avast

COMPILE TIME REGULAR EXPRESSIONS
WITH
A DETERMINISTIC FINITE AUTOMATON

HANA DUSÍKOVÁ

HANA DUSÍKOVÁ

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code
- Avast Prague C++ Meetup

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code
- Avast Prague C++ Meetup
- Czech National Body in WG21

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code
- Avast Prague C++ Meetup
- Czech National Body in WG21

- Occasional hiker

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code
- Avast Prague C++ Meetup
- Czech National Body in WG21

- Occasional hiker
- Enthusiastic photographer

HANA DUSÍKOVÁ

- Researcher in Avast
  - Improving things
  - High-performance code
- Avast Prague C++ Meetup
- Czech National Body in WG21

- Occasional hiker
- Enthusiastic photographer
- Fast book reader

@HANKADUSIKOVA

#CTRE #CPPNOW

"And if thou gaze long at a finite automaton,
a finite automaton also gazes into thee."

– Friedrich Nietzsche

(after taking a computer science class)

THE
COMPILE TIME REGULAR EXPRESSIONS
LIBRARY

EXAMPLE: BASIC USAGE

```cpp
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

EXAMPLE: BASIC USAGE

```
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

EXAMPLE: BASIC USAGE

```
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

EXAMPLE: BASIC USAGE

```
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

EXAMPLE: BASIC USAGE

```
bool is_a_date(std::string_view input) noexcept {
  return ctre::match<"[0-9]{4}/[0-9]{2}/[0-9]{2}">(input);
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

EXAMPLE: EXTRACTING A VALUE

```cpp
std::optional<std::string_view> extract_sha256(std::string_view input) noexcept {
  if (auto r = ctre::match<"[a-fA-F0-9]{64}">(input)) {
    return r;
  } else {
    return std::nullopt;
  }
}
```

FEATURES OF CTRE

FEATURES OF CTRE

- PCRE2 compatible dialect

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible

FEATURES OF CTRE

- PCRE2 compatible dialect
    - basic unicode support
    - optimizing greedy cycles into possessive if possible
- constexpr

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing
  - constexpr & runtime matching

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing
  - constexpr & runtime matching
  - doesn't support runtime defined patterns

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing
  - constexpr & runtime matching
  - doesn't support runtime defined patterns
- quick regex matching/searching

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing
  - constexpr & runtime matching
  - doesn't support runtime defined patterns
- quick regex matching/searching
  - structured-bindings for extracting captures

FEATURES OF CTRE

- PCRE2 compatible dialect
  - basic unicode support
  - optimizing greedy cycles into possessive if possible
- constexpr
  - compile time constructing
  - constexpr & runtime matching
  - doesn't support runtime defined patterns
- quick regex matching/searching
  - structured-bindings for extracting captures
  - generates compact assembly

TECHNICAL DETAILS OF CTRE

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0
  - gcc 7.4

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0
  - gcc 7.4
  - msvc 15.8.8+

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0
  - gcc 7.4
  - msvc 15.8.8+
- open-source

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0
  - gcc 7.4
  - msvc 15.8.8+
- open-source
  - apache-2.0 license

TECHNICAL DETAILS OF CTRE

- small, header only C++17/20 library (8k LoC)
- supports all major compilers
  - clang 6.0
  - gcc 7.4
  - msvc 15.8.8+
- open-source
  - apache-2.0 license
  - available on github.com

BASIC API

```cpp
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto match(ForwardIterable auto &&, ForwardIterable auto &&) noexcep

  template <fixed_string Pattern>
  constexpr auto search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto search(ForwardIterable auto &&, ForwardIterable auto &&) noexce

}
```

BASIC API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto match(ForwardIterable auto &&, ForwardIterable auto &&) noexcep

  template <fixed_string Pattern>
  constexpr auto search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto search(ForwardIterable auto &&, ForwardIterable auto &&) noexce

}
```

BASIC API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto match(ForwardIterable auto &&, ForwardIterable auto &&) noexcep

  template <fixed_string Pattern>
  constexpr auto search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto search(ForwardIterable auto &&, ForwardIterable auto &&) noexce

}
```

BASIC API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto match(ForwardIterable auto &&, ForwardIterable auto &&) noexcep

  template <fixed_string Pattern>
  constexpr auto search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto search(ForwardIterable auto &&, ForwardIterable auto &&) noexce

}
```

BASIC API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto match(ForwardIterable auto &&, ForwardIterable auto &&) noexcep

  template <fixed_string Pattern>
  constexpr auto search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto search(ForwardIterable auto &&, ForwardIterable auto &&) noexce

}
```

DFA BASED API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr bool fast_match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_match(ForwardIterable auto &&, ForwardIterable auto &&) nc

  template <fixed_string Pattern>
  constexpr bool fast_search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_search(ForwardIterable auto &&, ForwardIterable auto &&) r

}
```

DFA BASED API

```cpp
namespace ctre {

  template <fixed_string Pattern>
  constexpr bool fast_match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_match(ForwardIterable auto &&, ForwardIterable auto &&) n

  template <fixed_string Pattern>
  constexpr bool fast_search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_search(ForwardIterable auto &&, ForwardIterable auto &&) n

}
```

DFA BASED API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr bool fast_match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_match(ForwardIterable auto &&, ForwardIterable auto &&) no

  template <fixed_string Pattern>
  constexpr bool fast_search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_search(ForwardIterable auto &&, ForwardIterable auto &&) n

}
```

DFA BASED API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr bool fast_match(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_match(ForwardIterable auto &&, ForwardIterable auto &&) n

  template <fixed_string Pattern>
  constexpr bool fast_search(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr bool fast_search(ForwardIterable auto &&, ForwardIterable auto &&)

}
```

RETURN TYPE OF THE BASIC API

```cpp
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```cpp
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

RETURN TYPE OF THE BASIC API

```
namespace ctre {

  template <impl-spec> struct regex_results {
    constexpr operator bool() const noexcept;

    constexpr operator basic_string_view<char_type>() const noexcept;
    constexpr explicit operator basic_string<char_type>() const noexcept;

    constexpr auto begin() const noexcept;
    constexpr auto end() const noexcept;

    // number of captures
    constexpr static size_t size() const noexcept;

    // returns object similar to regex_results (without 'get' method)
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: STRUCTURED BINDING

```
// .get<0> is an implicit capture of whole pattern
auto [r, year, month, day] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(inpu

if (r) {
  // do something with successful match
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```cpp
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```cpp
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

EXAMPLE: ITERATE OVER CONCATENATED SEQUENCE OF NUMBERS

```
int sum(std::string_view input) {
  int output = 0;
  for (const auto & capt: ctre::range<"^,?([0-9]+)">(input))
    output += to_integer(capt.get<1>());
  return output;
}
```

ITERABLE API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto range(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto range(ForwardIterable auto &&, ForwardIterable auto &&) noexcept

}
```

ITERABLE API

```
namespace ctre {

  template <fixed_string Pattern>
  constexpr auto range(const Range auto &) noexcept;

  template <fixed_string Pattern>
  constexpr auto range(ForwardIterable auto &&, ForwardIterable auto &&) noexcept

}
```

EXAMPLE: 🥰🦄

```cpp
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}
```

EXAMPLE: 🥰🦄

```
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}
```

EXAMPLE: 🥰🦄

```
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}
```

EXAMPLE: 🥰🦄

```cpp
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}

static_assert( is_emoji_only(U"🥰😱😆"));
```

EXAMPLE: 🥰🦄

```
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}

static_assert( is_emoji_only(U"🥰😱😆"));
static_assert(!is_emoji_only(U"no!😡"));
```

EXAMPLE: 🥰🦄

```
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}

static_assert( is_emoji_only(U"🥰😱😆"));
static_assert(!is_emoji_only(U"no!😡"));
```

*unicode support is not yet fully merged

EXAMPLE: 🥰🦄

```
constexpr bool is_emoji_only(std::u32string_view input) noexcept {
  return ctre::match<"\\p{emoji}+">(input);
}
```

```
static_assert( is_emoji_only(U"🥰😱😆"));
static_assert(!is_emoji_only(U"no!😡"));
```

*unicode support is not yet fully merged

**thanks to Corentin Jabot for providing constexpr unicode tables ❤️

A Save/Load Add new... CppInsights C++

```
#include <ctre.hpp>

bool match(std::string_view input) {
    return ctre::match<"aloha|[a-z]+">(input);
}
```

GCC 9.1 -Os -std=c++2a

A 11010 .LX0: .text // \s+ Intel Demangle Libraries

Add new... Add tool...

Edit on Compiler Explorer

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`
input:

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `a`

`a`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `al`

```
al
```

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alo`

`alo`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `aloh`

`aloh`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `aloha`

```
aloha
```

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`
input: `alohah`

`aloha` (fail, backtrack)

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `a`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `al`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `alo`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `aloh`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `aloha`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohah`

`aloha` (fail, backtrack) `alohah`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohaha`

`aloha` (fail, backtrack) `alohaha`

CAVEATS OF A BACKTRACKING ENGINE

pattern: `aloha|[a-z]+`

input: `alohaha`

`aloha` (fail, backtrack) `alohaha` (accepts)

BACK-TRACKING

A common problem of many regular expression engines.

HOW CAN WE AVOID IT?

HOW CAN WE AVOID IT?

We need to talk (a little bit) about theory.

REGULAR EXPRESSIONS

`hello|aloha|guten tag|dobrý den|bonjour`

`[a-z]+[0-9]+`

A REGULAR EXPRESSION IS (FORMALLY)

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\varnothing$ or $\{\}$),

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\varnothing$ or {}),
- an empty string (ε or {""}),

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\emptyset$ or `{}`),
- an empty string (ε or `{""}`),
- a literal character (e.g. `a` or `{"a"}`),

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\varnothing$ or `{}`),
- an empty string (ε or `{""}`),
- a literal character (e.g. `a` or `{"a"}`),
- the concatenation of two REs ($A \cdot B$),

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\varnothing$ or `{}`),
- an empty string (ε or `{""}`),
- a literal character (e.g. `a` or `{"a"}`),
- the concatenation of two REs ($A \cdot B$),
- the alternation of two REs ($A|B$),

A REGULAR EXPRESSION IS (FORMALLY)

- the empty set ($\varnothing$ or `{}`),
- an empty string (ε or `{""}`),
- a literal character (e.g. `a` or `{"a"}`),
- the concatenation of two REs ($A \cdot B$),
- the alternation of two REs ($A|B$),
- or repetitions (Kleene star) (A^*)

NON-FORMAL SYNTAX CONSTRUCTS

NON-FORMAL SYNTAX CONSTRUCTS

- Optional A ($A?$ is $\varepsilon | A$)

NON-FORMAL SYNTAX CONSTRUCTS

- Optional A ($A?$ is $\varepsilon|A$)
- Plus repetition of A ($A \cdot A^*$)

NON-FORMAL SYNTAX CONSTRUCTS

- Optional A ($A?$ is $\varepsilon|A$)
- Plus repetition of A ($A \cdot A^*$)
- Back-reference

NON-FORMAL SYNTAX CONSTRUCTS

- Optional A ($A?$ is $\varepsilon|A$)
- Plus repetition of A ($A \cdot A^*$)
- ~~Back-reference~~

NON-FORMAL SYNTAX CONSTRUCTS

- Optional A ($A?$ is $\varepsilon|A$)
- Plus repetition of A ($A \cdot A^*$)
- ~~Back-reference~~

not all features of a "regular expression"
implementation is technically a regular expression

EXAMPLE: NON-FORMAL SYNTAX CONSTRUCTS

We are all used to a pattern like this:

`(ct)?re|[a-f]+`

The pattern is equivalent to:

`(ε|ct)re|(a|b|c|d|e|f)(a|b|c|d|e|f)*`

EXAMPLE: NON-FORMAL SYNTAX CONSTRUCTS

We are all used to a pattern like this:

`(ct)?re|[a-f]+`

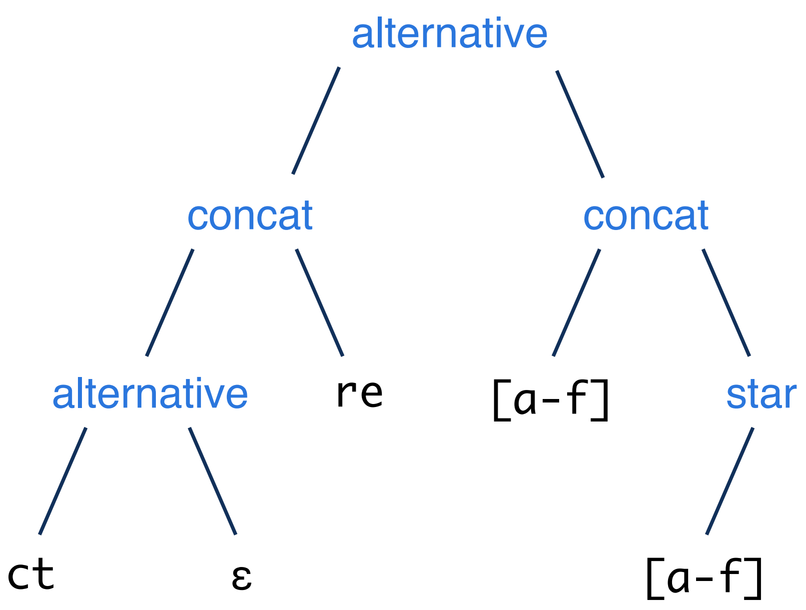
The pattern is equivalent to:

`(ε|ct)re|(a|b|c|d|e|f)(a|b|c|d|e|f)*`

Every pattern can be converted into a formal form.

A REGULAR EXPRESSION CAN BE DESCRIBED AS AN ABSTRACT SYNTAX TREE

`(ct)?re|[a-f]+`



HOW TO STORE THE AST
IN C++ COMPILE-TIME EVALUATED CODE?

HOW TO STORE THE AST
IN C++ COMPILE-TIME EVALUATED CODE?

- tree-like (allocated) data structure

HOW TO STORE THE AST
IN C++ COMPILE-TIME EVALUATED CODE?

- ~~tree-like (allocated) data structure~~ (we can't allocate in constexpr, yet)

HOW TO STORE THE AST IN C++ COMPILE-TIME EVALUATED CODE?

- ~~tree-like (allocated) data structure~~ (we can't allocate in constexpr, yet)
- type based expression
  - expression templates (like boost::xpressive)
  - tuple-like empty types

TYPES AS THE BUILDING BLOCKS

```cpp
// building blocks
struct epsilon { };
template <Character auto C> struct ch { };

// operations
template <typename...> struct concat { };
template <typename...> struct alt { };
template <typename> struct star { };

// for convenient usage
template <typename E> using opt = alternation<E, epsilon>;
template <typename E> using plus = concat<E, star<E>>;
```

EXAMPLE: A REGEX TYPE

```
(ct)?re
```

CONVERTING A PATTERN INTO AN AST: PARSING

HOW TO CONVERT THE PATTERN?

HOW TO CONVERT THE PATTERN?

- Use a generic LL(1) parser for converting a pattern into a type.

HOW TO CONVERT THE PATTERN?

- Use a generic LL(1) parser for converting a pattern into a type.
- The parser uses a provided PCRE compatible grammar.

HOW TO CONVERT THE PATTERN?

- Use a generic LL(1) parser for converting a pattern into a type.
- The parser uses a provided PCRE compatible grammar.
- Output type is the AST.

HOW DOES AN LL(1) PARSER WORK?

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:
  - push a string of symbols to the stack,

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:
  - push a string of symbols to the stack,
  - pop a character from the input,

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:
  - push a string of symbols to the stack,
  - pop a character from the input,
  - or reject.

HOW DOES AN LL(1) PARSER WORK?

- Starts with a start symbol on the stack.
- On every step it pops one symbol from the stack and checks the current character at the input.
- Based on the pair of symbol and character it decides to:
  - push a string of symbols to the stack,
  - pop a character from the input,
  - or reject.
- Repeat until the stack and input are empty then accept.

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) →

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) →

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) →

	(	)	*	+	?	|	other	**ε**
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) → (...)

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) → ε

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) → pop input

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) → reject

	(	)	*	+	?	|	other	**ε**
→ ***S***	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DOES THE GRAMMAR LOOK LIKE?

f(symbol,char) → accept

	(	)	*	+	?	|	other	ε
→ ***S***	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
z_0								accept

HOW DOES AN LL1 PARSER WORK?

input: a*b*ε step: 0

stack: S

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0 alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod seq*	ε				ε	other *mod seq*	ε
terminal	pop	pop	pop	pop	pop	pop	pop	
Z_0								accept

reset a*b*

HOW CAN WE REPRESENT THE SYMBOLS IN C++?

```cpp
struct S {};
struct alt0 {};
struct alt {};
struct mod {};
struct seq0 {};
struct seq {};

using start_symbol = S;
```

HOW CAN WE REPRESENT THE SYMBOLS IN C++?

```
struct S {};
struct alt0 {};
struct alt {};
struct mod {};
struct seq0 {};
struct seq {};

using start_symbol = S;
```

HOW CAN WE REPRESENT AN LL(1) TABLE IN C++?

```
// f (symbol, char) → (...)
// f (symbol, symbol) → pop input
// f (symbol, char) → reject
// f (Z0, ε) → accept
```

HOW CAN WE REPRESENT AN LL(1) TABLE IN C++?

```cpp
// f (symbol, char) → (...)
auto f(symbol, term<'c'>) -> list{...};

// f (symbol, symbol) → pop input


// f (symbol, char) → reject


// f (Z0, ε) → accept
```

HOW CAN WE REPRESENT AN LL(1) TABLE IN C++?

```
// f (symbol, char) → (...)
auto f(symbol, term<'c'>) -> list{...};

// f (symbol, symbol) → pop input
template <auto S> auto f(term<S>, term<S>) -> pop_input;

// f (symbol, char) → reject


// f (Z_0, ε) → accept
```

HOW CAN WE REPRESENT AN LL(1) TABLE IN C++?

```cpp
// f (symbol, char) → (...)
auto f(symbol, term<'c'>) -> list{...};

// f (symbol, symbol) → pop input
template <auto S> auto f(term<S>, term<S>) -> pop_input;

// f (symbol, char) → reject
auto f(...) -> reject;

// f (Z₀, ε) → accept

```

HOW CAN WE REPRESENT AN LL(1) TABLE IN C++?

```cpp
// f (symbol, char) → (...)
auto f(symbol, term<'c'>) -> list{...};

// f (symbol, symbol) → pop input
template <auto S> auto f(term<S>, term<S>) -> pop_input;

// f (symbol, char) → reject
auto f(...) -> reject;

// f (Z0, ε) → accept
auto f(empty_stack, epsilon) -> accept;
```

HOW CAN WE PASS THE GRAMMAR INTO THE PARSER?

```
struct S {};
struct alt0 {};
struct alt {};
// ...

using start_symbol = S;

auto f(...) -> reject;
// ...
```

HOW CAN WE PASS THE GRAMMAR INTO THE PARSER?

```
struct pcre {
  struct S {};
  struct alt0 {};
  struct alt {};
  // ...

  using start_symbol = S;

  auto f(...) -> reject;
  // ...
}
```

HOW IS THE PARSER USED?

```
constexpr bool ok = parser<pcre, "a+b+">::correct;
```

HOW IS THE PARSER IMPLEMENTED?

```cpp
template <typename Grammar, ...> struct parser {
  //...
  auto next_move = Grammar::f(top_of_stack, current_term);
  //...
}
```

HOW IS THE PARSER IMPLEMENTED?

```
template <typename Grammar, ...> struct parser {
  //...
  auto next_move = Grammar::f(top_of_stack, current_term);
  //...
}
```

HOW IS THE PARSER IMPLEMENTED?

```
template <typename Grammar, ...> struct parser {
  //...
  auto next_move = Grammar::f(top_of_stack, current_term);
  //...
}
```

HOW IS THE PARSER IMPLEMENTED?

```
template <typename Grammar, ...> struct parser {
  //...
  auto next_move = Grammar::f(top_of_stack, current_term);
  //...
}
```

HOW DO WE IMPLEMENT THE INPUT STRING?

```
template <typename CharT, size_t N> struct fixed_string {
  CharT data[N+1];
  // constexpr constructor from const char[N]
  constexpr auto operator[](size_t i) const noexcept { return data[i]; }
  constexpr size_t size() const noexcept { return N; }
  constexpr auto operator<=>(const fixed_string &) = default;
};

template <typename CharT, size_t N>
 fixed_string(const CharT[N]) -> fixed_string<CharT, N>;
// more info about class NTTP in p0732 by Jeff Snyder and Louis Dionne
```

HOW DO WE IMPLEMENT THE INPUT STRING?

```
template <typename CharT, size_t N> struct fixed_string {
  CharT data[N+1];
  // constexpr constructor from const char[N]
  constexpr auto operator[](size_t i) const noexcept { return data[i]; }
  constexpr size_t size() const noexcept { return N; }
  constexpr auto operator<=>(const fixed_string &) = default;
};

template <typename CharT, size_t N>
 fixed_string(const CharT[N]) -> fixed_string<CharT, N>;
// more info about class NTTP in p0732 by Jeff Snyder and Louis Dionne
```

HOW DO WE IMPLEMENT THE INPUT STRING?

```
template <typename CharT, size_t N> struct fixed_string {
  CharT data[N+1];
  // constexpr constructor from const char[N]
  constexpr auto operator[](size_t i) const noexcept { return data[i]; }
  constexpr size_t size() const noexcept { return N; }
  constexpr auto operator<=>(const fixed_string &) = default;
};

template <typename CharT, size_t N>
 fixed_string(const CharT[N]) -> fixed_string<CharT, N>;
// more info about class NTTP in p0732 by Jeff Snyder and Louis Dionne
```

HOW DO WE IMPLEMENT THE STACK?

```
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DO WE IMPLEMENT THE STACK?

```
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DO WE IMPLEMENT THE STACK?

```
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DO WE IMPLEMENT THE STACK?

```cpp
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DO WE IMPLEMENT THE STACK?

```
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DO WE IMPLEMENT THE STACK?

```
template <typename... Ts> struct list { };

template <typename... Ts, typename... As>
 constexpr auto push(list<Ts...>, As...) -> list<As..., Ts...>;

template <typename T, typename... As>
 constexpr auto pop(list<T, Ts...>) -> list<Ts...>;

template <typename T, typename... Ts>
 constexpr auto top(list<T, Ts...>) -> T;

struct empty { };
constexpr auto top(list<>) -> empty;
```

HOW DOES IT FIT TOGETHER?

```
constexpr bool ok = parser<pcre, "a*b*">::correct;
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

LL1 CONSTEXPR PARSER

```
template <typename Grammar, fixed_string Str> struct parser {

  static constexpr bool correct = parse(list<Grammar::start_symbol>{});

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S> static constexpr bool parse(S stack) {

    auto symbol = top(stack);
    auto current_term = get_character<Pos>();
```

WE NEED MORE THAN
JUST RETURNING A BOOLEAN.

A PATTERN TO THE AST:
LET THERE BE A TYPE

HOW CAN WE BUILD A TYPE FROM A STRING?

WHERE ARE THE SEMANTIC ACTIONS PLACED?

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other *mod seq alt*	
alt		ε				| *seq0* *alt*		ε
mod	ε	ε	*	+	?	ε	ε	ε
seq0	(*alt0*) *mod seq*						other *mod seq*	
seq	(*alt0*) *mod* *seq*	ε				ε	other *mod* *seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHERE ARE THE SEMANTIC ACTIONS PLACED?

	(	)	*	+	?	|	other	ε
→ *S*	(*alt0*) *mod seq alt*						other char *mod seq alt*	ε
alt0	(*alt0*) *mod seq alt*						other char *mod seq alt*	
alt		ε				| *seq0* alt *alt*		ε
mod	ε	ε	* star	+ plus	? opt	ε	ε	ε
seq0	(*alt0*) *mod seq*						other char *mod seq*	
seq	(*alt0*) *mod* concat *seq*	ε				ε	other char *mod* concat *seq*	ε
(	pop							
)		pop						
*			pop					
+				pop				
?					pop			
|						pop		
other							pop	
Z_0								accept

WHAT DO THE SEMANTIC ACTION SYMBOLS LOOK LIKE?

```
struct pcre {
  struct _char: action {};
  struct alpha: action {};
  struct digit: action {};
  struct seq: action {};
  struct star: action {};
  struct plus: action {};
  struct opt: action {};
  // ...
}
```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{}

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT ARE THE CHANGES TO THE PARSER?

```cpp
template <typename Grammar, fixed_string Str> struct parser {

  template <typename Subject>
  static constexpr auto output = parse(list<Grammar::start_symbol>{}, Subject{})

  // return current term
  template <size_t Pos> constexpr auto get_character() const {
    if constexpr (Pos < Str.size()) return term<Str[Pos]>{};
    else return epsilon{};
  }

  // prepare each step and move to next
  template <size_t Pos = 0, typename S, typename T>
  static constexpr auto parse(S stack, T subject) {

```

WHAT DOES BUILDING FROM A STRING LOOK LIKE?

```
a*b*
```

WHAT ABOUT THE MODIFY FUNCTION?

BUILDING THE AST ON A STACK

```cpp
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```cpp
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```cpp
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```cpp
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```cpp
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;

```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

BUILDING THE AST ON A STACK

```
// pushing a character
template <Character auto C, typename... Ts>
 auto modify(pcre::_char, term<C>, list<Ts...>)
  -> list<ch<C>, Ts...>;

// concatenating a sequence (notice the switched order of A & B on the stack)
template <Character auto C, typename A, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, A, Ts...>)
  -> list<concat<A, B>, Ts...>;

// adding to a concatenated sequence
template <Character auto C, typename... As, typename B, typename... Ts>
 auto modify(pcre::seq, term<C>, list<B, concat<As...>, Ts...>)
  -> list<concat<As..., B>, Ts...>;
```

WE HAVE THE AST!

```
static_assert(std::is_same_v<
  ,
  decltype( top(parser<pcre,"">::type) )
>);
```

WE HAVE THE AST!

```
static_assert(std::is_same_v<
  ,
  decltype( top(parser<pcre,"">::type) )
>);
```

WE HAVE THE AST!

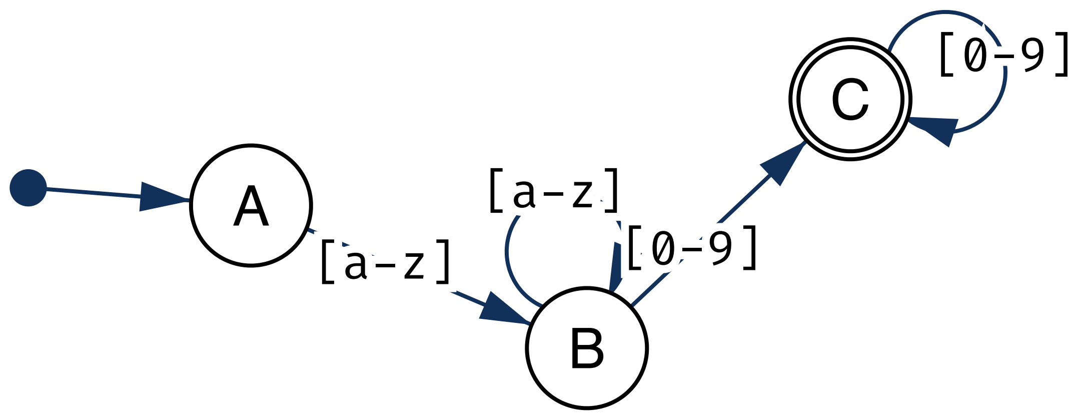
```
static_assert(std::is_same_v<
  ,
  decltype( top(parser<pcre,"">::type) )
>);
```

WE HAVE THE AST!

```
static_assert(std::is_same_v<
  ,
  decltype( top(parser<pcre,"">::type) )
>);
```

HOW DO WE MATCH A REGULAR EXPRESSION IN THE FORM OF AN AST?

FINITE AUTOMATON



WHAT'S A FINITE AUTOMATON?

$(Q, \Sigma, \delta, q_0, F)$

WHAT'S A FINITE AUTOMATON?

$$(Q, \Sigma, \delta, q_0, F)$$

- a finite set of states Q

WHAT'S A FINITE AUTOMATON?

$$(Q, \Sigma, \delta, q_0, F)$$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ

WHAT'S A FINITE AUTOMATON?

$$(Q, \Sigma, \delta, q_0, F)$$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \to$
  - $P(Q)$ (nondeterministic FA)
  - Q (deterministic FA)

WHAT'S A FINITE AUTOMATON?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - $P(Q)$ (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$

WHAT'S A FINITE AUTOMATON?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - $P(Q)$ (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

WHAT'S A FINITE AUTOMATON?

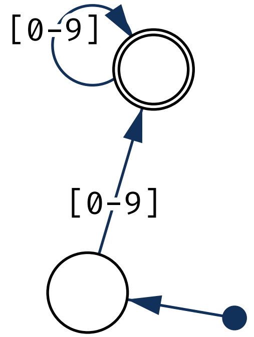
$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \to$
  - $P(Q)$ (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

A finite automaton accepts exactly the same class of languages as a regular expression.

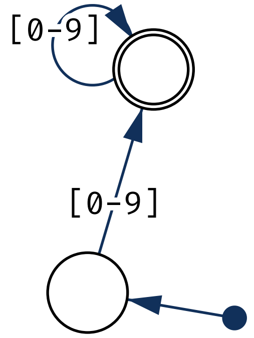
EXAMPLE: INTEGRAL NUMBERS

[0-9]+



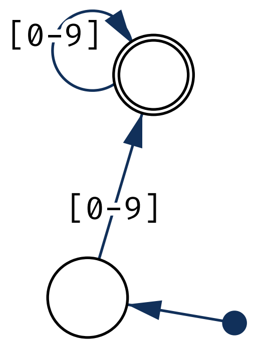
EXAMPLE: INTEGRAL NUMBERS

`[0-9]+`



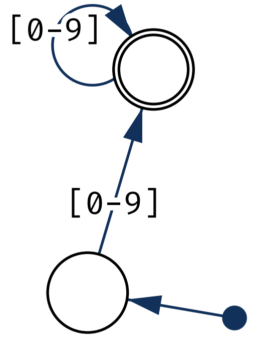
EXAMPLE: INTEGRAL NUMBERS

[0-9]+



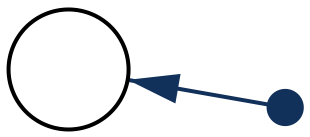
EXAMPLE: INTEGRAL NUMBERS

[0-9]+

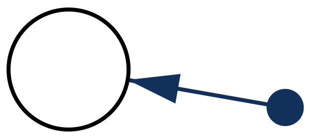


HOW DO YOU CONVERT A REGEX INTO AN FA?

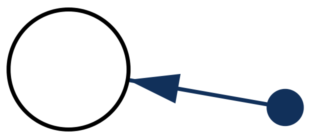
∅ (EMPTY SET)



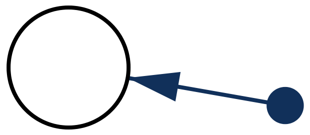
∅ (EMPTY SET)



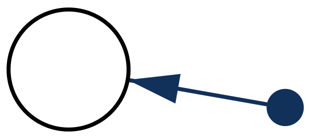
∅ (EMPTY SET)



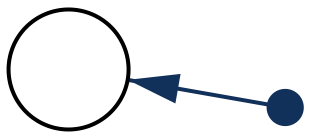
∅ (EMPTY SET)



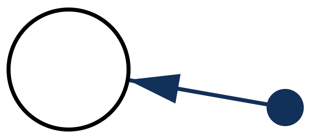
∅ (EMPTY SET)



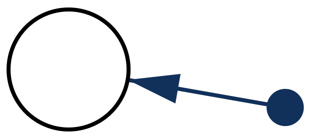
∅ (EMPTY SET)



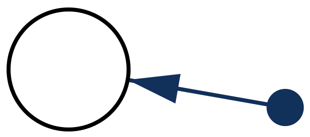
∅ (EMPTY SET)



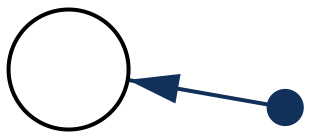
∅ (EMPTY SET)



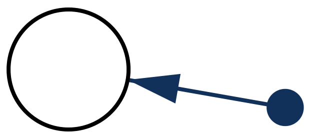
∅ (EMPTY SET)



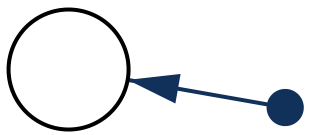
∅ (EMPTY SET)



∅ (EMPTY SET)



∅ (EMPTY SET)



THERE IS A DIRECT MAPPING BETWEEN
THE RE DEFINITION AND OPERATIONS OVER THE FAs.

THERE IS A DIRECT MAPPING BETWEEN THE RE DEFINITION AND OPERATIONS OVER THE FAs.

And we need to map it onto C++ code.

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \to$
  - $P(Q)$ (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q – `set<int>`
- a finite set of input symbols (the alphabet) Σ
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - P(Q) (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

HOW TO REPRESENT AN FA IN C++?

$$(Q, \Sigma, \delta, q_0, F)$$

- a finite set of states Q – `set<int>`
- a finite set of input symbols (the alphabet) Σ – `char32_t`
- a transition function $\delta: Q \times \Sigma \to$
  - P(Q) (nondeterministic FA)
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

HOW TO REPRESENT AN FA IN C++?

$$(Q, \Sigma, \delta, q_0, F)$$

- a finite set of states Q – `set<int>`
- a finite set of input symbols (the alphabet) Σ – `char32_t`
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - P(Q) (nondeterministic FA) – `set<tuple<int, int, char32_t>>`
  - Q (deterministic FA)
- a start state $q_0 \in Q$
- a set of accept (final) states $F \subseteq Q$

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q – `set<int>`
- a finite set of input symbols (the alphabet) Σ – `char32_t`
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - $P(Q)$ (nondeterministic FA) – `set<tuple<int, int, char32_t>>`
  - Q (deterministic FA)
- a start state $q_0 \in Q$ – `int(0)`
- a set of accept (final) states $F \subseteq Q$

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q – `set<int>`
- a finite set of input symbols (the alphabet) Σ – `char32_t`
- a transition function $\delta: Q \times \Sigma \to$
  - $P(Q)$ (nondeterministic FA) – `set<tuple<int, int, char32_t>>`
  - Q (deterministic FA)
- a start state $q_0 \in Q$ – `int(0)`
- a set of accept (final) states $F \subseteq Q$ – `set<int>`

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q – ~~set<int>~~
- a finite set of input symbols (the alphabet) Σ – char32_t
- a transition function $\delta: Q \times \Sigma \rightarrow$
  - P(Q) (nondeterministic FA) – set<tuple<int, int, char32_t>>
  - Q (deterministic FA)
- a start state $q_0 \in Q$ – int(0)
- a set of accept (final) states $F \subseteq Q$ – set<int>

HOW TO REPRESENT AN FA IN C++?

$(Q, \Sigma, \delta, q_0, F)$

- a finite set of states Q – ~~`set<int>`~~ `int` (implicit)
- a finite set of input symbols (the alphabet) Σ – `char32_t`
- a transition function $\delta: Q \times \Sigma \to$
  - $P(Q)$ (nondeterministic FA) – `set<tuple<int, int, char32_t>>`
  - Q (deterministic FA)
- a start state $q_0 \in Q$ – `int(0)`
- a set of accept (final) states $F \subseteq Q$ – `set<int>`

CONSTEXPR IMPLEMENTATION OF A FA

"There is no such thing as a zero-cost abstraction."

– Chandler Carruth

CONSTEXPR PROBLEMS

CONSTEXPR PROBLEMS

- every operation counts

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits (and different in every compiler)

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits (and different in every compiler)
- move semantics don't matter

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits (and different in every compiler)
- move semantics don't matter
- problematic error handling & debugging

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits (and different in every compiler)
- move semantics don't matter
- problematic error handling & debugging
  - use a conditional UB or throw-statement as form of an assert

CONSTEXPR PROBLEMS

- every operation counts
  - there are limits (and different in every compiler)
- move semantics don't matter
- problematic error handling & debugging
  - use a conditional UB or throw-statement as form of an assert
  - printf styled debugging

```
template <typename T> struct identify_type;
```

THE CONSTEXPR PROBLEM

THE CONSTEXPR PROBLEM

slooooooooow

THE CONSTEXPR PROBLEM

slooooooooooow

2 minutes

(NFA determinization)

THE CONSTEXPR PROBLEM

slooooooooow

2 minutes (gcc 9.1)

(NFA determinization)

THE CONSTEXPR PROBLEM

sloooooooooow

2 minutes (gcc 9.1)
40 seconds

(NFA determinization)

THE CONSTEXPR PROBLEM

slooooooooooow

2 minutes (gcc 9.1)
40 seconds (clang 8)

(NFA determinization)

THE CONSTEXPR PROBLEM

sloooooooooow

2 minutes (gcc 9.1)
40 seconds (clang 8)
< 5 second

(NFA determinization)

THE CONSTEXPR PROBLEM

slooooooooow

2 minutes (gcc 9.1)
40 seconds (clang 8)
< 5 second (php 7.1)

(NFA determinization)

CONSTEXPR set

```
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR set

```
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR `set`

```
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR set

```cpp
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR `set`

```cpp
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR set

```
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR set

```cpp
template <typename T> class set {

public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR `fixed_set`

```
template <size_t Sz, typename T> class fixed_set {
  T data[Sz];
public:
  constexpr auto begin();
  constexpr auto end();
  constexpr auto begin() const;
  constexpr auto end() const;

  constexpr size_t size() const;
  constexpr auto & operator[](size_t);
  constexpr const auto & operator[](size_t) const;

  constexpr auto insert(T);

  template <typename K> constexpr auto erase(K);
```

CONSTEXPR finite_automaton

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

CONSTEXPR `finite_automaton`

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

CONSTEXPR `finite_automaton`

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

CONSTEXPR `finite_automaton`

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

CONSTEXPR finite_automaton

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

CONSTEXPR `finite_automaton`

```
struct transition {
  int source;
  int target;
  char32_t term;

  constexpr transition(int, int, char32_t);
  constexpr bool match(char32_t c) const;
  // comparable with int against source
};

template <size_t Tr, size_t FSz> struct finite_automaton {
  fixed_set<Tr, transition> transitions;
  fixed_set<FSz, int> final_states;

  // normal constructor / copy constructor ...
```

BASIC BUILDING BLOCKS

```
static constexpr auto empty = finite_automaton<0,0>{};

static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};

template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
  {transition(0, 1, C)},
  {1}
};
```

BASIC BUILDING BLOCKS

```
static constexpr auto empty = finite_automaton<0,0>{};

static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};

template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
  {transition(0, 1, C)},
  {1}
};
```

BASIC BUILDING BLOCKS

```
static constexpr auto empty = finite_automaton<0,0>{};

static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};

template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
  {transition(0, 1, C)},
  {1}
};
```

BASIC BUILDING BLOCKS

```
static constexpr auto empty = finite_automaton<0,0>{};

static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};

template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
  {transition(0, 1, C)},
  {1}
};
```

BASIC MANIPULATION

```
auto operator>>(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rh

auto operator|(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rhs

auto star(const FiniteAutomaton auto & lhs);
```

BASIC MANIPULATION

```
auto operator>>(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rh

auto operator|(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rhs

auto star(const FiniteAutomaton auto & lhs);
```

BASIC MANIPULATION

```
auto operator>>(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rh

auto operator|(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rhs

auto star(const FiniteAutomaton auto & lhs);
```

BASIC MANIPULATION

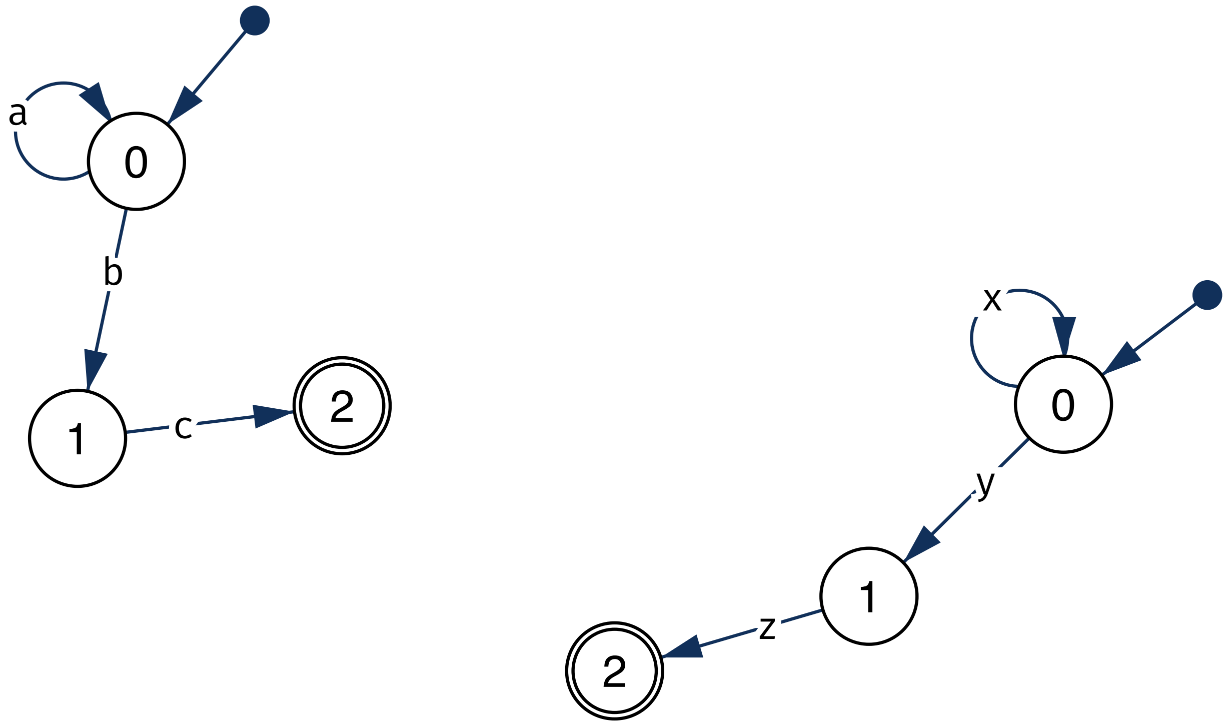
```
auto operator>>(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rh

auto operator|(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rhs

auto star(const FiniteAutomaton auto & lhs);
```

BASIC MANIPULATION

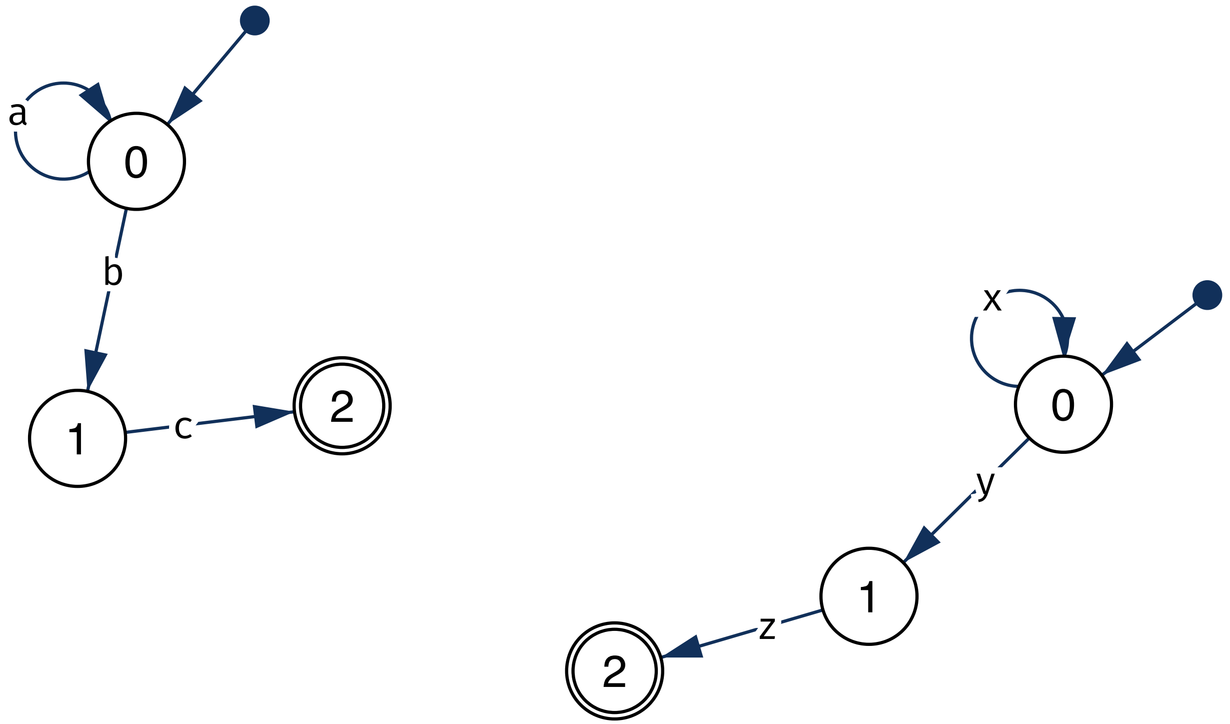
```
auto operator>>(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rh

auto operator|(const FiniteAutomaton auto & lhs, const FiniteAutomaton auto & rhs

auto star(const FiniteAutomaton auto & lhs);
```

Result of operation is dependent
not just on a input type but also on a input value.

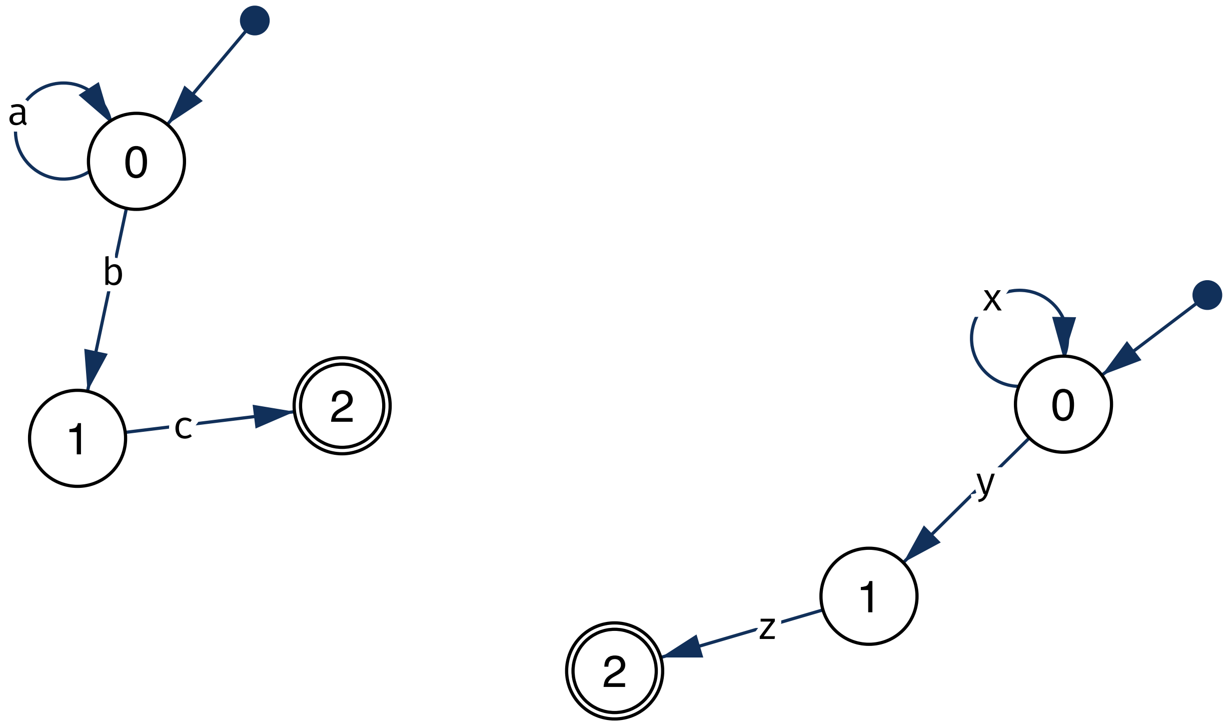
CONCATENATION ALGORITHM



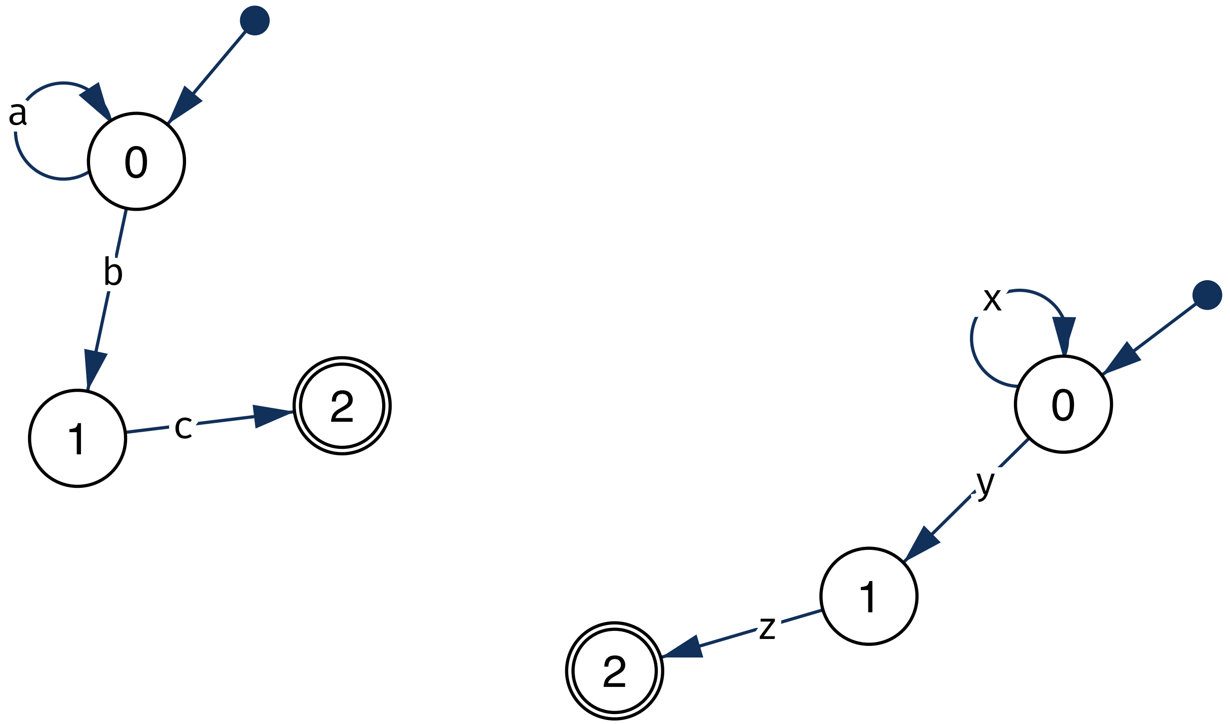
CONCATENATION ALGORITHM



CONCATENATION ALGORITHM



CONCATENATION ALGORITHM



CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()
    
    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```cpp
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

CONCATENATION ALGORITHM

```
template <finite_automaton Lhs, finite_automaton Rhs> struct concat_two {

  constexpr static auto calculate() {
    constexpr size_t tr_count = Lhs.transitions.size() + Rhs.transitions.size();
    constexpr size_t tr_start = Rhs.count_transitions(0);

    constexpr int prefix = Lhs.get_max_state() + 1;

    finite_automaton<(tr_count + tr_start), Rhs.final_states.count()> out;

    copy(Lhs.transitions.begin(), Lhs.transitions.end(), out.transitions.begin()

    for (int f: Lhs.final_states) {
      Rhs.foreach_transition_from(0, [&](transition t){
        t.source = f;
```

OTHER ALGORITHMS

OTHER ALGORITHMS

- alternation

OTHER ALGORITHMS

- alternation
- repeating (Kleene star)

OTHER ALGORITHMS

- alternation
- repeating (Kleene star)

Both other algorithms work the same,
pre-calculate the size and <u>then</u> populate the content.

TEMPLATE STYLE MANIPULATION

```
template <finite_automaton... Fas>
static constexpr auto concat =

template <finite_automaton... Fas>
static constexpr auto alternation =

template <finite_automaton... Fas>
static constexpr auto star =
```

TEMPLATE STYLE MANIPULATION

```
template <finite_automaton... Fas>
static constexpr auto concat = multi_helper<concat_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto alternation = multi_helper<alternation_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto star = star_one<concat<Fas...>>::value;
```

TEMPLATE STYLE MANIPULATION

```
template <finite_automaton... Fas>
static constexpr auto concat = multi_helper<concat_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto alternation = multi_helper<alternation_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto star = star_one<concat<Fas...>>::value;
```

TEMPLATE STYLE MANIPULATION

```
template <finite_automaton... Fas>
static constexpr auto concat = multi_helper<concat_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto alternation = multi_helper<alternation_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto star = star_one<concat<Fas...>>::value;
```

TEMPLATE STYLE MANIPULATION

```
template <finite_automaton... Fas>
static constexpr auto concat = multi_helper<concat_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto alternation = multi_helper<alternation_two, Fas...>::value;

template <finite_automaton... Fas>
static constexpr auto star = star_one<concat<Fas...>>::value;
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

RECURSIVE APPLICATION OF A BINARY FUNCTION

```
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton... Arg
>
struct multi_helper;



// specialization for 2+
template <
  template <finite_automaton, finite_automaton> Op,
  finite_automaton A,
  finite_automaton B,
  finite_automaton... Tail
>
```

WE HAVE A CLASS-VALUE BASED TEMPLATE META-PROGRAMMING!

WE HAVE A CLASS-VALUE BASED TEMPLATE META-PROGRAMMING!

And if you use `const auto &` it's C++17 compatible!

```
constexpr auto fa = concat<one_char<'a'>, one_char<'b'>>;
```

```
constexpr auto fa = concat<one_char<'a'>, one_char<'b'>>;
```

- I called it a "compile time" function with compile-time arguments.

```
constexpr auto fa = concat<one_char<'a'>, one_char<'b'>>;
```

- I called it a "compile time" function with compile-time arguments.
- Unlike a constexpr function, it can take constexpr variables as arguments and maintain their "constexpr-bility".

```cpp
constexpr auto fa = concat<one_char<'a'>, one_char<'b'>>;
```

- I called it a "compile time" function with compile-time arguments.
- Unlike a constexpr function, it can take constexpr variables as arguments and maintain their "constexpr-bility".
- It makes sure the value is cached.

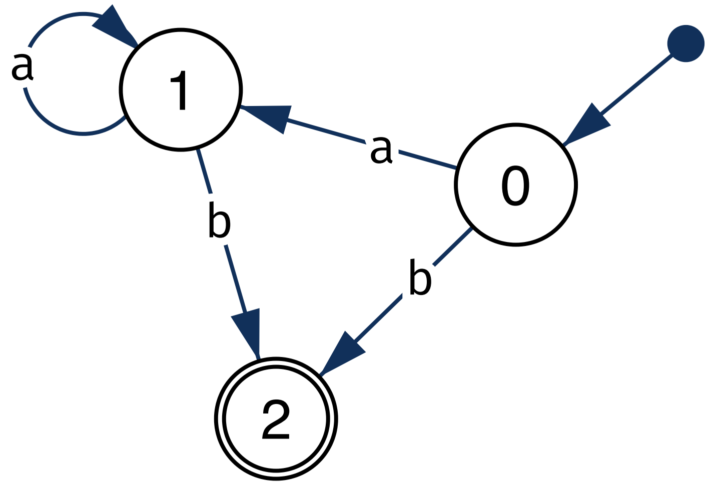
```
constexpr auto fa = concat<one_char<'a'>, one_char<'b'>>;
```

- I called it a "compile time" function with compile-time arguments.
- Unlike a constexpr function, it can take constexpr variables as arguments and maintain their "constexpr-bility".
- It makes sure the value is cached.
- It looks nice :)

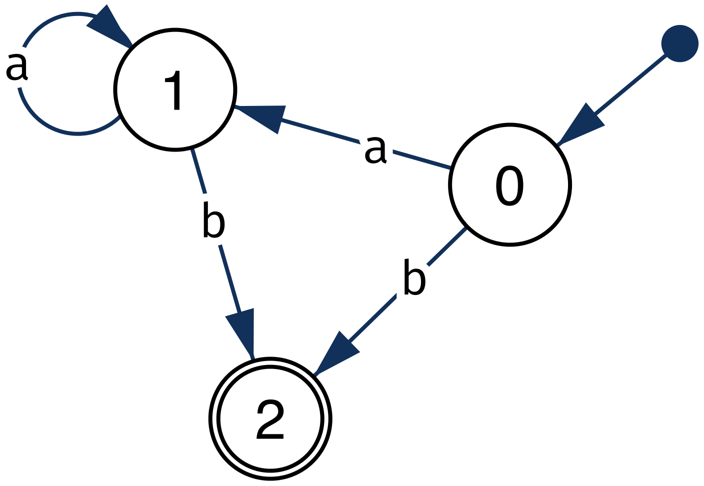
MINIMIZATION

Remove unnecessary states and links by merging the same states.

MINIMIZATION ALGORITHM



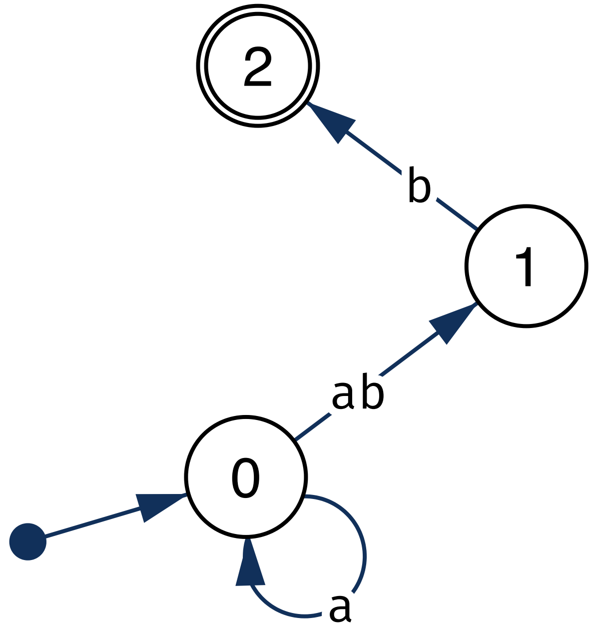
MINIMIZATION ALGORITHM



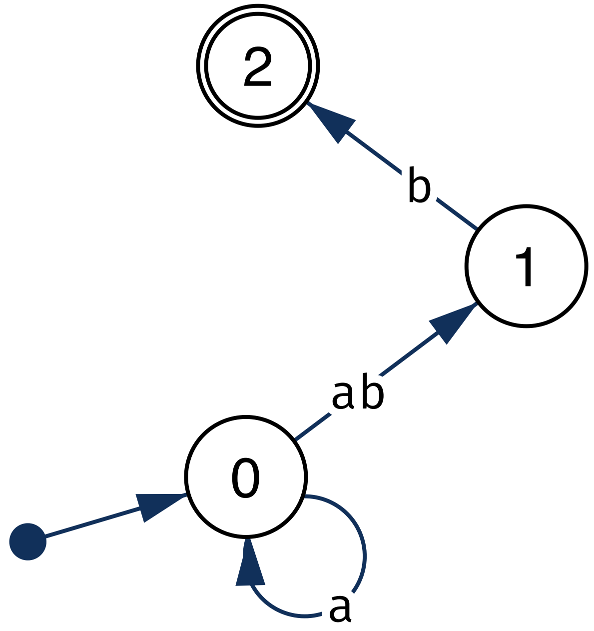
DETERMINIZATION

Remove nondeterministic transitions, can generate 2^n new states.

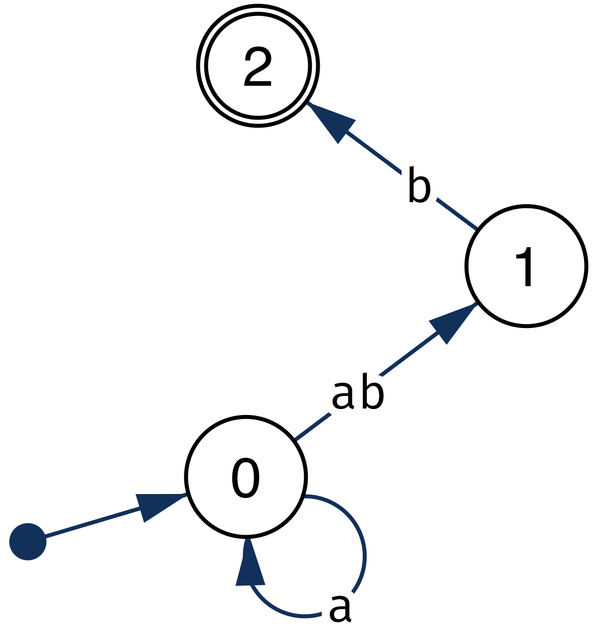
DETERMINIZATION ALGORITHM



DETERMINIZATION ALGORITHM



DETERMINIZATION ALGORITHM



DETERMINIZATION

Remove nondeterministic transitions, can generate 2^n new states.

DETERMINIZATION

Remove nondeterministic transitions, can generate 2^n new states.

Constexpr implementation must be iterative,
as the whole output can't be calculated in one step.

DETERMINIZATION

Remove nondeterministic transitions, can generate 2^n new states.

Constexpr implementation must be iterative,
as the whole output can't be calculated in one step.

It's useful to minimize the FA after the determinization.

WE HAVE A DETERMINISTIC FINITE AUTOMATON

MATCHING A REGULAR EXPRESSION

```
"hello" =~ /aloha|[a-z]+/
```

MATCHING AND SEARCHING

```
ctre::match<"aloha|[a-z]+">("aloha"sv);

// search(x) is actually match(.*x.*)
ctre::search<"aloha|[a-z]+">("...aloha..."sv);
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING A RE'S AST INTO AN FA

```
namespace fa {
  static constexpr auto empty = finite_automaton<0, 0>{};
}

template <typename RE>
static constexpr auto nfa_from = convert(fa::empty, list<RE>{});
```

`RE` is a type, `finite_automaton` is a value.

CONVERTING THE EMPTY RE

```
auto convert(const FinAutomaton auto seed, list<>) {
  // for an empty RE seed is empty FA

  return seed;

}
```

CONVERTING THE EMPTY RE

```
auto convert(const FinAutomaton auto seed, list<>) {
  // for an empty RE seed is empty FA

  return seed;

}
```

CONVERTING THE EMPTY RE

```
auto convert(const FinAutomaton auto seed, list<>) {
  // for an empty RE seed is empty FA

  return seed;

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```cpp
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING EPSILON (EMPTY STRING)

```
namespace fa {
  // epsilon is FA with a start state final
  static constexpr auto epsilon = finite_automaton<0, 1>{{}, {0}};
}

template <typename... Ts>
auto convert(const FinAutomaton auto lhs, list<epsilon, Ts...>) {

  return convert(fa::concat<lhs, fa::epsilon>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```cpp
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CHARACTER

```
namespace fa {
  // epsilon is FA with a start state final
  template <char32_t Term> static constexpr auto
  character = finite_automaton<0, 1>{{transition{0, 1, Term}}, {1}};
}

template <char32_t Term, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<character<Term>, Ts...>) {

  return convert(fa::concat<lhs, fa::character<Term>>, list<Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING A CONCAT SEQUENCE

```
template <typename... Content, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<concat<Content...>, Ts...>) {

  return convert(lhs, list<Content..., Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```cpp
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING AN ALTERNATION

```
template <typename... Options, typename... Ts>
auto convert(const FinAutomaton auto lhs, list<alt<Options...>, Ts...>) {

  constexpr auto inner = fa::alternative<convert(fa::empty, Options)...>;

  return convert(fa::concat<lhs, inner>, list<Ts...>{});

}
```

CONVERTING A KLEENE STAR (CYCLE)

```
template <typename Content, typename Ts...>
auto convert(const FinAutomaton auto seed, list<star<Content>, Ts...>) {

  return convert(fa::concat<lhs, fa::star<Content>>, list<Ts...>{});

}
```

CONVERTING A KLEENE STAR (CYCLE)

```
template <typename Content, typename Ts...>
auto convert(const FinAutomaton auto seed, list<star<Content>, Ts...>) {

  return convert(fa::concat<lhs, fa::star<Content>>, list<Ts...>{});

}
```

CONVERTING A KLEENE STAR (CYCLE)

```
template <typename Content, typename Ts...>
auto convert(const FinAutomaton auto seed, list<star<Content>, Ts...>) {

  return convert(fa::concat<lhs, fa::star<Content>>, list<Ts...>{});

}
```

CONVERTING A KLEENE STAR (CYCLE)

```
template <typename Content, typename Ts...>
auto convert(const FinAutomaton auto seed, list<star<Content>, Ts...>) {

  return convert(fa::concat<lhs, fa::star<Content>>, list<Ts...>{});

}
```

CONVERTING A KLEENE STAR (CYCLE)

```
template <typename Content, typename Ts...>
auto convert(const FinAutomaton auto seed, list<star<Content>, Ts...>) {

  return convert(fa::concat<lhs, fa::star<Content>>, list<Ts...>{});

}
```

WE HAVE A FINITE AUTOMATON,
NOW WE NEED TO DO THE MATCHING.

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

DETERMINISTIC MATCH WRAPPER

```
template <fixed_string re> bool fast_match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  constexpr auto dfa = fa::minimize<fa::determinize<nfa_from<RE>>>;

  return match_state<dfa>(rng.begin(), rng.end());
}
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```cpp
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```cpp
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A STATE

```
template <finite_automaton DFA, int State = 0, typename Iterator>
constexpr bool match_state(Iterator it, Iterator end) noexcept {

  // we can end correctly only if this state is final
  if constexpr (DFA.is_final(State)) {
    if (end == it) return true;
  } else {
    if (end == it) return false;
  }

  // transitions are sorted by source state
  constexpr auto index = DFA.first_transition_index_of(State);

  // tail-recursion
  return choose_transition<DFA, index, State>(it, end);
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```cpp
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```cpp
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```cpp
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

MATCHING A TRANSITION

```
template <finite_automaton DFA, int Index, int State, typename Iterator>
constexpr bool choose_transition(Iterator it, const Iterator end) noexcept {
  // check if we are still in same state
  if constexpr (Index >= DFA.transitions.size()
    || DFA.transitions[Index].source != State) {

    return false;
  }

  if (DFA.transitions[Index].match(*it)) {
    // this transition is the one, go to next state (tail recursion)
    return match_state<DFA, DFA.transitions[Index].target>(std::next(it), end);
  }

  // try next one (also tail recursion)
```

THE DETERMINISTIC MATCHING

- $O(n * t)$ time complexity
- $O(1)$ dynamic memory complexity

n = length of the input, t = average number of transitions per state

THE DETERMINISTIC MATCHING

- O(n * t) time complexity
- O(1) dynamic memory complexity

n = length of the input, t = average number of transitions per state

- doesn't support capture tracking

THE DETERMINISTIC MATCHING

- $O(n * t)$ time complexity
- $O(1)$ dynamic memory complexity

n = length of the input, t = average number of transitions per state

- doesn't support capture tracking
- theoretically the quickest approach

THE DETERMINISTIC MATCHING

- $O(n * t)$ time complexity
- $O(1)$ dynamic memory complexity

n = length of the input, t = average number of transitions per state

- doesn't support capture tracking
- theoretically the quickest approach
- allows streamed data

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

BACKTRACKING MATCH WRAPPER

```
template <fixed_string re> bool match(const Range auto & rng) {
  static_assert(parser<pcre, re>::correct);
  using RE = parser<pcre, re>::type;

  auto out = evaluate(rng.begin(), rng.end(), list<RE>{});

  return out.success && out.it == rng.end();
}
```

RETURN TYPE OF EVALUATE(...)

```
template <ForwardIterator It> struct result {
  bool success;
  ForwardIterator it;
};
```

RETURN TYPE OF EVALUATE(...)

```
template <ForwardIterator It> struct result {
  bool success;
  ForwardIterator it;
};
```

RETURN TYPE OF EVALUATE(...)

```
template <ForwardIterator It> struct result {
  bool success;
  ForwardIterator it;
};
```

MATCHING EPSILON (EMPTY STRING)

```
template <ForwardIterator It, typename... Ts>
result<It> evaluate(It it, It end, list<epsilon, Ts...>) {
  // tail-recursion => no backtracking
  return evaluate(it, end, list<Ts...>{});
}
```

MATCHING EPSILON (EMPTY STRING)

```
template <ForwardIterator It, typename... Ts>
result<It> evaluate(It it, It end, list<epsilon, Ts...>) {
  // tail-recursion => no backtracking
  return evaluate(it, end, list<Ts...>{});
}
```

MATCHING EPSILON (EMPTY STRING)

```
template <ForwardIterator It, typename... Ts>
result<It> evaluate(It it, It end, list<epsilon, Ts...>) {
  // tail-recursion => no backtracking
  return evaluate(it, end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```cpp
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CHARACTER

```
template <ForwardIterator It, auto C, typename... Ts>
result<It> evaluate(It it, It end, list<ch<C>, Ts...>) {

  if (it == end) return {it, false};
  if (*it != C) return {it, false};

  // tail-recursion => no backtracking
  return evaluate(std::next(it), end, list<Ts...>{});
}
```

MATCHING A CONCATENATED SEQUENCE

```
template <ForwardIterator It, typename... Seq, typename... Ts>
result<It> evaluate(It it, It end, list<concat<Seq...>, Ts...>) {

  // tail-recursion => no backtracking
  return evaluate(it, end, list<Seq..., Ts...>{});

}
```

MATCHING A CONCATENATED SEQUENCE

```
template <ForwardIterator It, typename... Seq, typename... Ts>
result<It> evaluate(It it, It end, list<concat<Seq...>, Ts...>) {

  // tail-recursion => no backtracking
  return evaluate(it, end, list<Seq..., Ts...>{});

}
```

MATCHING A CONCATENATED SEQUENCE

```
template <ForwardIterator It, typename... Seq, typename... Ts>
result<It> evaluate(It it, It end, list<concat<Seq...>, Ts...>) {

  // tail-recursion => no backtracking
  return evaluate(it, end, list<Seq..., Ts...>{});

}
```

MATCHING A CONCATENATED SEQUENCE

```
template <ForwardIterator It, typename... Seq, typename... Ts>
result<It> evaluate(It it, It end, list<concat<Seq...>, Ts...>) {

  // tail-recursion => no backtracking
  return evaluate(it, end, list<Seq..., Ts...>{});

}
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{}));
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{}));
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{});
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{});
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{}));
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{});
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING AN ALTERNATE

```
template <ForwardIterator It, typename Head, typename... Tail, typename... Ts>
result<It> evaluate(It it, It end, list<alt<Head, Tail...>, Ts...>) {

  // recursion => backtracking point
  if (auto out = evaluate(it, end, list<Head, Ts...>{})) {
    return out;
  } else {
    // tail-recursion => no backtracking
    return evaluate(it, end, list<alt<Tail...>, Ts...>{}));
  }
}

template <ForwardIterator It, typename... Ts>
 result<It> evaluate(It it, It end, list<alt<>, Ts...>) {
  // no option was successful
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```cpp
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

MATCHING A KLEENE STAR (CYCLE)

```
template <ForwardIterator It, typename Star, typename... Ts>
result<It> match(It it, It end, list<star<Star>, Ts...>) {

  for (;;) {
    // recursion => backtracking point
    if (auto out = match(it, end, list<Ts...>{})) {
      return out;
    }

    // recursion => backtracking point
    if (auto inner = match(it, end, list<Star..., end_of_cycle>{})) {
      it = inner.it;
    } else {
      // inner cycle failed and outer too => whole cycle fail
      return {it, false};
```

THE BACKTRACKING MATCHING

- $O(n * 2^b)$ time complexity
- $O(b)$ dynamic memory complexity

b = number of backtracking points, n = length of the input

THE BACKTRACKING MATCHING

- $O(n * 2^b)$ time complexity
- $O(b)$ dynamic memory complexity

b = number of backtracking points, n = length of the input

- easiest to implement

THE BACKTRACKING MATCHING

- $O(n * 2^b)$ time complexity
- $O(b)$ dynamic memory complexity

b = number of backtracking points, n = length of the input

- easiest to implement
- supports capture tracking

THE BACKTRACKING MATCHING

- $O(n * 2^b)$ time complexity
- $O(b)$ dynamic memory complexity

b = number of backtracking points, n = length of the input

- easiest to implement
- supports capture tracking
- slow worst case scenario

THE BACKTRACKING MATCHING

- $O(n * 2^b)$ time complexity
- $O(b)$ dynamic memory complexity

b = number of backtracking points, n = length of the input

- easiest to implement
- supports capture tracking
- slow worst case scenario
- doesn't allow streamed data

NFA WITHOUT BACKTRACKING

NFA WITHOUT BACKTRACKING

- supports capture tracking

NFA WITHOUT BACKTRACKING

- supports capture tracking
- slower to match, but much quicker to build (than DFA)

NFA WITHOUT BACKTRACKING

- supports capture tracking
- slower to match, but much quicker to build (than DFA)
- allows streamed data

NFA WITHOUT BACKTRACKING

- supports capture tracking
- slower to match, but much quicker to build (than DFA)
- allows streamed data

But I haven't implemented it yet

NFA WITHOUT BACKTRACKING

- supports capture tracking
- slower to match, but much quicker to build (than DFA)
- allows streamed data

But I haven't implemented it yet (some future talk maybe)

COMPARISON

LET'S READ SOME ASSEMBLY :)

A Save/Load Add new... CppInsights C++

GCC 9.1 -Os -std=c++2a

A 11010 .LX0: .text // \s+ Intel Demangle

```
#include <regex>
#include <string_view>

static std::regex re("[a-z0-9]+abc[0-9]");

bool match(std::string_view line) {
    return std::regex_match(line.begin(), line.end(), re);
}
```

1

Edit on Compiler Explorer

Save/Load Add new... CppInsights C++

```cpp
#include <boost/regex.hpp>
#include <string_view>

static boost::regex re("[a-z0-9]+abc[0-9]");

bool match(std::string_view line) {
    return boost::regex_match(line.begin(), line.end(), re)
}
```

GCC 9.1 -Os -std=c++2a

11010 .LX0: .text // \s+ Intel Demangle

1

Edit on Compiler Explorer

A Save/Load Add new... CppInsights C++

```cpp
#include <boost/xpressive/xpressive.hpp>

using svregex = boost::xpressive::basic_regex<std::string_

static svregex re
= (+boost::xpressive::alnum)
>> 'a' >> 'b' >> 'c'
>> boost::xpressive::digit;

bool match(std::string_view line) {
    return bool(boost::xpressive::regex_search(line, re));
}
```

GCC 9.1 -Os -std=c++2a

A 11010 .LX0: .text // \s+ Intel Demangle

```
1
```

Edit on Compiler Explorer

A Save/Load Add new... CppInsights C++

```cpp
#include <ctre.hpp>

bool match(std::string_view line) {
    return ctre::match<"[a-z0-9]+abc[0-9]">(line);
}
```

GCC 9.1 -Os -std=c++2a

A 11010 .LX0: .text // \s+ Intel Demangle

Libraries Add new... Add tool...

```
1
```

Edit on Compiler Explorer

BENCHMARKS

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

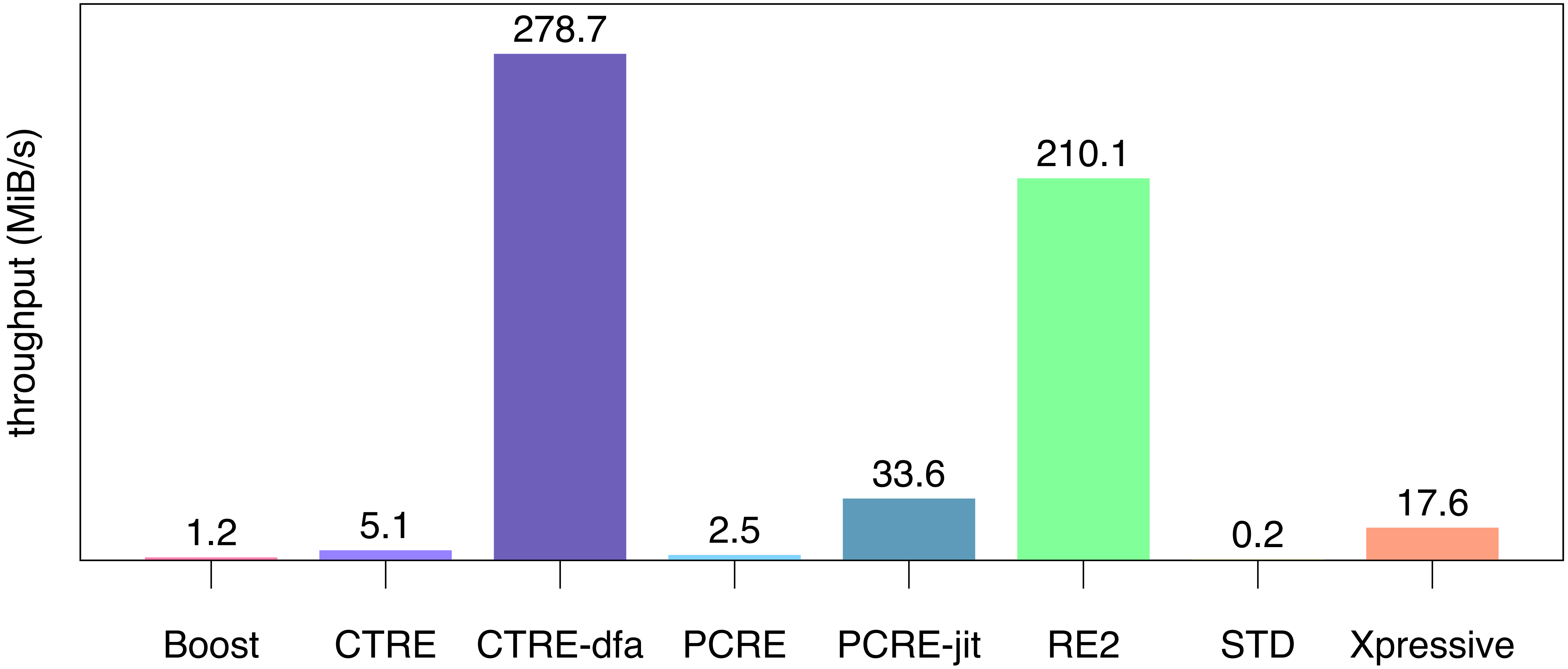
MEASURED CODE (GREP-LIKE UTILITY)

```
int main(int argc, char ** argv) {
  auto re = PREPARE("PATTERN");
  auto lines = load_file_by_lines(argv[1]);

  size_t count = 0;

  // some warm-up

  auto start = steady_clock::now();
  for (string_view line: lines) {
    if (re.SEARCH(line)) count++;
  }
  auto end = steady_clock::now();

  cout << count << "\n";
```

MEASUREMENT METHODOLOGY

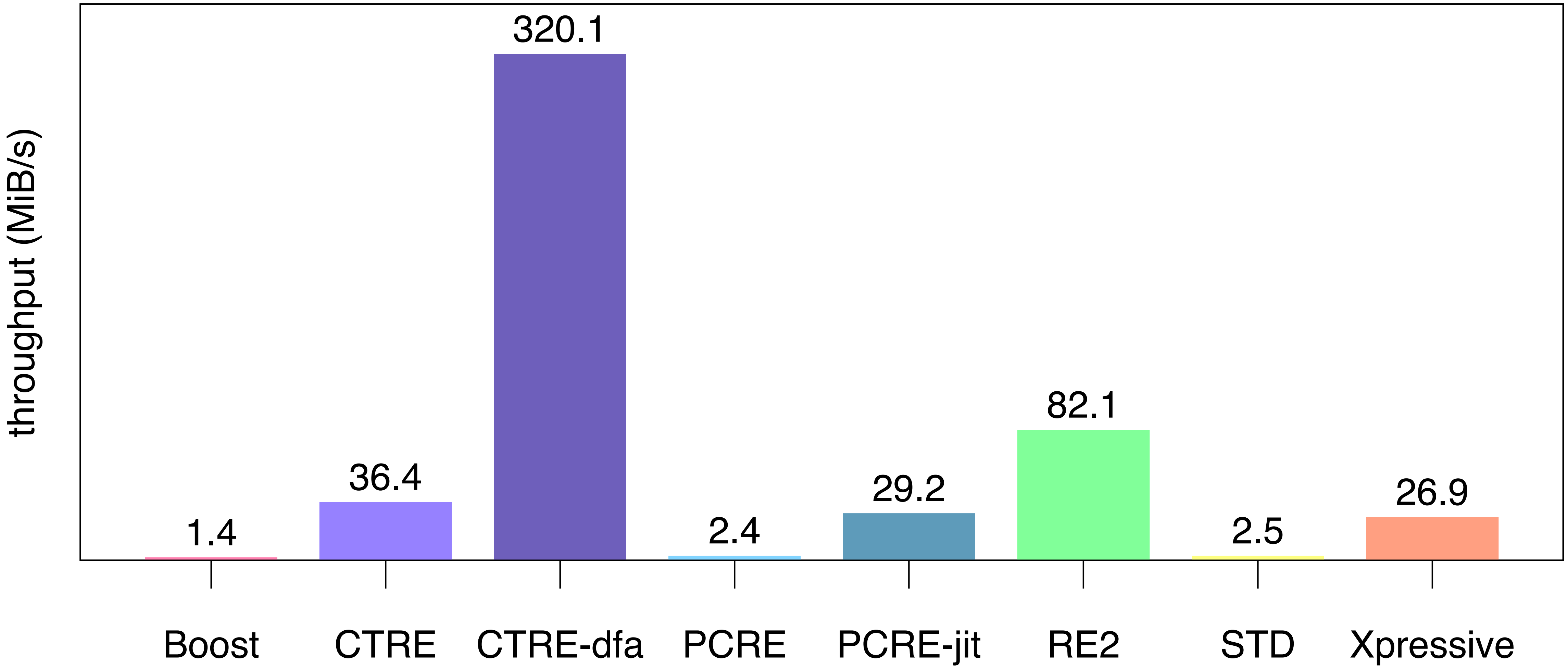
- CSV file (1.3GiB) with 6.5 MLoC
- Median of 3 measurements
- MacBook Pro 13" 2016 i7 3.3Ghz
- GCC 9.1 & clang 8 (-std=2a -O3)

RUNTIME SEARCHING (CLANG 8): `[a-z0-9]+abc[0-9]`



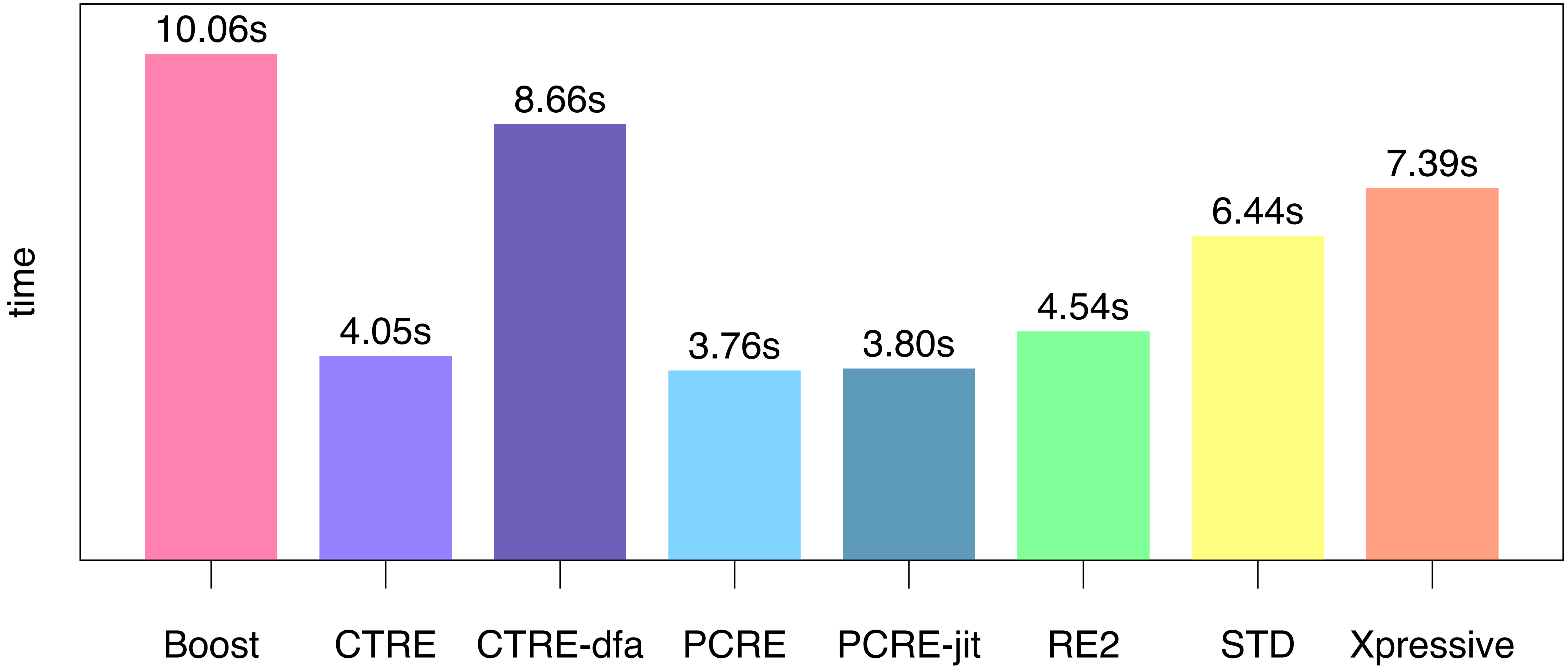
(higher is better)

RUNTIME SEARCHING (GCC 9): `[a-z0-9]+abc[0-9]`



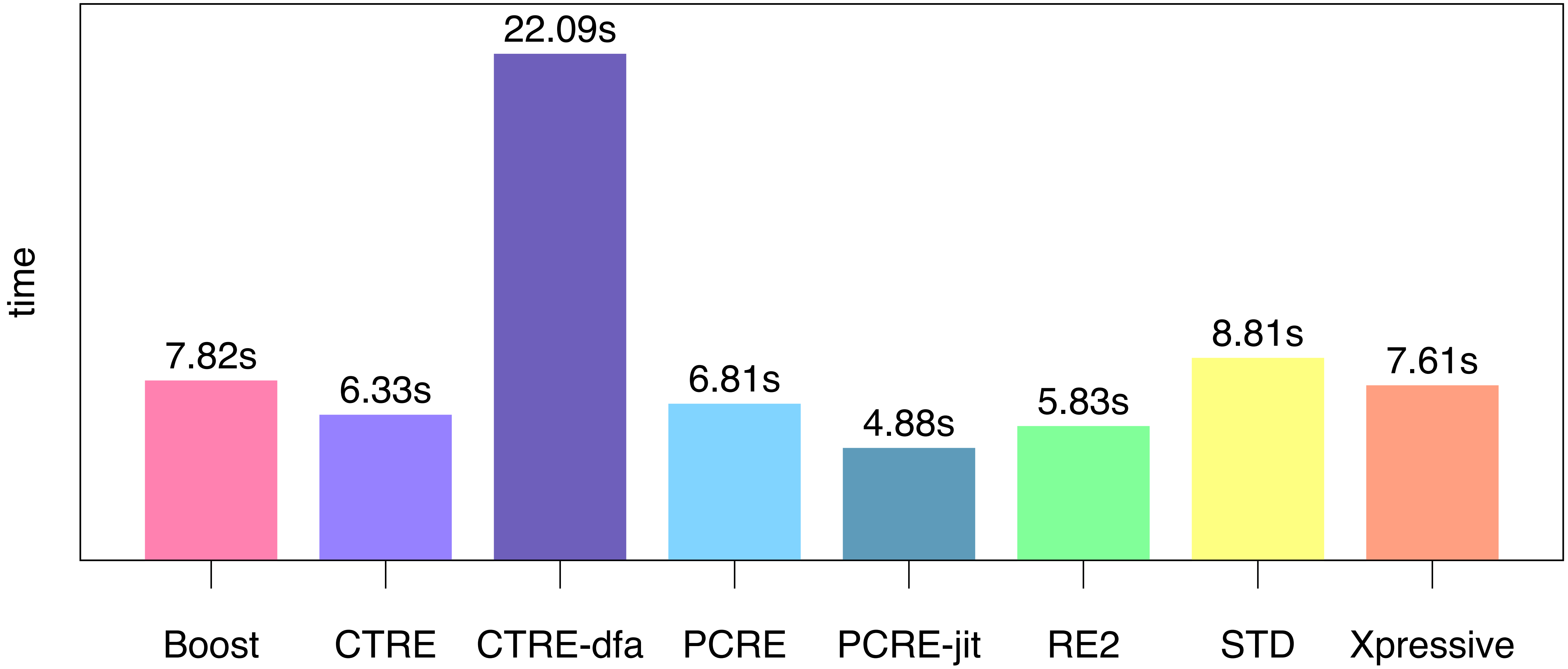
(higher is better)

COMPILATION (CLANG 8): `[a-z0-9]+?abc[0-9]`



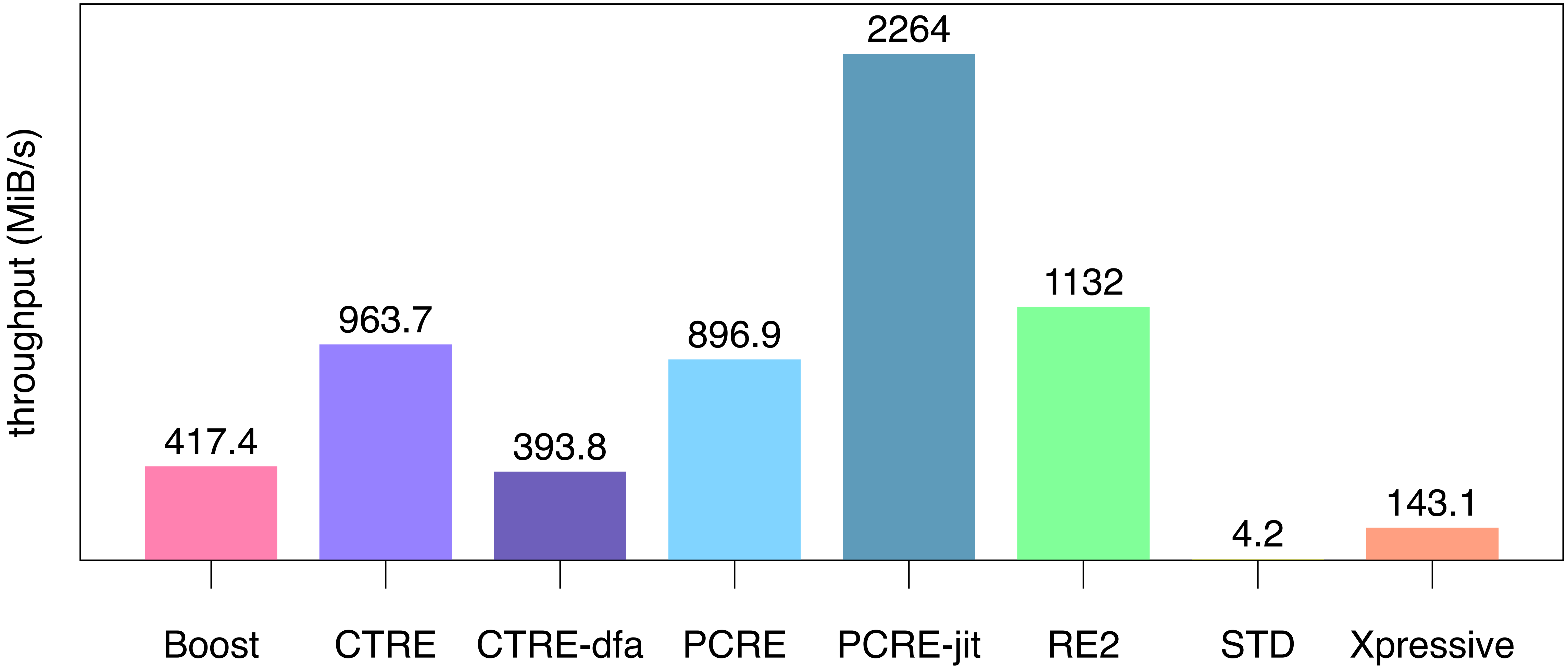
(lower is better)

COMPILATION (GCC 9): `[a-z0-9]+?abc[0-9]`



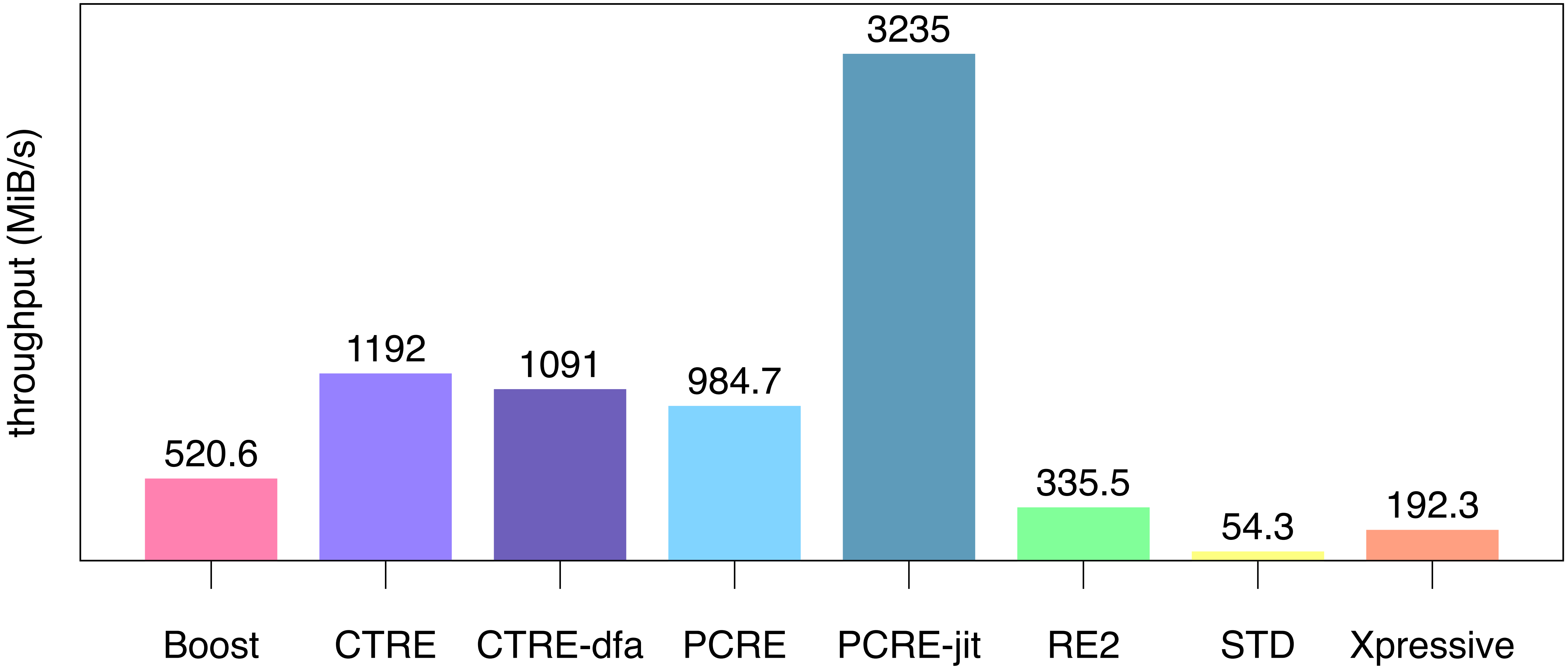
(lower is better)

RUNTIME SEARCHING (CLANG 8): `ABCDE-[0-9]+`



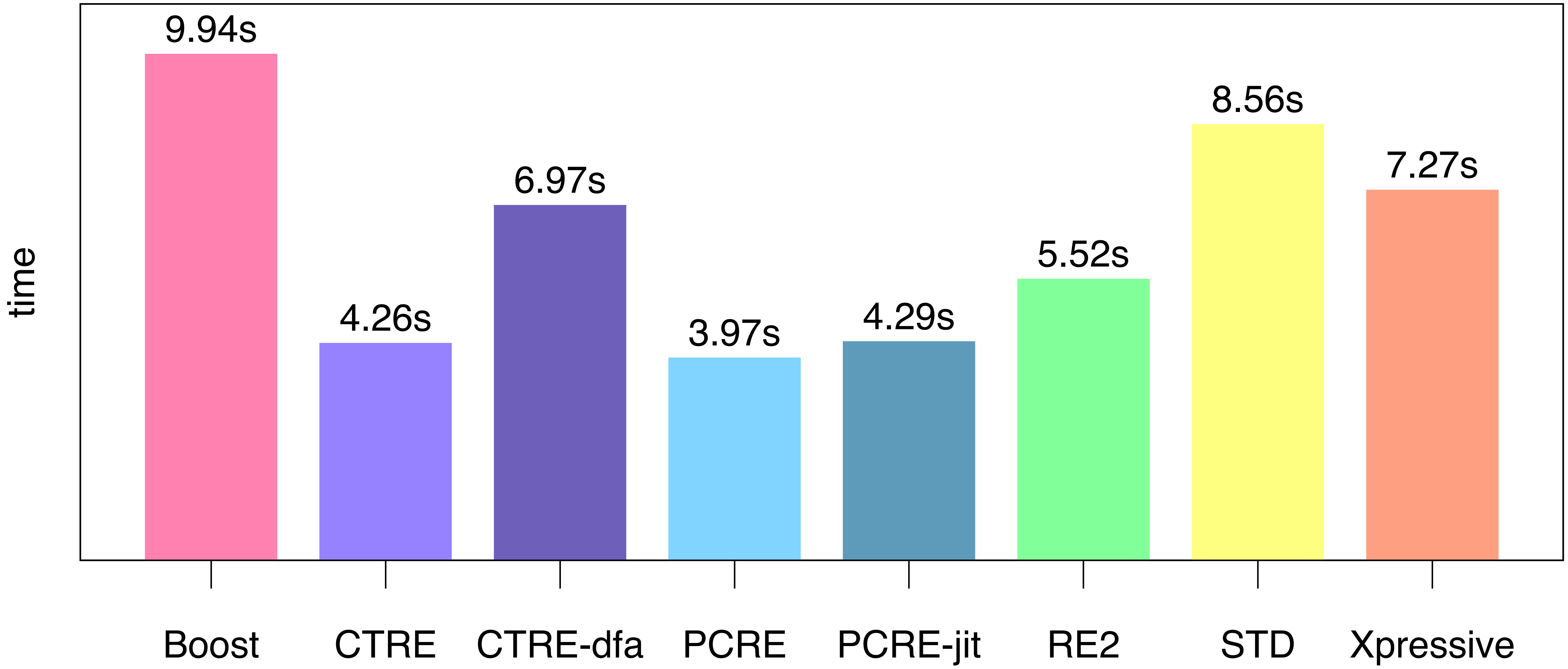
(higher is better)

RUNTIME SEARCHING (GCC 9): `ABCDE-[0-9]+`



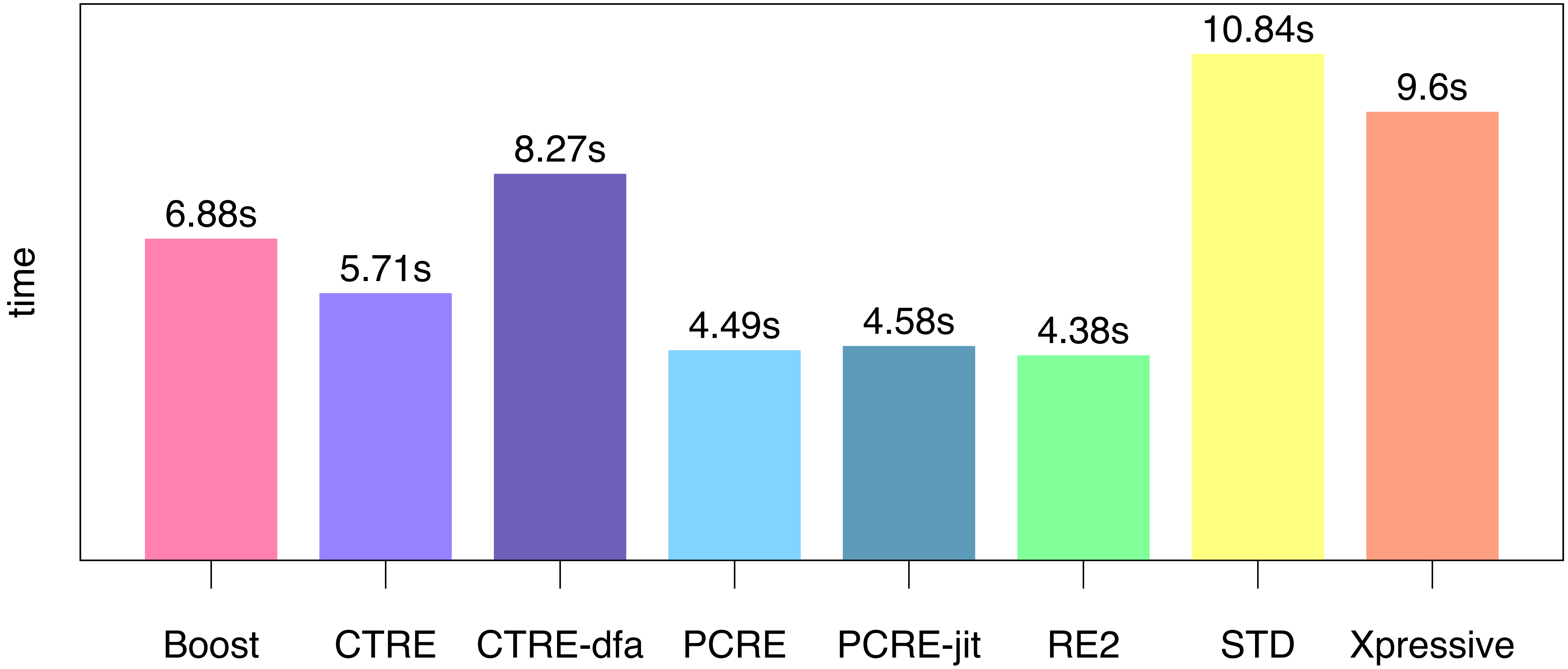
(higher is better)

COMPILATION (CLANG 8): `ABCDE-[0-9]+`



(lower is better)

COMPILATION (GCC 9): `ABCDE-[0-9]+`



(lower is better)

SIX THINGS I SHOWED YOU…

SIX THINGS I SHOWED YOU…

- How regular expression engines work.

SIX THINGS I SHOWED YOU…

- How regular expression engines work.
- Connection between theory and code.

SIX THINGS I SHOWED YOU...

- How regular expression engines work.
- Connection between theory and code.
- Parsing of a pattern during compilation.

SIX THINGS I SHOWED YOU…

- How regular expression engines work.
- Connection between theory and code.
- Parsing of a pattern during compilation.
- Building of a finite automaton.

SIX THINGS I SHOWED YOU...

- How regular expression engines work.
- Connection between theory and code.
- Parsing of a pattern during compilation.
- Building of a finite automaton.
- Matching a regex with various algorithms.

SIX THINGS I SHOWED YOU...

- How regular expression engines work.
- Connection between theory and code.
- Parsing of a pattern during compilation.
- Building of a finite automaton.
- Matching a regex with various algorithms.
- Comparison between different regex implementations.

AND FOUR THINGS YOU SHOULD REMEMBER...

AND FOUR THINGS YOU SHOULD REMEMBER...

- Compilers can optimize your code, if you help them.

AND FOUR THINGS YOU SHOULD REMEMBER...

- Compilers can optimize your code, if you help them.
- Combining TMP and constexpr code can
  help you express more things more precisely.

AND FOUR THINGS YOU SHOULD REMEMBER...

- Compilers can optimize your code, if you help them.
- Combining TMP and constexpr code can
  help you express more things more precisely.
- Stop pretending that constexpr C++ is a compiled language.

AND FOUR THINGS YOU SHOULD REMEMBER...

- Compilers can optimize your code, if you help them.
- Combining TMP and constexpr code can
  help you express more things more precisely.
- Stop pretending that constexpr C++ is a compiled language.
- Try the CTRE library

AND FOUR THINGS YOU SHOULD REMEMBER...

- Compilers can optimize your code, if you help them.
- Combining TMP and constexpr code can
  help you express more things more precisely.
- Stop pretending that constexpr C++ is a compiled language.
- Try the CTRE library, not just for speed but for code clarity.

THANK YOU!

You can find slides & code at `compile-time.re`