuf }
  | 'c'    { 20 }
  | 'c'+   { 21 }
```

Tokens for 'c' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'a'*   { my_lexer lexbuf } (*Matches ``*)
  | 'c'    { 20 } (*Matches `c` *)
  | 'c'+   { 21 } (*Matches `c` *)


  (* result tokens: [20]*)
```

Tokens for 'c' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'*   { my_lexer lexbuf }
  | 'c'    { 20 }
  | 'c'+   { 21 }
```

Tokens for ‘cc‘ ?

Lexer generators

Generating formally verified lexers with Coqlex

W. Ouedraogo, G. Scherer, L. Straßburger

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'a'*   { my_lexer lexbuf } (*Matches ''*)
  | 'c'    { 20 } (*Matches 'c' *)
  | 'c'+   { 21 } (*Matches 'cc' *)


  (* result tokens: [21]*)
```

Tokens for 'cc' ?

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'* { my_lexer lexbuf }
  | 'c'     { 20 }
  | 'c'+   { 21 }
```

Tokens for 'aabbc' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'a'* { my_lexer lexbuf } (*Matches `aa`*)
  | 'c'    { 20 } (*No match*)
  | 'c'+   { 21 } (*No match*)


  (* result tokens: ...*)
```

Tokens for 'aabbc' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*Matches `bb`*)
  | 'a'* { my_lexer lexbuf } (*No match*)
  | 'c'    { 20 } (*No match*)
  | 'c'+   { 21 } (*No match*)


  (* result tokens: [0; ...]*)
```

Tokens for ‘~~aa~~bbc‘ ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'a'* { my_lexer lexbuf } (*No match*)
  | 'c'    { 20 } (*Matches `c` *)
  | 'c'+   { 21 } (*Matches `c` *)



  (* result tokens: [0; 20]*)
```

Tokens for '~~aabbc~~' ?

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 }
  | 'a'*   { my_lexer lexbuf }
  | 'c'    { 20 }
  | 'c'+   { 21 }
```

Tokens for 'd' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'a'*   { my_lexer lexbuf } (*Matches ''*)
  | 'c'    { 20 } (*No match*)
  | 'c'+   { 21 } (*No match*)


  (* result tokens: Infinite loop*)
```

Tokens for 'd' ?

Lexer generators

Important concepts

1. Lexical buffer
2. Lexical rules
3. Selection system

```
rule my_lexer = parse
  'b' 'a'* 'b'   { 0 } (*No match*)
  | 'c'     { 20 } (*No match*)
  | 'c'+   { 21 } (*No match*)


  (* result tokens: Error*)
```

Tokens for 'd' ?

Coqlex: What? Why?

Contribution

1. A Coq library
2. A generator

Goals

1. Simplify lexer implementation
2. Allow to write proofs on implemented lexers
3. Usable
4. Easy to understand/review/improve

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives
3. Selection system
4. Optimization

```
lexer(T) :=
nat ->lexbuf ->Result(T) x lexbuf

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

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives (1/4)
3. Selection systems
4. Optimization

Regex definition

$$
\begin{array}{lll}
\mathit{regex} ::= & & \\
 & \varnothing_r & L(\varnothing_r) = \varnothing \\
 & \mid \epsilon_r & L(\epsilon_r) = \{\epsilon\} \\
 & \mid [\![a]\!] & L([\![a]\!]) = \{a\} \\
 & \mid e_1 + e_2 & L(e_1 + e_2) = L(e_1) \cup L(e_1) \\
 & \mid e_1 \cdot e_2 & L(e_1 \cdot e_2) = \\
 & & \{s_1 +\!\!+\, s_2 | s_1 \in L(e_1) \wedge s_2 \in L(e_2)\} \\
 & \mid e^* & L(e^*) = \{s^n | s \in L(e) \wedge n \in \mathbb{N}\}
\end{array}
$$

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives (2/4)
3. Selection system
4. Optimization

the `nullable` funtion

$$\texttt{nullable}\ \varnothing_r = \texttt{false}$$
$$\texttt{nullable}\ \epsilon_r = \texttt{true}$$
$$\texttt{nullable}\ [\![a]\!] = \texttt{false}$$
$$\texttt{nullable}\ (e_1 + e_2) = \texttt{nullable}\ e_1 \lor \texttt{nullable}\ e_2$$
$$\texttt{nullable}\ (e_1 \cdot e_2) = \texttt{nullable}\ e_1 \land \texttt{nullable}\ e_2$$
$$\texttt{nullable}\ e^* = \texttt{true}$$

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives (3/4)
3. Selection system
4. Optimization

the `derive` funtion

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

Implementation details

1. Typing a lexer
2. Brzozowski derivatives (4/4)
3. Selection system
4. Optimization

Matching a string

$$r//\epsilon = r \quad r//az = (r/a)//z$$

$$\texttt{matches}\ \ r\ \ z = \texttt{nullable}\ \ (r//z)$$

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives
3. Selection system
   - Score: $\mathbb{S}_l$
   - Selection
4. Optimization

Score computation

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

$$\frac{\mathbb{S}_l(r/a, z) = n}{\mathbb{S}_l(r, az) = n + 1}$$

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

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

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives
3. Selection systems
   - Scores
   - Selection
4. Optimization

Semantic rule selection Choosing the first rule with the highest score (first argmax).

Problem: Lexing in quadratic time.

Coqlex Library implementation details

Implementation details

1. Typing a lexer
2. Brzozowski derivatives
3. Selection systems
4. Optimization

Idea: Stop $\mathbb{S}_l$ as soon as possible

Details

1. Adding faster constructions
   ex: ['0'- '9'] vs '0' | '1' | ... | '9'
2. Regexp simplification
   ex: $(r^*)^* \equiv r^*$, $r \cdot \varnothing_r \equiv \varnothing_r$
3. Stopping for trivial cases
   ex: $\mathbb{S}_l(\epsilon_r, s) = 0$, $\mathbb{S}_l(\varnothing_r, s) = -\infty$

Coqlex generator

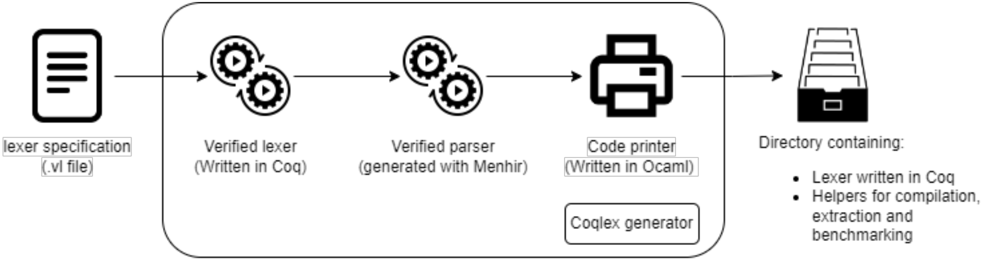


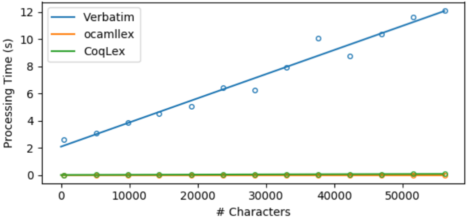
Figure: Coqlex generator architecture.

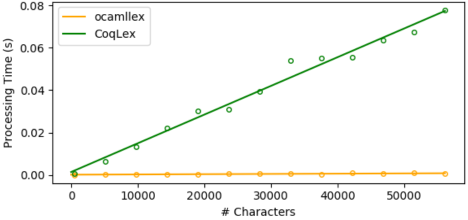
State of the art

Related work and tools

1. Lexers written by hand (ex: CakeML)
2. Nipkow
3. OCamllex (Lex, Flex)
4. Verbatim/Verbatim++

Coqlex vs OCamllex vs Verbatim++





	Verbatim++	Coqlex	OCamllex
Tokens per sec.	1.7×10^3	2.23×10^5	3.9×10^7
Time to process 56ko.	12.11 s	7.7×10^{-2} s	4.4×10^{-4} s

Coqlex vs OCamllex vs Verbatim++

	Coqlex	OCamllex	Verbatim++
Lexer language	Coq	OCaml	Coq
Semantic action	`lexbuf -> Result(token) x lexbuf`	OCaml code	`token`
Error handling mechanism	yes	yes	no
Formally verified lexers	yes	no	yes
Execution speed	100x slower	fastest (reference)	10000x slower
Generator	yes	yes	no

Generator language

.vl syntax vs .mll syntax

Listing 1: looping.vl

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

Listing 2: looping.mll

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

Remark: We proved that this lexer loops if the input string starts by a character x such that $x \neq$ 'a', and $x \neq$ 'b'

Coqlex industrial use-case

Coqlex in industry

1. In a Ada-to-Ada formally verified compiler
2. Biggest program: 2380 files (25MB of code)
3. Formally verified front-end
4. Compilation timeè: x4 compared to the unverified front-end version.

Coqlex in a nutshell

1. Usable
2. Simple
3. Formally verified
4. Common lexer features are implemented

- Coq proof of $\mathbb{S}_l$, correctness and completeness
- Coq proof of Lexical rule selection is correctness and completeness
- Coq proof of Optimizations correctness

Future work

Improvements for Coqlex

1. Speed up
2. Termination proof
3. OCamllex <−> Coqlex converter
4. CompCert

The End

References

- Brzozowski Janusz A. (1964)
  Derivatives of regular expressions
  JACM 1964
- Joshua B Smith (2007)
  Ocamllex and Ocamlyacc
- Derek Egolf and al. (2022)
  Verbatim++: verified, optimized, and semantically rich lexing with derivatives
  CPP 2022 27 – 39.
- W. OUEDRAOGO and al. (2022)
  Git link for Coqlex: `https://gitlab.inria.fr/wouedrao/coqlex`

Thank you.
Questions?
Comments?