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

- 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 <*Σ, N, P, S*> where:

- Σ 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

$$
\begin{aligned}
&Exp \longrightarrow Exp + Exp \mid Exp - Exp \mid Num\\
&Num \longrightarrow Dgt \mid Dgt\ Num\\
&Dgt \longrightarrow 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

Derivation for “1+2”:

$$
\begin{aligned}
&Exp \longrightarrow Exp + Exp \longrightarrow Num + Exp\\
&\quad \longrightarrow Dgt + Exp \longrightarrow 1 + Exp\\
&\quad \longrightarrow 1 + Num \longrightarrow 1 + Dgt \longrightarrow 1 + 2
\end{aligned}
$$

Chomsky hierarchy

Noam Chomsky, 1959: a hierarchy of formal grammars:

Type	*Languages*	*Production rules*	*Automaton*
Type 0	Recursively enumerable	$\alpha \longrightarrow \gamma$	Turing machine
Type 1	Context-sensitive	$\alpha A \beta \longrightarrow \alpha \gamma \beta$	Linear bounded Turing machine
Type 2	Context-free	$A \longrightarrow \gamma$	Pushdown automaton
Type 3	Regular	$A \longrightarrow \varepsilon \mid a \mid aB$	Finite-state automaton

Chomsky, N. (1959) On certain formal properties of grammars. https://doi.org/10.1016/S0019-9958(59)90362-6

Regular expressions

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. ε (empty word) and a in Σ (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

Kozen. (1981) A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events
https://www.cs.cornell.edu/~kozen/Papers/ka.pdf

Extensions

Extension	*Syntax*	*Languages*
Character classes/sets	*[a-zA-Z], [[:lower:]]* ...	Regular
Escape sequences	*\t, \n* ...	Regular
Generalized repetition	*e?, e+, e{n,}, e{n,m}*	Regular
Non-greedy repetition	*e??, e*?, e+?*	Regular
Unanchored matches	Search anywhere in the string	Regular
Assertions	*^, $, /e, ?!e* ...	Regular (?)
Negation, intersection	$\neg e$, e_1 & e_2	Regular
Submatch extraction	Capturing groups: *(e)*	Regular
Backreferences	*(e)\n* ...	Non-regular (CS?)

Cox. (2007) Regular Expression Matching Can Be Simple And Fast https://swtch.com/~rsc/regexp/regexp1.html

Recognition & parsing

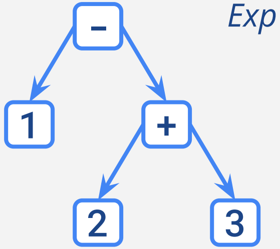
- **Recognition**: determine if a string belongs to the language defined by the grammar (yes/no answer).
- **Parsing**: find a derivation of a string in the grammar (construct a **parse graph**, more widely known as a **parse tree**).

Grune, Jacobs. (1990) Parsing Techniques - a Practical Guide. https://dickgrune.com/Books/PTAPG_2nd_Edition

Ambiguity

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

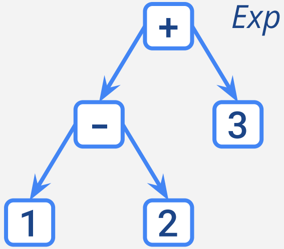
Ambiguity is a property of grammar — a language can have many grammars, some of them ambiguous and some unambiguous.



Exp ⟶ Exp - Exp ⟶ Num - Exp
⟶ Dgt - Exp ⟶ 1 - Exp
⟶ 1 - Exp + Exp ⟶ 1 - Num + Exp
⟶ 1 - Dgt + Exp ⟶ 1 - 2 + Exp
⟶ 1 - 2 + Num ⟶ 1 - 2 + Dgt
⟶ 1 - 2 + 3

Example of an ambiguous grammar:
Exp ⟶ Exp + Exp | Exp - Exp | Num
Num ⟶ Dgt | Dgt Num
Dgt ⟶ 0|1|2|3|4|5|6|7|8|9

Multiple parse trees for "1-2+3".



Exp ⟶ Exp + Exp ⟶ Exp - Exp + Exp
⟶ Num - Exp + Exp ⟶ Dgt - Exp + Exp
⟶ 1 - Exp + Exp ⟶ 1 - Num + Exp
⟶ 1 - Dgt + Exp ⟶ 1 - 2 + Exp
⟶ 1 - 2 + Num ⟶ 1 - 2 + Dgt
⟶ 1 - 2 + 3

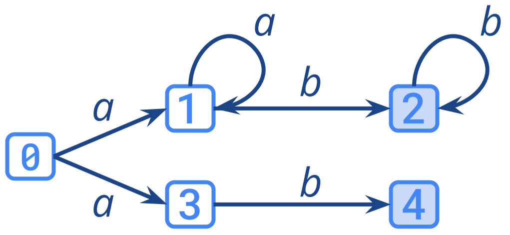
Finite-state automata

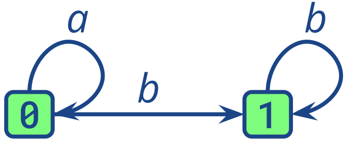
NFA is a tuple $(Q, \Sigma, \Delta, q_0, F)$, where:

- Q is a finite set of states
- Σ 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
- Σ 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*

NFA

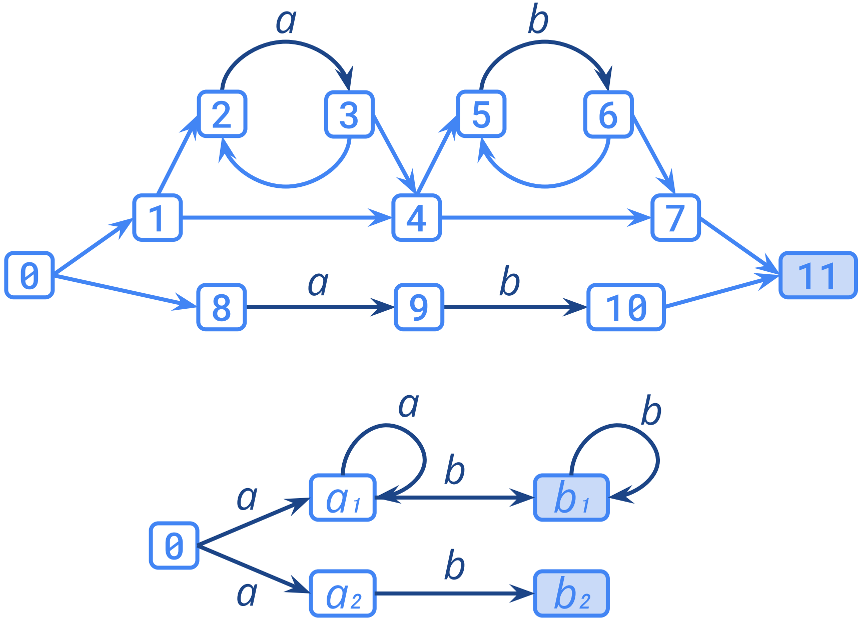
There are many different NFA constructions:

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

No single best construction. Key properties:

- ε-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.

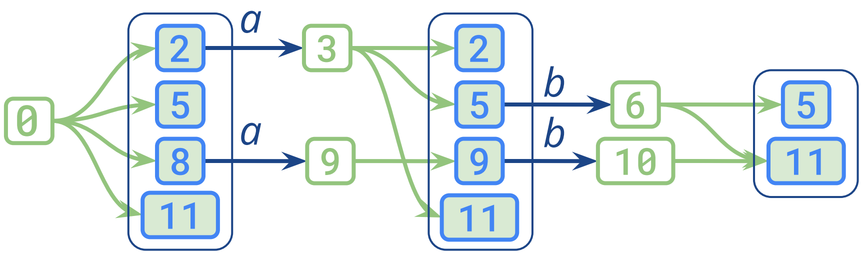


*Different NFA constructions for RE a*b*|ab*

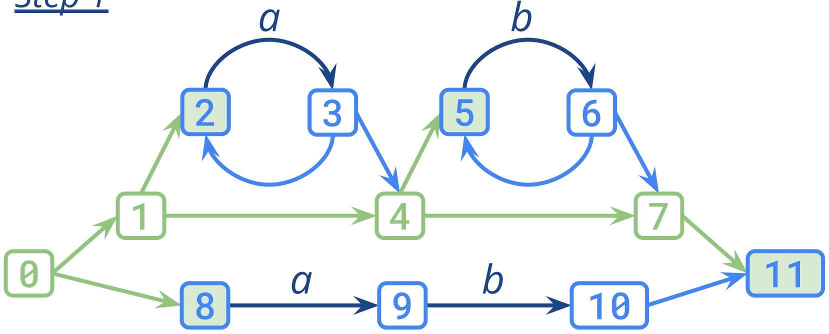
Allauzen, Mohri. (2006) A Unified Construction of the Glushkov, Follow, and Antimirov Automata. https://cs.nyu.edu/~mohri/pub/glush.pdf

Simulation

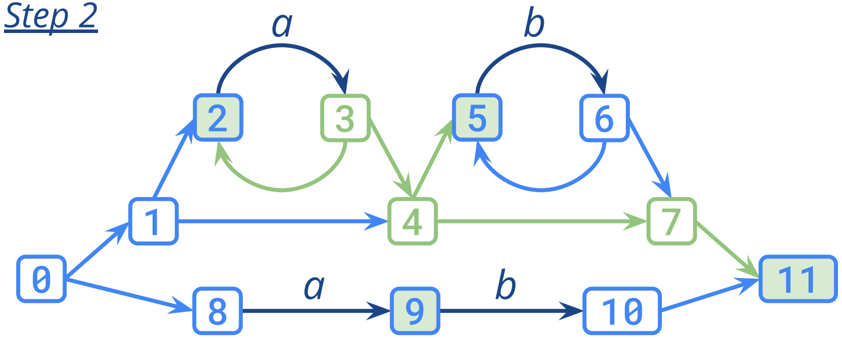
NFA simulation for string ***"ab"****: build ε-closure, step on symbol, repeat. Keep a set of active states at each step.*



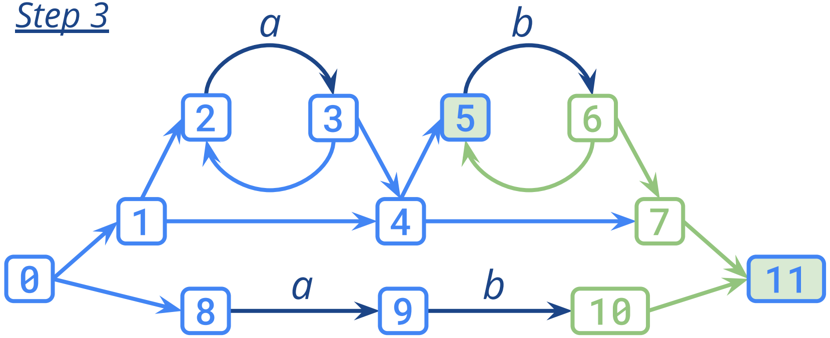
Step 1



Step 2



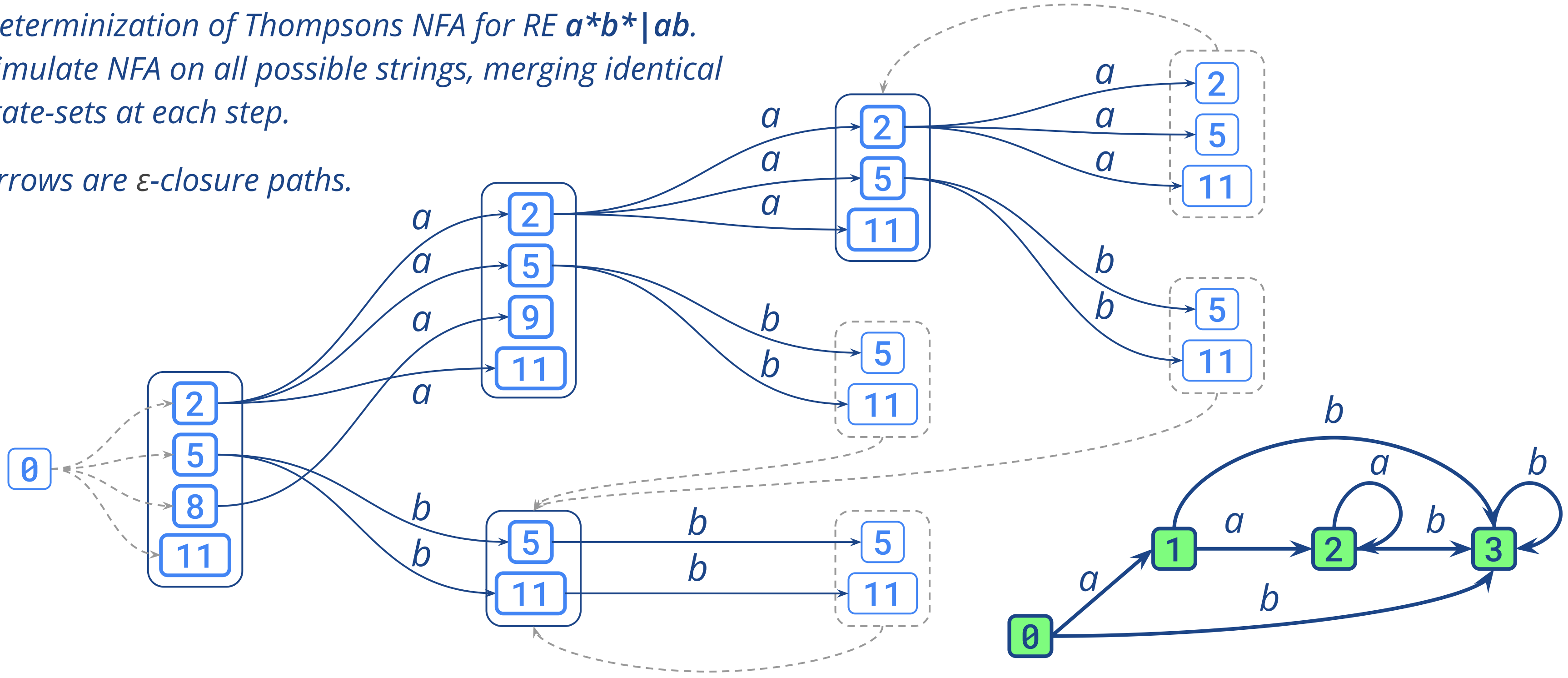
Step 3



Determinization

Determinization of Thompsons NFA for RE ***a*b*|ab****.*
Simulate NFA on all possible strings, merging identical state-sets at each step.

Arrows are ε-closure paths.



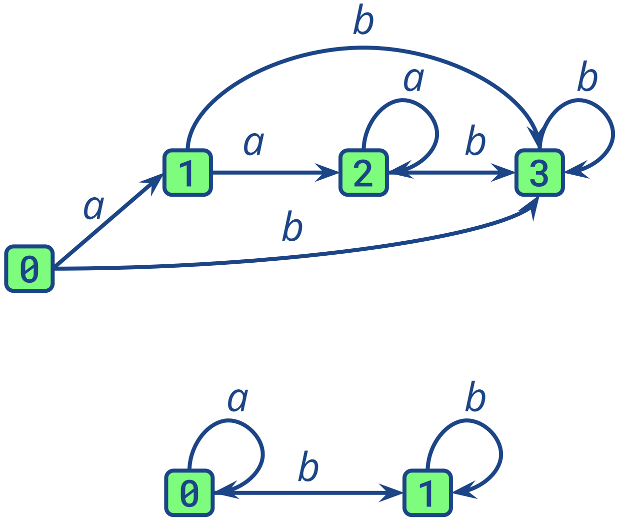
DFA

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 *Θ(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*

Recap

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 (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)

Bumbulis, Cowan. (1993) RE2C --- a More Versatile Lexer Generator.
https://re2c.org/1994_bumbulis_cowan_re2c_a_more_versatile_scanner_generator.ps

RE2C (example for 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;                     //
}
```

```
#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':
        case 'a' ... 'z': goto yy4;
        default: goto yy6;
    }
yy6:
    { return 0; }
}                                   //
                                    //
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

Generalized problem

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:

```
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

Submatch extraction

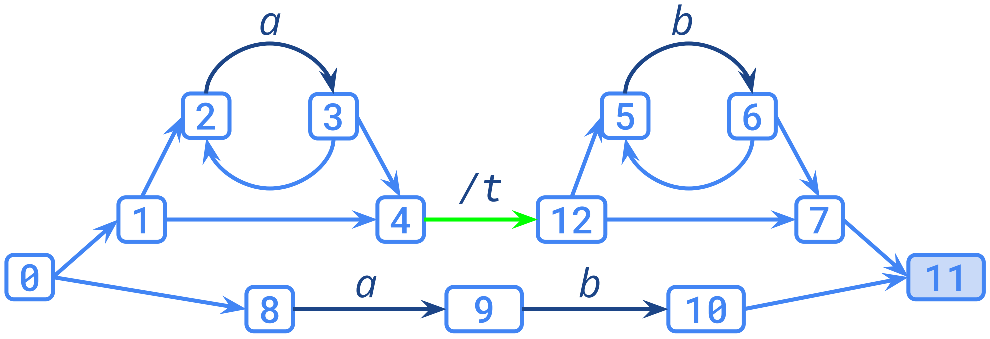
What do we expect of submatch extraction on DFA?

- ➔ Worst case is as hard as parsing
- ➔ Best case should be as efficient as a bare DFA
- ➔ Overhead should be proportional to submatch detalization
- ➔ Lexer generators need to generate efficient code
- ➔ Have to deal with ambiguity in regular expressions

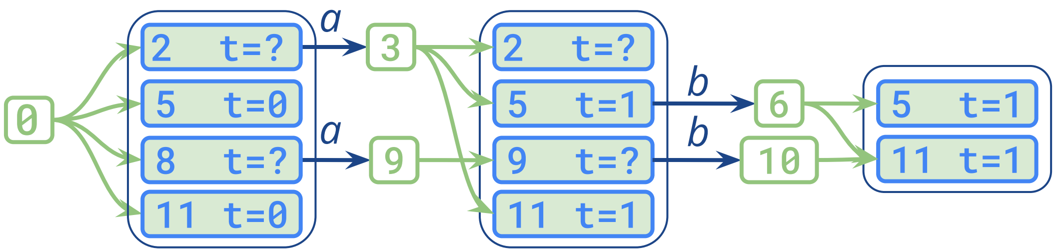
TNFA

Ville Laurikari, 2000: **TNFA** — NFA with tagged transitions. **Tags** are submatch markers that can be placed anywhere in RE, e.g. _a* @t b*|ab_.

Simulation needs to track tag values.



*TNFA for RE **a* @t b*|ab***



*TNFA simulation on string **"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):

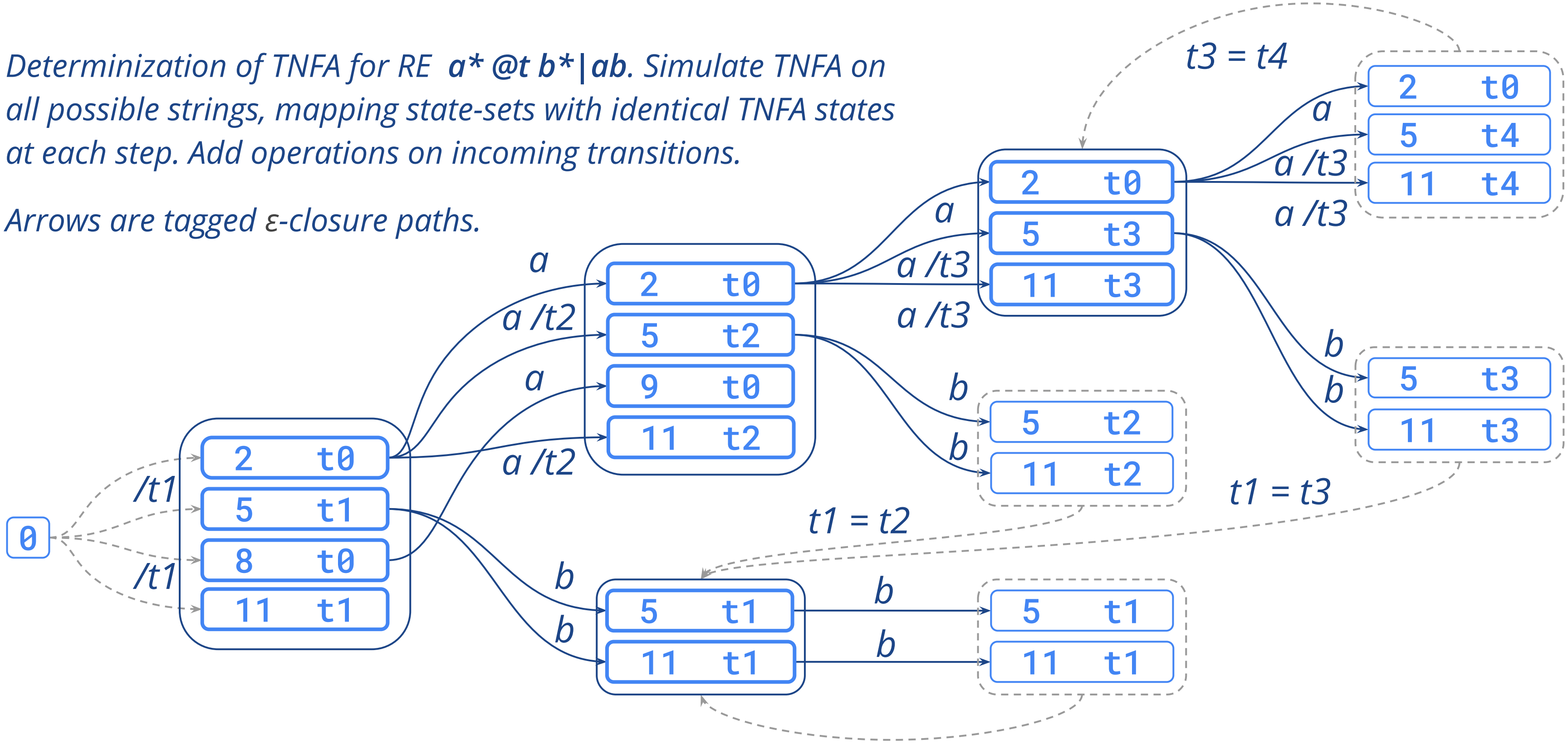
> Use **references to tag locations** rather than immediate values! Add **operations on DFA transitions** that will update tag values at locations. When mapping states with different locations, add **copy operations to reorder tag values** at locations.
>
> Separate **static** and **dynamic** part in 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

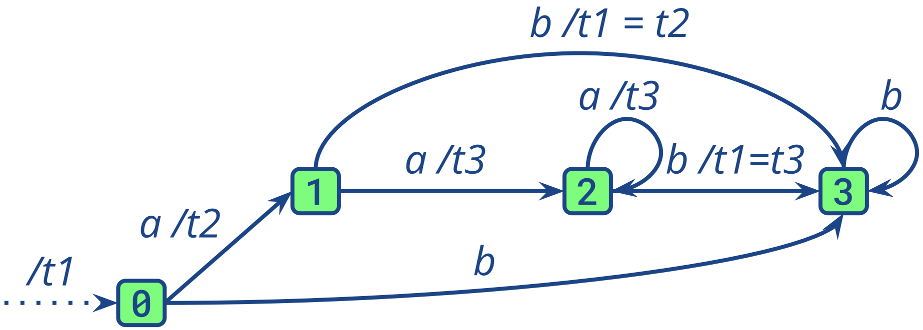
Laurikari determinization

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.*

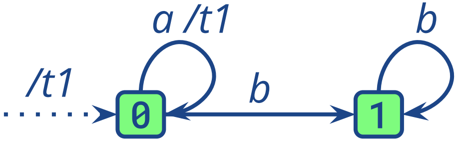
Arrows are tagged ε-closure paths.



TDFA



TDFA for RE ***a* @t b*|ab***



optimized TDFA for RE ***a* @t b*|ab***

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:

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

And the optimized TDFA is equivalent to:

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

Eliminating redundancy

Problem:

> **How to eliminate redundant register operations?**

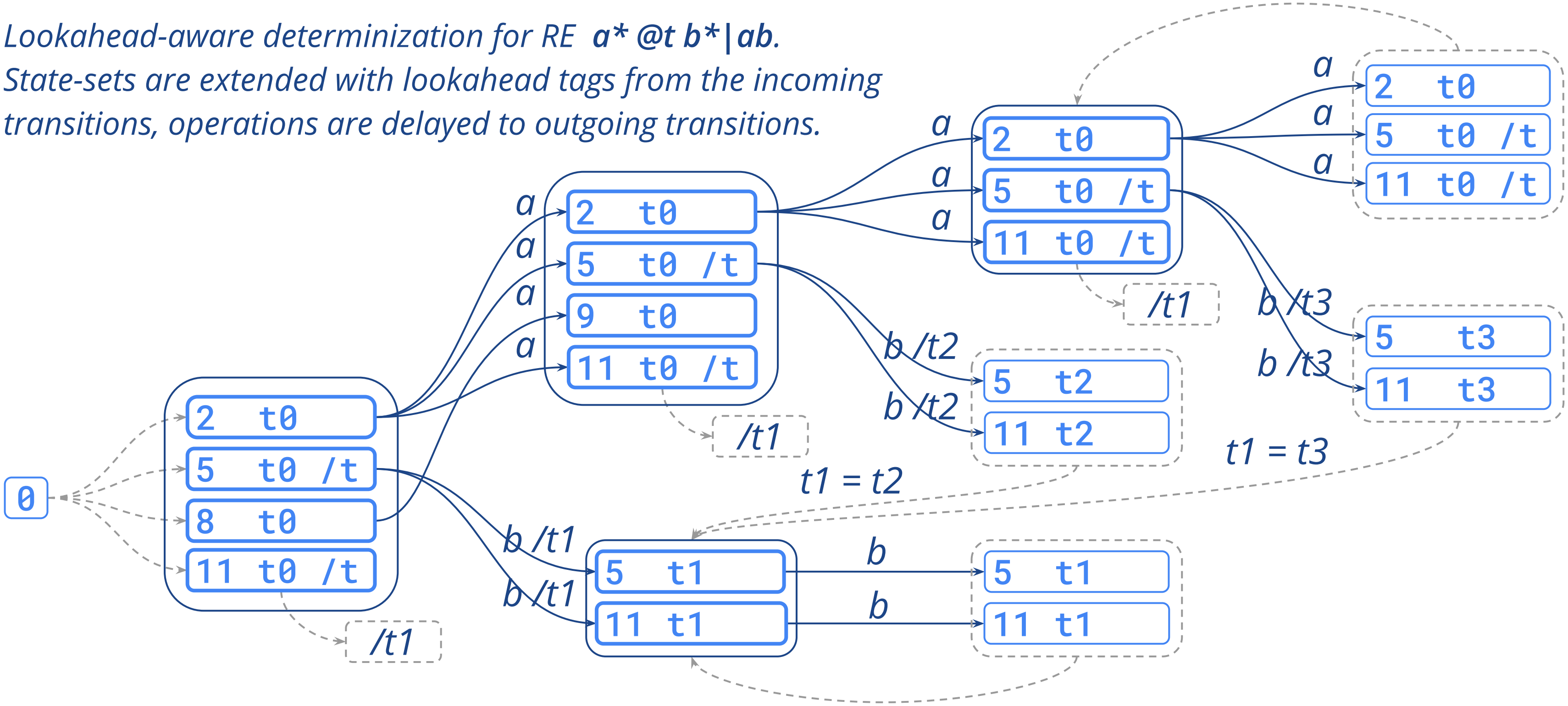
Solution:

> 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)**.

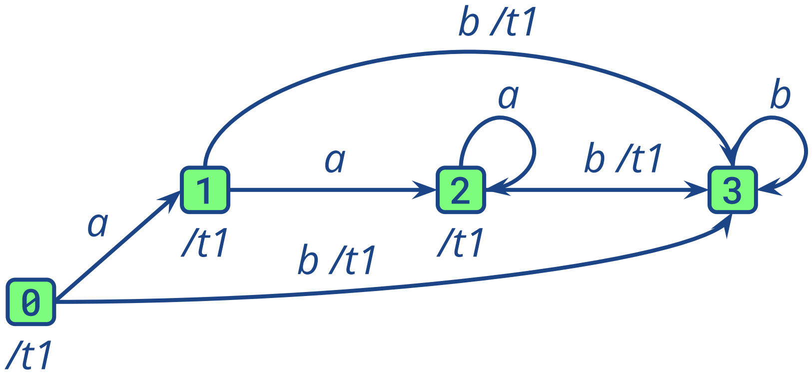
Trofimovich. (2017) Tagged Deterministic Finite Automata with Lookahead. https://arxiv.org/pdf/1907.08837.pdf

Lookahead determinization

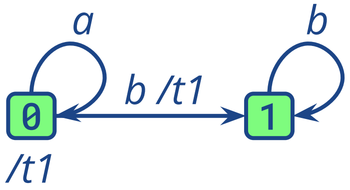
Lookahead-aware determinization for RE ***a* @t b*|ab****.*
State-sets are extended with lookahead tags from the incoming transitions, operations are delayed to outgoing transitions.



Lookahead TDFA



*Lookahead TDFA for RE **a* @t b*|ab***



*Optimized lookahead TDFA for RE **a* @t b*|ab***

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:

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

Real-world code

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.

```
$ re2c -i -tags -
/*!re2c
      re2c:yyfill:enable = 0;
      [a]* @t [b]* | [a][b] {/* use t ... */}
*/
```

```
    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 ε-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

Full parsing can be done on TDFA by adding tags (or captures) around each symbol, but it is not elegant and **DSST**s 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).

Grathwohl. (2015) Parsing with Regular Expressions & Extensions to Kleene Algebra.
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.699.9957&rep=rep1&type=pdf

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**

Benchmarks

A few different groups of benchmarks:

- ➔ AOT-compiled RE (different lexer generators / automata types)
  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
- ➔ TDFA(0) vs. TDFA(1)
  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!