RE2C — a lexer generator based on lookahead TDFA

by Ulya Trofimovich, April 2021
Motivation for this talk

➔ Recent development of parsing theory: TDFA, deterministic finite-state machines capable of regular expression parsing, not only recognition.

➔ RE2C: a tool for generating fast lexical analyzers.
Agenda

➔ Background: languages & automata
➔ Lexer generators
➔ Submatch extraction & lookahead TDFA
➔ Benchmarks
Background: languages & automata

- Formal grammars → Chomsky hierarchy → Regular expressions → Extensions
  → Recognition & parsing → Ambiguity → Finite-state automata → NFA
  → Simulation → Determinization → DFA

- Lexer generators
- Submatch extraction & lookahead TDFA
- Benchmarks
Formal grammars

Formal grammars are a way to give a finite definition for a possibly infinite set of strings (a language). Each string in a language is derived from the start symbol by applying a sequence of production rules.

A **formal grammar** is a tuple \(\langle \Sigma, N, P, S \rangle\) where:

- \(\Sigma\) is the alphabet of terminal symbols
- \(N\) is the alphabet of non-terminal symbols
- \(P\) is the set of production rules
- \(S\) is the start symbol

### Example: additive expressions

\[
\begin{align*}
\text{Exp} & \rightarrow \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp} \mid \text{Num} \\
\text{Num} & \rightarrow \text{Dgt} \mid \text{Dgt} \text{Num} \\
\text{Dgt} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

**Derivation for “1+2”:**

\[
\begin{align*}
\text{Exp} & \rightarrow \text{Exp} + \text{Exp} \\
& \rightarrow \text{Dgt} + \text{Exp} \\
& \rightarrow 1 + \text{Exp} \\
& \rightarrow 1 + \text{Num} \\
& \rightarrow 1 + 2
\end{align*}
\]
Noam Chomsky, 1959: a hierarchy of formal grammars:

<table>
<thead>
<tr>
<th>Type</th>
<th>Languages</th>
<th>Production rules</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0</td>
<td>Recursively enumerable</td>
<td>$\alpha \rightarrow \gamma$</td>
<td>Turing machine</td>
</tr>
<tr>
<td>Type 1</td>
<td>Context-sensitive</td>
<td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td>
<td>Linear bounded Turing machine</td>
</tr>
<tr>
<td>Type 2</td>
<td>Context-free</td>
<td>$A \rightarrow \gamma$</td>
<td>Pushdown automaton</td>
</tr>
<tr>
<td>Type 3</td>
<td>Regular</td>
<td>$A \rightarrow \varepsilon</td>
<td>a</td>
</tr>
</tbody>
</table>

Chomsky, N. (1959) *On certain formal properties of grammars.* [https://doi.org/10.1016/S0019-9958(59)90362-6](https://doi.org/10.1016/S0019-9958(59)90362-6)
Regular expressions is another way of describing regular languages, equivalent to Type 3 grammars. They were invented by Stephen Cole Kleene in 1951. A rigorous definition via Kleene algebra was given by Dexter Kozen, 1981.

A widely used recursive definition:

1. \( \varepsilon \) (empty word) and \( a \) in \( \Sigma \) (alphabet symbol) are regular expressions.
2. If \( e_1, e_2 \) are regular expressions, then \( e_1 e_2 \) (concatenation), \( e_1 | e_2 \) (alternative) and \( e_1^* \) (repetition) are regular expressions.

RE can express concatenation, alternative, repetition, but not nested constructs.

Kleene. (1951) Representation of Events in Nerve Nets and Finite Automata
https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf

<table>
<thead>
<tr>
<th>Extension</th>
<th>Syntax</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character classes/sets</td>
<td>[a-zA-Z], [[[:lower:]]] ...</td>
<td>Regular</td>
</tr>
<tr>
<td>Escape sequences</td>
<td>\t, \n ...</td>
<td>Regular</td>
</tr>
<tr>
<td>Generalized repetition</td>
<td>e?, e+, e{n}, e{n,m}</td>
<td>Regular</td>
</tr>
<tr>
<td>Non-greedy repetition</td>
<td>e??, e*?, e+?</td>
<td>Regular</td>
</tr>
<tr>
<td>Unanchored matches</td>
<td>Search anywhere in the string</td>
<td>Regular</td>
</tr>
<tr>
<td>Assertions</td>
<td>^, $, /e, ?!e ...</td>
<td>Regular (?)</td>
</tr>
<tr>
<td>Negation, intersection</td>
<td>¬e, e₁ &amp; e₂</td>
<td>Regular</td>
</tr>
<tr>
<td>Submatch extraction</td>
<td>Capturing groups: (e)</td>
<td>Regular</td>
</tr>
<tr>
<td>Backreferences</td>
<td>(e)\n ...</td>
<td>Non-regular (CS?)</td>
</tr>
</tbody>
</table>
Recognition & parsing

→ **Recognition**: determine if a string belongs to the language defined by the grammar (yes/no answer).

→ **Parsing**: find a derivation of a string in the grammar (construct a parse graph, more widely known as a parse tree).

---

**Ambiguity** is the existence of more than one parse graph for the same string.

Ambiguity is a property of grammar — a language can have many grammars, some of them ambiguous and some unambiguous.

**Example of an ambiguous grammar:**

\[
\begin{align*}
Exp & \rightarrow Exp + Exp \mid Exp - Exp \mid Num \\
Num & \rightarrow Dgt \mid Dgt \ Num \\
Dgt & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

*Multiple parse trees for “1-2+3”.*
Finite-state automata

**NFA** is a tuple \((Q, \Sigma, \Delta, q_0, F)\), where:
- \(Q\) is a finite set of states
- \(\Sigma\) is a finite set of input symbols (alphabet)
- \(\Delta \subseteq Q \times (\Sigma \cup \varepsilon)\) is a transition relation
- \(q_0\) is the initial state
- \(F\) is a set of final states

**DFA** is a tuple \((Q, \Sigma, \delta, q_0, F)\), where:
- \(Q\) is a finite set of states
- \(\Sigma\) is a finite set of input symbols (alphabet)
- \(\delta : Q \times \Sigma \rightarrow Q\) is a transition function
- \(q_0\) is the initial state
- \(F\) is a set of final states

Example: NFA and DFA that recognize the regular language defined by \(RE\ a^*b^*|ab\)
There are many different NFA constructions:

- Thompson
- Glushkov (a.k.a. position NFA)
- ...

No single best construction. Key properties:

- $\epsilon$-transitions?
- Ambiguity-preserving?
- How many states?
- Maximum in/out-degree of a state?

We will use Thompson construction.
It is ambiguity-preserving and linear in RE size.

NFA simulation for string “ab”: build ε-closure, step on symbol, repeat. Keep a set of active states at each step.
Determinization of Thompsons NFA for RE $a^*b^*|ab$.
Simulate NFA on all possible strings, merging identical state-sets at each step.

Arrows are $\varepsilon$-closure paths.
There is a unique minimal DFA. Any other DFA can be converted to it.

**DFA execution** is very simple: starting from the initial state, follow a unique transition labeled by the next input symbol.

Time complexity is $\Theta(n)$, where $n$ is the length of the input string. The algorithm works in constant memory independent of $n$.

Determinization may take exponential time (due to the worst-case exponential DFA size).

*Non-minimal and minimal DFA for RE $a^*b^*|ab$*
The following formalisms are equivalent and describe regular languages:

- Type 3 grammars
- Regular expressions
- NFA
- DFA

Basic NFA simulation / DFA execution algorithms do recognition, not parsing.

DFA execution is very fast, provided that determinization is done ahead of time.

This theory is well-known and goes back to 1950s.
Lexer generators

- Background: languages & automata
- Lexer generators
  AOT-compilers for RE → RE2C → An old “unfixable” bug → Generalized problem
- Submatch extraction & lookahead TDFA
- Benchmarks
AOT-compilers for RE

Lexer generators:

- **Extend syntax** of programming languages
- Allow one to **map RE to semantic actions** that are executed on match
- **Compile** to code in the target language
- Usually implemented as preprocessors, compile **ahead of time**
- Use **deterministic** automata (determinization is not included in run time)
- Suitable for **static RE** (known in advance), not dynamic RE

Key features of a lexer generator:

- **DFA encoding** (table-based, direct-code)
- Handling the **end-of-input** situation (bounds checking, sentinel symbol, hybrid, user-defined)
- **Input model** (fixed, flexible, user-defined)
- Support for **RE extensions**
RE2C (re2c.org and https://github.com/skvadrik/re2c) is a lexer generator with the main goal of generating **fast** code. Second-main goal is **flexibility** of the user interface.

- Peter Bumbulis, 1993 (name means “regular expressions to C”)
- C/C++ and Go backends (want Rust!)
- Flexible interface (no fixed program template — users write their own interface code)
- Different input models, from simple *-terminated strings to very large buffered input
- Different end-of-input handling methods
- Allows multiple interrelated lexer blocks
- Encodings: ASCII, Unicode (UTF8/16/32, UCS2), EBCDIC
- Header files / include files
- Self-validation for optimizations (generates path cover for unoptimized DFA)


https://re2c.org/1994_bumbulis_cowan_re2c_a_more_versatile.Scanner-generator.ps
RE2C (example for C)

```c
#include <assert.h>                 // C/C++ code
int lex(const char *YYCURSOR) {     //
    char yych;
    yych = *YYCURSOR;         //
    switch (yych) {          //
        case 'a' ... 'z': goto yy4;    //
        default: goto yy2;        //
    }
    yy2:                         //
        ++YYCURSOR;            //
        { return 1; }         //
    yy4:                         //
        yych = *++YYCURSOR;    //
        switch (yych) {       //
            case '0' ... '9': goto yy6; //
            case 'a' ... 'z': goto yy4; //
            default: goto yy6;     //
        }
        yy6:                        //
            { return 0; }        //
    }
}
int main() {                        // C/C++ code
    assert(lex("zer0") == 0);       //
    return 0;                       //
}
```

```c
#include <assert.h>               // C/C++ code
int lex(const char *YYCURSOR) {   //
    /*!re2c                       // start of block
        re2c:define:YYCTYPE = char;   // config
        re2c:yyfill:enable = 0;       // config
        re2c:flags:case-ranges = 1;   // config
        ident = [a-z][a-z0-9]*;       // named def
        ident { return 0; }           // normal rule
        *     { return 1; }           // default rule
        */
    }

    int main() {                      // C/C++ code
        assert(lex("zer0") == 0);     //
        return 0;                     //
    }
}
```
An old “unfixable” bug

A bug in the trailing context (a.k.a. “lookahead operator”) that won’t get fixed: if regular expressions $R$ and $S$ match overlapping languages, the generated lexer may produce incorrect results:

$$R / S$$

Flex calls this ‘dangerous trailing context’ and generates warnings. For example:

$$zx*/xy*$$

*Flex manual → Limitations.* [https://westes.github.io/flex/manual/Limitations.html#Limitations](https://westes.github.io/flex/manual/Limitations.html#Limitations)
Consider a simple RE $a^*b^*|ab$ with submatch marker between $a^*$ and $b^*$ (in RE2C syntax):

$[a]^*@t[b]^*|[a][b]$

A C/C++ programmer can write something like this:

```c
while (*s++ == 'a');
t = s;
while (*s++ == 'b');
```

Can RE2C generate code as efficient and simple as the above?
Submatch extraction & lookahead TDFA

→ Background: languages & automata
→ Lexer generators

→ Submatch extraction & lookahead TDFA
  Submatch extraction → TNFA → How to fold? → Laurikari determinization → TDFA → Eliminating redundancy → Lookahead determinization → Lookahead TDFA → Real-world code → Optimizations → Disambiguation → Full parsing

→ Benchmarks
What do we expect of submatch extraction on DFA?

➔ Worst case is as hard as parsing
➔ Best case should be as efficient as a bare DFA
➔ Overhead should be proportional to submatch detailization
➔ Lexer generators need to generate efficient code
➔ Have to deal with ambiguity in regular expressions
Ville Laurikari, 2000: **TNFA** — NFA with tagged transitions. **Tags** are submatch markers that can be placed anywhere in RE, e.g. $a^* \cdot t \cdot b^* | ab$.

Simulation needs to track tag values.

**TNFA simulation on string “ab”**

**TNFA for RE $a^* \cdot t \cdot b^* | ab$**
How to fold?

Problem:

**How to fold DFA?** Seems impossible to merge states, because state-sets extended with tag information are no longer identical (tag values are different at each step).

Solution (Ville Laurikari, 2000):

Use references to tag locations rather than immediate values! Add operations on DFA transitions that will update tag values at locations. When mapping states with different locations, add copy operations to reorder tag values at locations.

Separate static and dynamic part in the state-sets.

*Laurikari. (2000) NFAs with Tagged Transitions, their Conversion to Deterministic Automata and Application to Regular Expressions.*

https://laurikari.net/ville/spire2000-tnfa.pdf
Laurikari determinization

Determinization of TNFA for RE $a^* @ t b^* | ab$. Simulate TNFA on all possible strings, mapping state-sets with identical TNFA states at each step. Add operations on incoming transitions.

Arrows are tagged $\varepsilon$-closure paths.
TDFA is like ordinary DFA extended with a fixed number of registers and register operations on transitions.

But this is not what we want! We want:

```c
while (*s++ == 'a');
t = s;
while (*s++ == 'b');
```

And the optimized TDFA is equivalent to:

```c
while (*s++ == 'a') t = s;
while (*s++ == 'b');
```
Eliminating redundancy

Problem:

How to eliminate redundant register operations?

Solution:

Use the **lookahead symbol** to filter them out!

Delay register operations one step. Store **lookahead tags** in TDFA states under construction and take them into account when mapping TDFA states.

This reminds of the difference between LR(1)/LALR(1) and LR(0), therefore Laurikari construction is called **TDFA(0)**, and the lookahead construction is called **TDFA(1)**.

Lookahead-aware determinization for RE $a^* @ t b^* | ab$.
State-sets are extended with lookahead tags from the incoming transitions, operations are delayed to outgoing transitions.
Lookahead TDFA

Lookahead TDFA has fewer register operations, and it has the effect of lifting operations out of loops.

Optimized lookahead TDFA for regular expression $a^* @ t b^* | ab$ is equivalent to:

```c
while (*s++ == 'a');
t = s;
while (*s++ == 'b');
```
This is the real code that RE2C generates for regular expression \(a^* @ t b^*|ab\) (for the C/C++ backend, modulo whitespace).

There is one tag variable `yyt1` and exactly one tag variable assignment on any code path.

```c
YYCTYPE yych;
goto yy0;

yy1:
    ++YYCURSOR;

yy0:
    yych = *YYCURSOR;
    switch (yych) {
        case 'a': goto yy1;
        case 'b': yyt1 = YYCURSOR; goto yy4;
        default:  yyt1 = YYCURSOR; goto yy3;
    }

yy3:
    t = yyt1; {/* use t ... */}

yy4:
    yych = *++YYCURSOR;
    switch (yych) {
        case 'b': goto yy4;
        default:  goto yy3;
    }
```
Optimizations

In lexer generators registers are mapped to variables ⇒ TDFA induces a CFG ⇒ the usual compiler optimizations are applicable.

- Liveness analysis on registers (iterative data-flow, or on SSA)
- Dead code elimination
- Variable allocation (analogue of the usual register allocation)
- Copy coalescing (particularly helpful, removes copy operations)
- Lifting common operations out of branches
- ...

Minimization.

- Canonical algorithms (e.g. Moore’s), adapted to distinguish transitions with operations
- Must go after CFG optimizations to reduce transition interference
Disambiguation

Not to be confused:

- **Non-determinism**: multiple versions of a tag in the same TDFA state.
- **Ambiguity**: multiple versions of a tag in the same TNFA state reached by different paths.

Registers take care of non-determinism, **disambiguation policy** takes care of ambiguity.

- **POSIX (longest-match)**: difficult to implement (libraries like RE2 gave up).
- **Perl (leftmost-greedy)**: very easy to implement (just use leftmost DFS in $\varepsilon$-closure).

Disambiguation is applied during determinization. No matter which policy, the resulting TDFA has no overhead (disambiguation decisions are embedded in its structure).

RE2C supports both Perl (@-tags syntax) and POSIX policies (capturing parentheses).

*Borsotti, Trofimovich. (2019) Efficient POSIX Submatch Extraction on NFA.*
https://re2c.org/2019_borsotti_trofimovich_efficient_posix_submatch_extraction_on_nfa.pdf
Full parsing can be done on TDFA by adding tags (or captures) around each symbol, but it is not elegant and DSSTs are better suited to this (Deterministic Streaming String Transducers).

In practice a more useful feature is the ability to extract submatch on all repetitions, not just the last one (as specified by POSIX regcomp/regexec interface).

Don’t use vectors to represent tag values, they make copy operations very expensive. Instead encode tag values in a trie — a tree stored as an array of pairs (tag value, parent index). This way tag variables remain scalar, operations are cheap, and common prefixes of tag histories are deduplicated.

RE2C supports s-tags (single-value tags) and m-tags (multiple-value tags).

Recap

➔ The problem of submatch extraction on DFA has been solved (Laurikari, 2000).
➔ TDFA is an ordinary DFA extended with registers and register operations.
➔ Lookahead TDFA is a practical improvement that allows to greatly reduce the number of registers and register operations.
➔ TDFA is parameterized over disambiguation policy (e.g. POSIX, Perl) and has no runtime overhead on disambiguation.
➔ TDFA supports full parsing or repeated submatch extraction.
➔ TDFA can be minimized.
➔ TDFA in lexer generators benefits from compiler optimizations.
Benchmarks

➔ Background: languages & automata
➔ Lexer generators
➔ Submatch extraction & lookahead TDFA
➔ Benchmarks
A few different groups of benchmarks:

- **AOT-compiled RE (different lexer generators / automata types)**
  
  [https://re2c.org/benchmarks/benchmarks.html#submatch-lexer-generators](https://re2c.org/benchmarks/benchmarks.html#submatch-lexer-generators)

- **JIT-compiled RE (registerless-TDFA vs. TDFA)**
  
  [https://re2c.org/benchmarks/benchmarks.html#submatch-libraries-dfa](https://re2c.org/benchmarks/benchmarks.html#submatch-libraries-dfa)

- **TDFA(0) vs. TDFA(1)**
  
  [https://re2c.org/2017_trofimovich_tagged_deterministic_finite_automata_with_lookahead.pdf](https://re2c.org/2017_trofimovich_tagged_deterministic_finite_automata_with_lookahead.pdf)

Benchmarks show that for submatch extraction:

- TDFA(1) are faster and smaller than TDFA(0)
- TDFA are faster and smaller than other parsing deterministic automata (sta-DFA or DSST)
- Submatch overhead is small (performance is close to bare DFA)
The END.

Thank you!