HokurikuRubyKaigi 01

計算機科学を Ruby と歩む

～ DFA 型正規表現エンジンをつくる～

@ydah

Saturday, December 6, 2025

Partner with RubyStackNews

Independent Ruby & Rails publication for senior developers

Why RubyStackNews?

- Focused on Ruby and Ruby on Rails
- Long-form articles based on real conference talks
- Audience of senior developers and tech leads
- Readers from the US, Europe, and Asia

RubyStackNews turns conference talks and real-world experience into practical, production-focused technical articles.

Partnerships & Sponsorships

- Article sponsorships
- Inline placements inside articles
- Sidebar visibility

View partnership details

rubystacknews.com

Partnership info

髙田 雄大

ID: @ydah

- プロダクトエンジニア @ 株式会社 SmartHR
- (CRuby | Lrama) コミッター
- Kyobashi.rb 創設メンバー
- (関西 | 大阪)Ruby 会議チーフオーガナイザー

Unsolicited Ads

関西 Ruby 会議 09

Otsu Traditional Performing Arts Center

2026-07-18(Sat)

RubyKansai, Kyoto.rb, KOBE.rb, AKASHI.rb, RubyMaizuru
Kyobashi.rb, Ruby Tuesday, Shinosaka.rb, naniwa.rb, Wakayama.rb

Unsolicited Ads

関ケ原 Ruby 会議 01

Sekigahara Community Center

2026-05-30(Sat)

@osyoyu @corocn @ydah
@pndcat @exSOUL @pastak @attsumi

正規表現

とは?

正規表現

正規表現とは

- 文字列の集合を一つの記法で表現するための数学的・計算論的概念
- 主にテキストデータの検索・置換・抽出や、入力値検査など、文字列パターンに合致するかを判別する用途で広く使われている
- 正規表現でないものを正規表現と呼んでいることがある
- 正規表現でないものを正規表現と呼んでいることがある

厳密な意味での正規表現を超えた拡張

- 有限オートマトンに対応する「厳密な意味での正規表現」より強い表現力を持つものはよく出会う
  - 後方参照： (\1), (\2)
  - 先読み・後読み： (?=...), (?!...), (?<=...), (?<!...)
  - 条件分岐： (?(1)yes|no)
  - 再帰・ネスト： (?R) , (?&name)

ラリー・ウォールもこう言ってる

"This is the Apocalypse on Pattern Matching, generally having to do with what we call “regular expressions”, which are **only marginally related to real regular expressions**. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here."

https://www.perl.com/pub/2002/06/04/apo5.html/

正規表現の厳密な定義

数学的に、「正則言語」と呼ばれる文字列の集合を、最小の要素から機能的に構築するための構造体

Primitives

- **空言語** ∅ : 要素を含まない言語
- **空文字列** **ε**: 長さ 0 の文字列のみ
- **単一文字** **a∈Σ**: 一文字のみ

Operators

- **和集合** **E|F**: E または F に含まれる集合
- **連結** **EF**: E と F を連結した集合
- **クリーネ閉包** **E***: E を 0 回以上連結

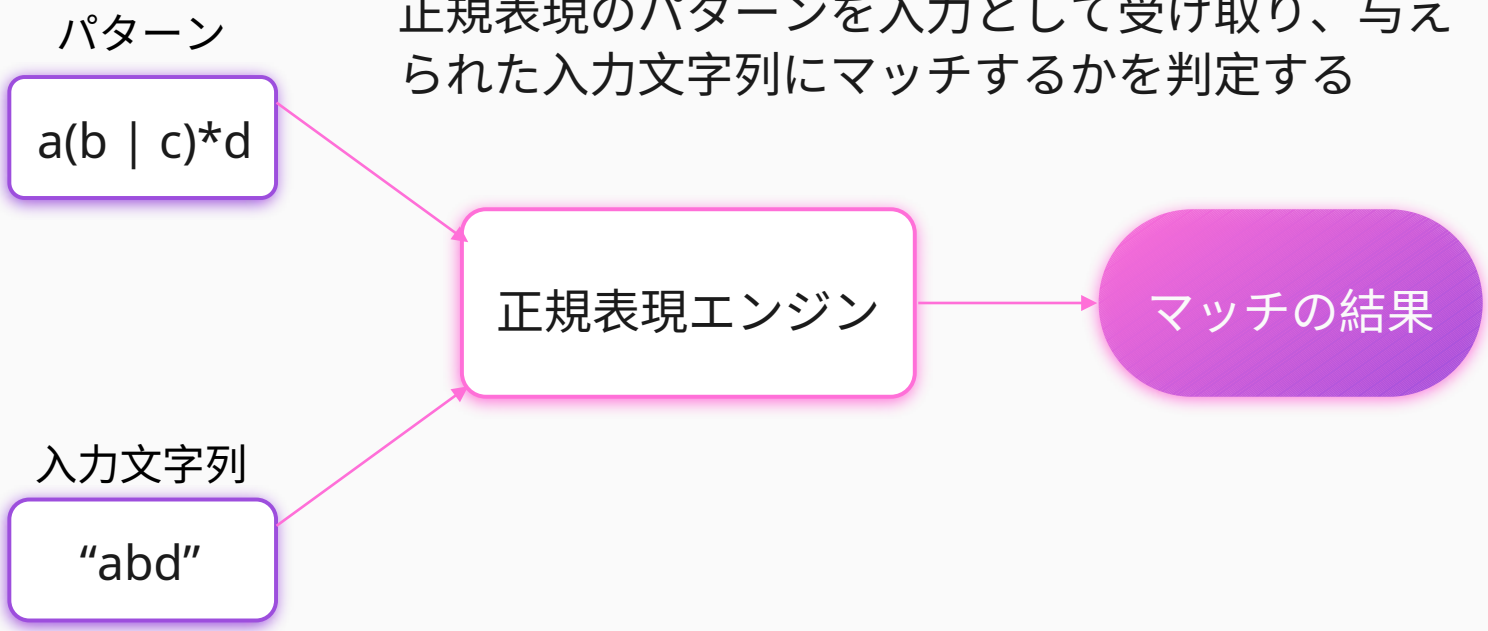
これらを有限回適用して得られる**すべての言語**が「正則言語」

正規表現エンジン

とは?

正規表現エンジンとは

正規表現のパターンを入力として受け取り、与えられた入力文字列にマッチするかを判定する



正規表現のマッチ方法のタイプ

正規表現エンジンは大きく 4 つのタイプが存在します

1. DFA 型

- **代表例 :** RE2, Hyperscan
- **概要 :** 正規表現を等価な DFA にコンパイルしてマッチ判定する

2. バックトラッキング NFA 型

- **代表例 :** PCRE, .NET, Python
- **概要 :** パターンを NFA 風の内部表現に変換し、バックトラックしながら探索する

3. VM 型（バイトコード実行）

- **代表例 :** RE2C
- **概要 :** 独自のバイトコードにコンパイルし、小さな VM 上で命令列として実行する

4. 正規表現微分（ Brzozowski 微分）

- **代表例 :** 理論研究や実験的エンジン
- **概要 :** 正規表現R と文字a に対して、微分 Da(R) を定義し、入力文字ごとに更新する

正規表現のマッチ方法のタイプ

この方式について話します

正規表現エンジンは大きく 4 つのタイプが存在します

1. DFA 型

- **代表例：** RE2, Hyperscan
- **概要：** 正規表現を等価な DFA にコンパイルしてマッチ判定する

2. バックトラッキング NFA 型

- **代表例：** PCRE, .NET, Python
- **概要：** パターンを NFA 風の内部表現に変換し、バックトラックしながら探索する

3. VM 型（バイトコード実行）

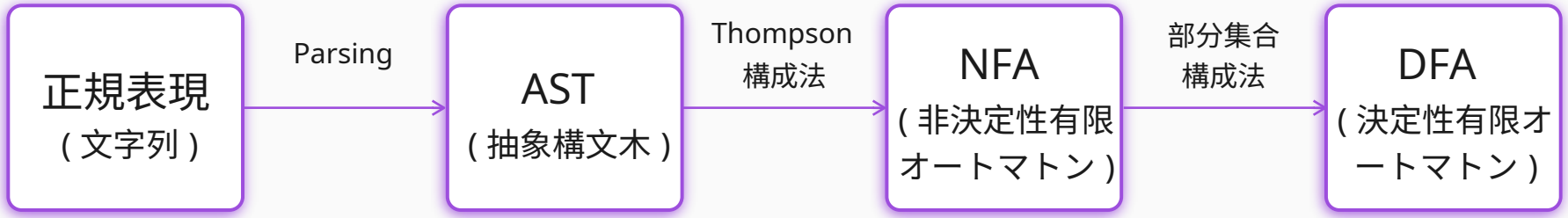
- **代表例：** RE2C
- **概要：** 独自のバイトコードにコンパイルし、小さな VM 上で命令列として実行する

4. 正規表現微分（ Brzozowski 微分）

- **代表例：** 理論研究や実験的エンジン
- **概要：** 正規表現R と文字a に対して、微分 Da(R) を定義し、入力文字ごとに更新する

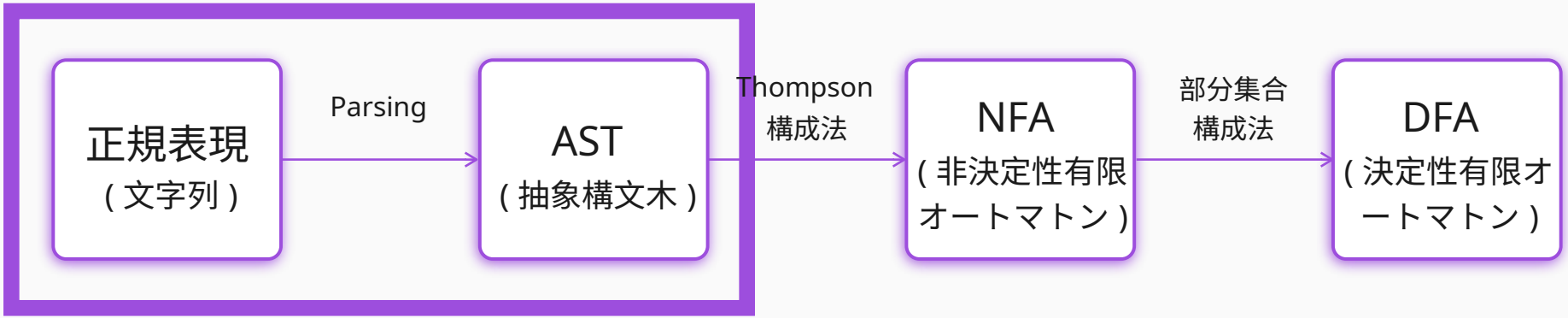
正規表現を DFA に変換する道のり

正規表現エンジンは、パターン文字列を解析し、最終的に高速なマッチングマシンに変換するパイプラインです。このプロセスは、一般的に 3 つの主要なステップで構成されます。



パターン文字列を構造化する

正規表現から線形時間マッチングが可能な DFA への道筋の最初のステップは、パターン文字列を「構造化」することです。



なぜ、ただの文字列ではダメ?

パターン **a | b *** はどう解釈すべきでしょう?

(a | b)*

[a または b] の 0 回以上の繰り返し

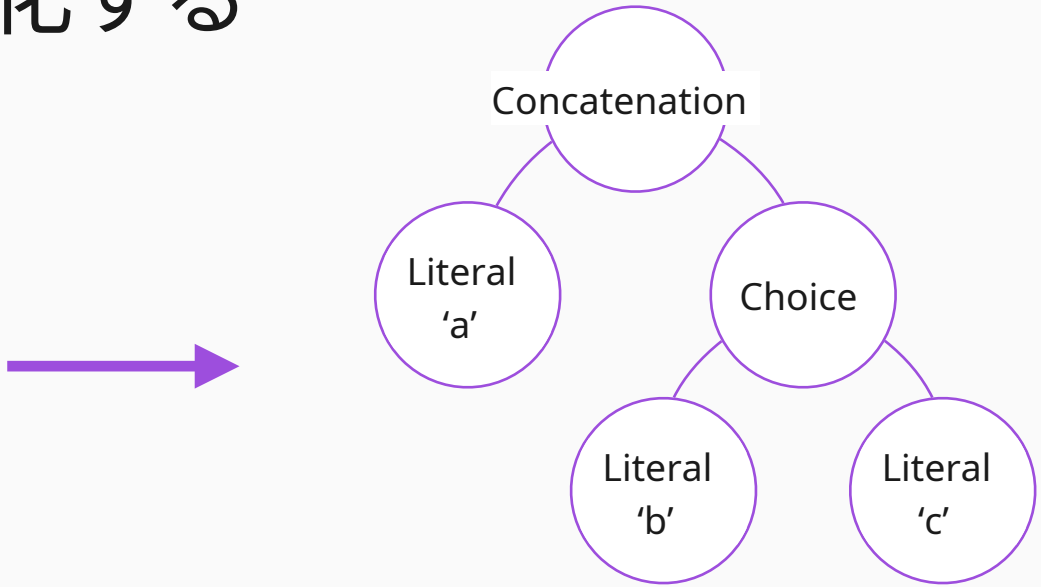
a | (b*)

[a] または [b の 0 回以上の繰り返し]

文字列のままでは、連結や選択といった演算子の「優先順位」や「適用範囲」が曖昧です。この曖昧さを解決するために、構造的な表現が必要になります。

パターンを構造化する

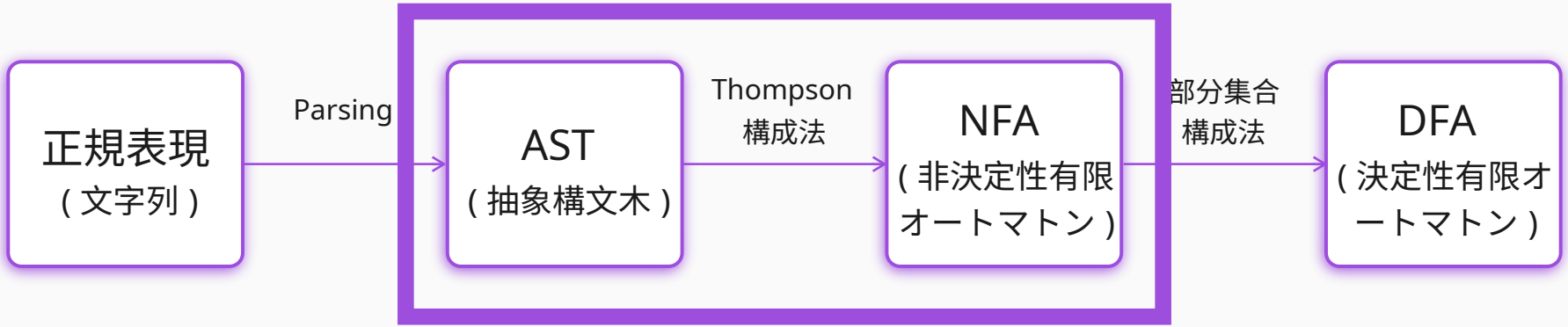
a(b | c)



パターン文字列を抽象構文木（ Abstract Syntax Tree, AST ）と呼ばれる木構造に変換します。この操作は構文解析器が担います。

AST から中間表現としての NFA へ

AST から DFA へ直接変換するのは複雑です。そこで、中間表現として NFA を利用します。 NFA は、 AST の構造から比較的簡単に構築できます。



計算モデルとしての有限オートマトン

有限個の状態を持つ計算モデルです。入力文字列を一文字ずつ読み取り、状態を遷移させる。最終的に受理状態であれば、「受理」されます。

決定性有限オートマトン

- **定義** ： 任意の状態と入力文字に対し、次の状態が一意に定まる
- **遷移関数** ：$\delta = Q \times \Sigma \to Q$

非決定性有限オートマトン

- **定義** ： 任意の状態と入力文字に対し、次の状態が複数存在しうる、 **ε による遷移も許容する**
- **遷移関数** ：$\delta = Q \times (\Sigma \cup \{\varepsilon\} \to 2^{Q}$

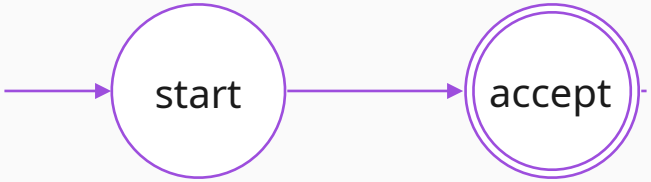
どちらも形式的に $M = (Q, \Sigma, \delta, q_0, F)$ で定義されるが、遷移関数 δ が異なる

Thompson 構成法

正規表現から非決定性有限オートマトンを構築する基本的なアルゴリズムです。文字、連結、選択、繰り返しに対して、それぞれ対応する NFA の「部品」を定義し、再帰的に組み合わせて NFA を構築します。

すべての部品は、単一の開始状態と単一の受理状態を持ちます

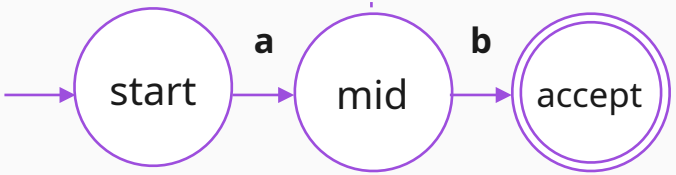
リテラル `a`




```
def to_nfa(state)
  start = state.new_state
  accept = state.new_state
  nfa = Automaton::NFA.new(start, [accept])
  nfa.add_transition(start, @value, accept)
  nfa
end
```

リテラルはとてもシンプルな変換になります。

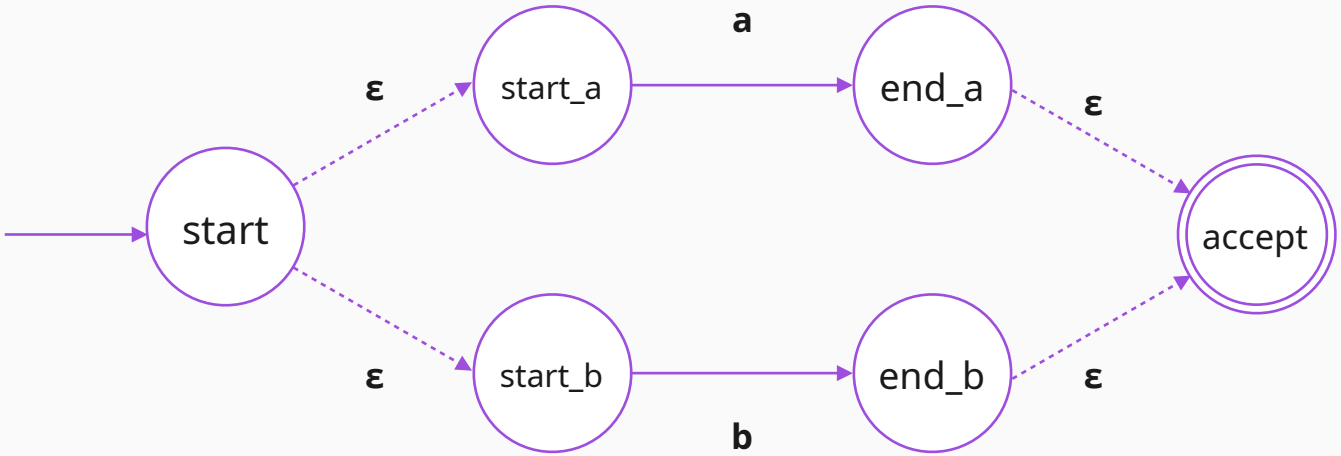
連結 `ab`




```
def to_nfa(state)
  nfas = @children.map { |child| child.to_nfa(state) }
  nfa = nfas.first
  nfas.drop(1).each do |next_nfa|
    nfa.merge_transitions(next_nfa)
    nfa.accept.each do |accept|
      nfa.add_epsilon_transition(accept, next_nfa.start)
    end
    nfa.accept = next_nfa.accept
  end
  nfa
end
```

`a` と `b` の NFA を直列に繋ぐ。 `a` の受理状態が、 `b` の開始状態になります。

選択 `a|b`



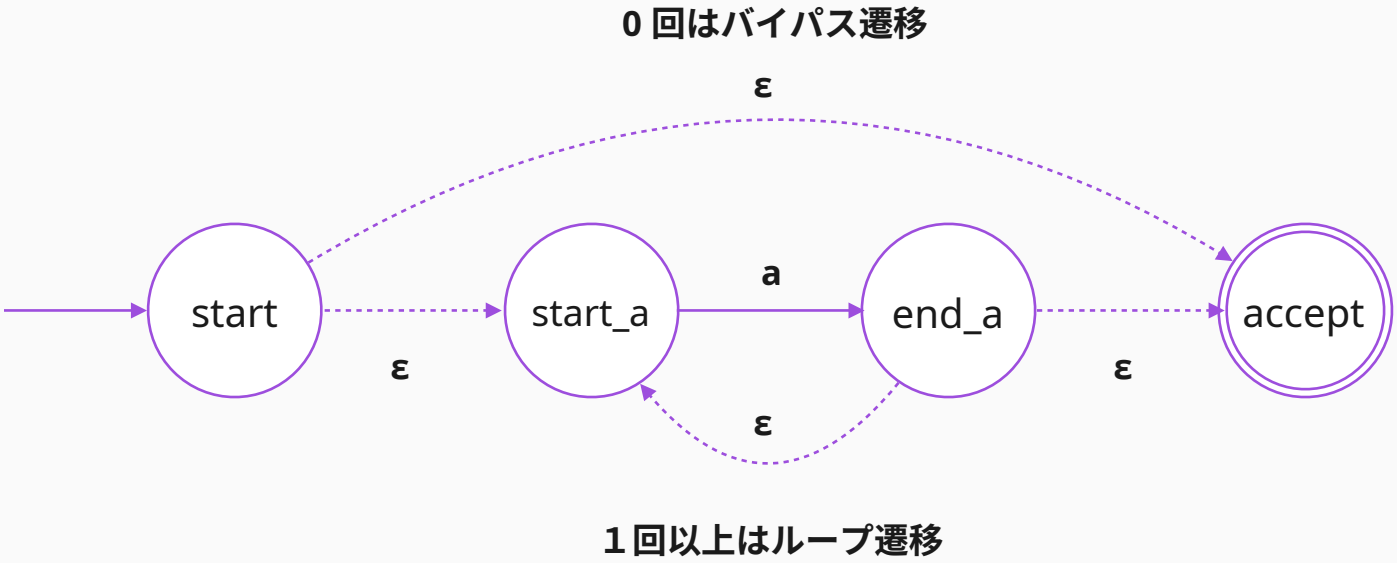
`a` と `b` の NFA を並列に配置し、新しい開始状態と受理状態を ε 遷移で繋ぐ。

選択 `a|b`

```
def to_nfa(state)
  child_nfas = @children.map { |child| child.to_nfa(state) }
  start_state = state.new_state
  accepts = child_nfas.flat_map(&:accept).to_set
  nfa = Automaton::NFA.new(start_state, accepts)
  child_nfas.each do |child_nfa|
    nfa.merge_transitions(child_nfa)
    nfa.add_epsilon_transition(start_state, child_nfa.start)
  end
  nfa
end
```

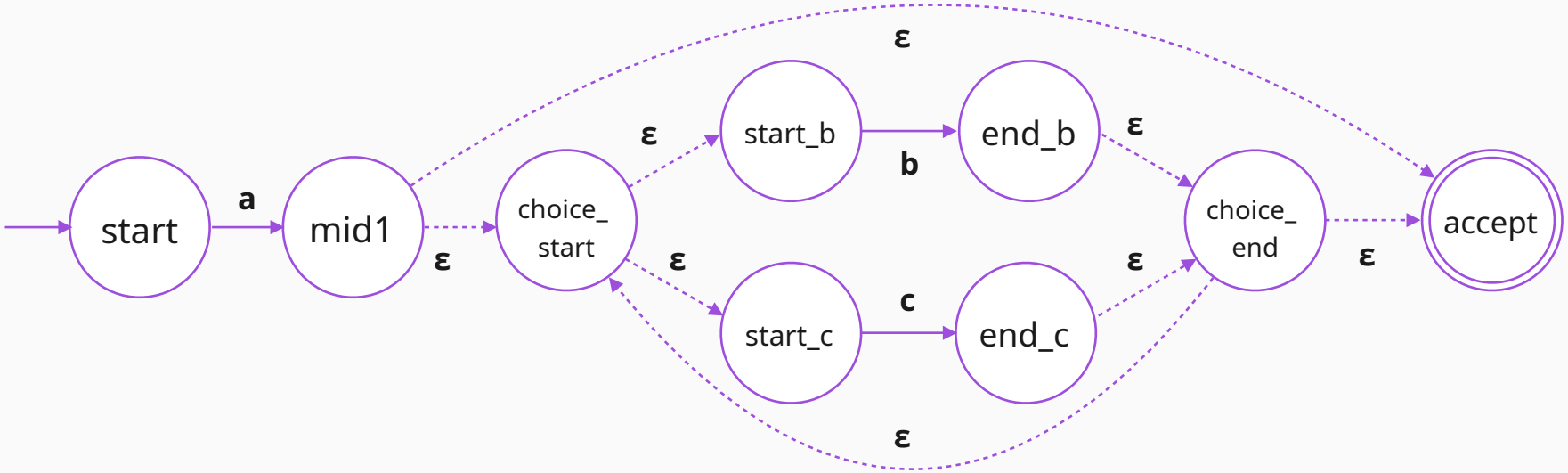
子を NFA に変換して、新しい開始状態を作成し、各 NFA の先頭へ ε 遷移をつなぎ分岐構造を作る。

繰り返し `a*`



`a` の NFA を ε 遷移でループさせ、全体をバイパスする ε 遷移を追加します。

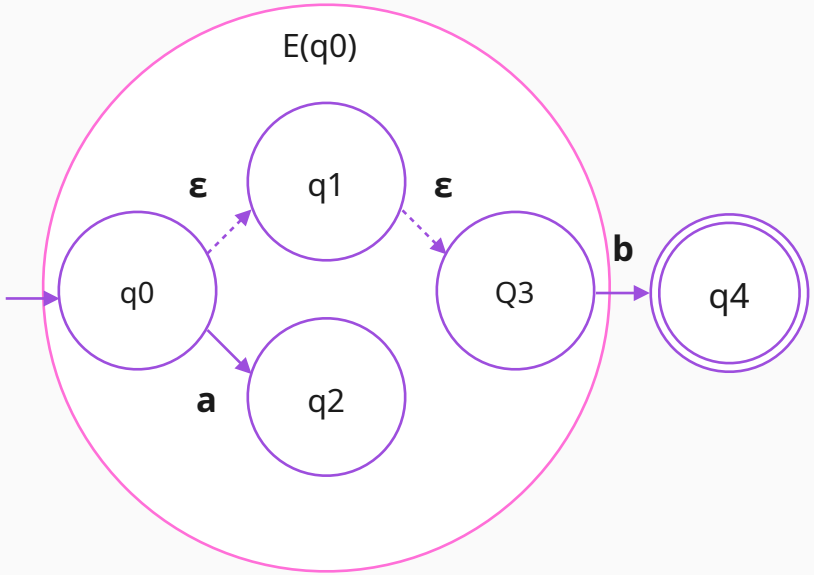
組み合わせる：`a(b|c)*`



部品を再帰的に組み合わせると、複雑な正規表現でも NFA に変換できます。

ε- 遷移と ε- 閉包

入力文字列を 1 文字も消費せずに状態が自由に移れる遷移を ε- 遷移といいます。そして、ある状態集合 から、 ε 遷移だけを 0 回以上繰り返して到達可能な全状態の集合を ε- 閉包といいます。



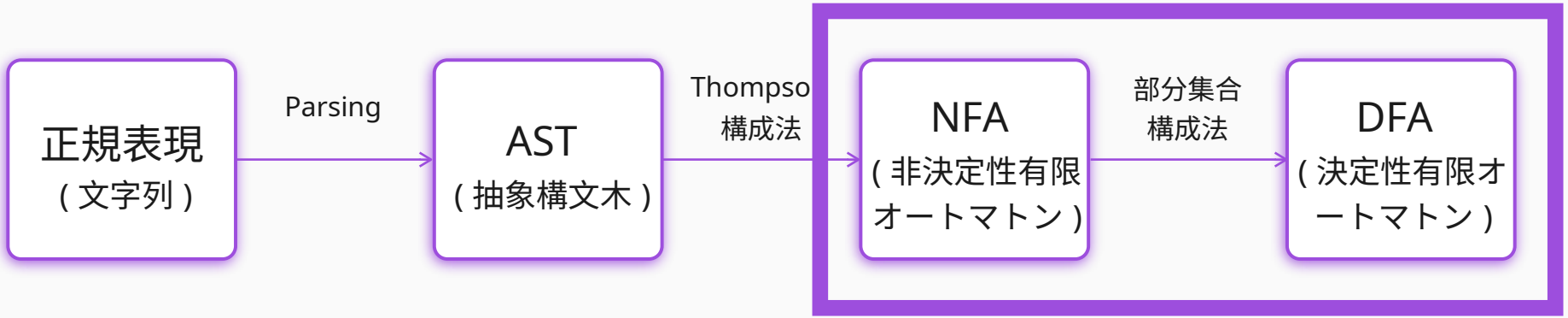
ε- 閉包の実装（幅優先探索）

```ruby
def epsilon_closure(start)
  visited = start.dup
  queue = start.to_a
  while (current = queue.shift)
    destinations = @transitions.select do |from, label, _|
      from == current && label.nil?
    end.map(&:last)
    destinations.each do |dest|
      queue << dest if visited.add?(dest)
    end
  end
  :::SortedSet.new(visited)
end
```

キューから取り出す、εで行ける隣を探す、未訪問ならキューに入れるサイクルを、キューが空になるまで繰り返す

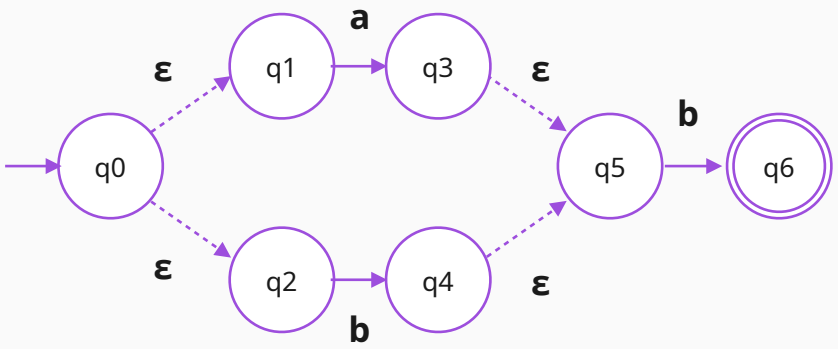
NFA から DFA へ

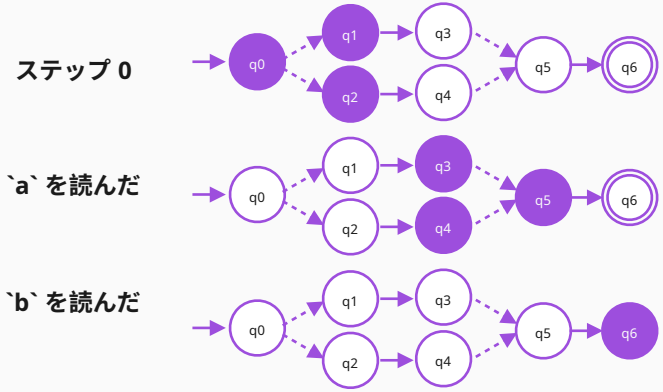
NFA は構築が簡単でしたが、マッチングにはまだ非決定性が残っています。そこで高速なマッチングが可能な DFA に変換します。



NFA のジレンマ：非決定性のコスト

マッチング時、 NFA は「現在ありうる全ての状態」を同時に追跡し続ける必要がある。





- 入力文字を読むたびに、遷移可能なすべての状態を計算し、その集合を保持する必要があります
- この「状態集合の管理」が計算コストを増大させます
- 作りやすさの代償として、実行速度が犠牲になる可能性があります

高速な実行エンジン： DFA

DFA の主要な特徴

- 任意の状態と入力文字に対して、遷移先は常にただ 1 つに決まる
- ε- 遷移は存在しない
- 遷移先は常に一意で曖昧さがない

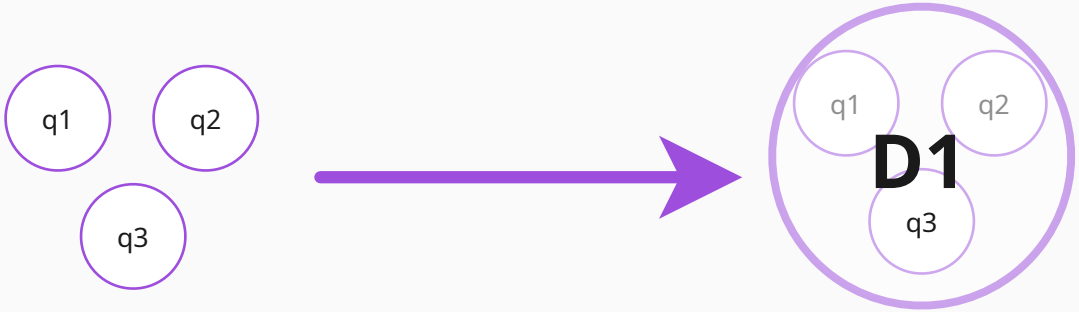
マッチングアルゴリズム

- シンプルで高速な処理で実現可能

```
# DFAでのマッチング（擬似コード）
current = dfa.start_state
input.each_char do |char|
  current = dfa.transition(current, char)
end
dfa.end_states.include?(current)
```

部分集合構成法

NFA を DFA に変換する標準的なアルゴリズムです。 NFA の状態の集合を、 DFA の 1 つの状態とみなします。 DFA の各状態は、 NFA が「今、同時に存在しうるすべての状態」を表します。

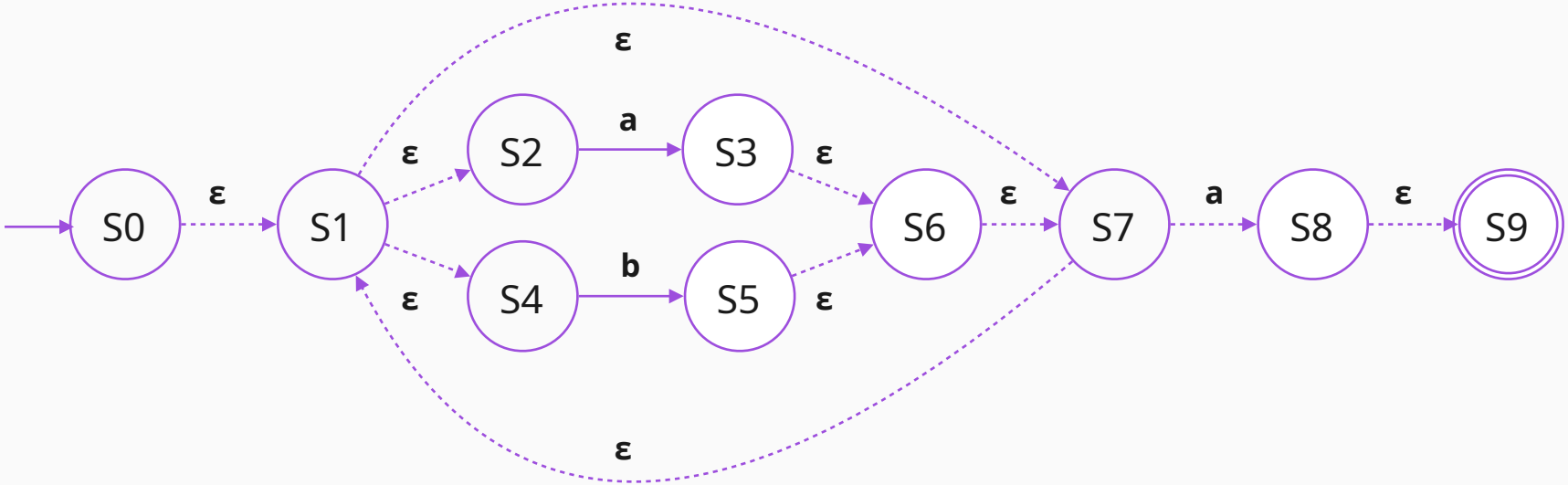


1 つの DFA 状態 = NFA 状態の 「集合」

例：正規表現 `( a|b )*a` に対応する NFA

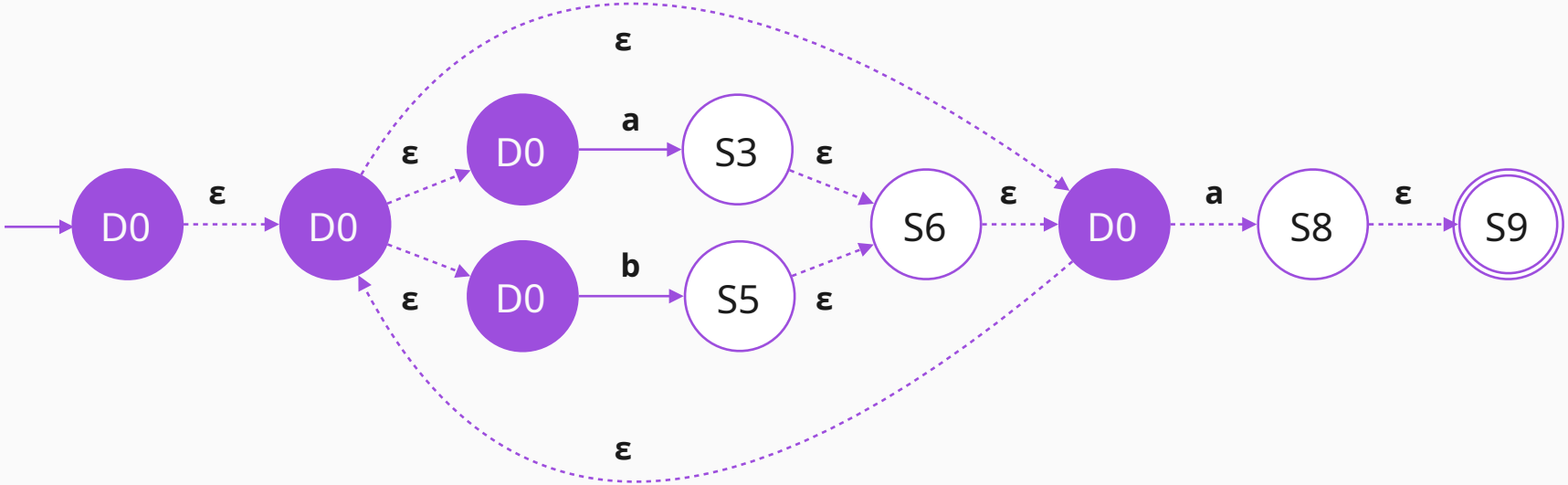
NFA を DFA に変換するプロセスをステップごとに追っていきます。

NFA は正規表現 `( a|b )*a` から生成されたものです。



ステップ 1： DFA の開始状態を決定する

DFA の構築は、 NFA の開始状態の ε- 閉包を求めることから始まります。これが DFA の最初の状態 `D0` となります。



ステップ 1： DFA の開始状態を決定する

開始状態の ε-closure を計算

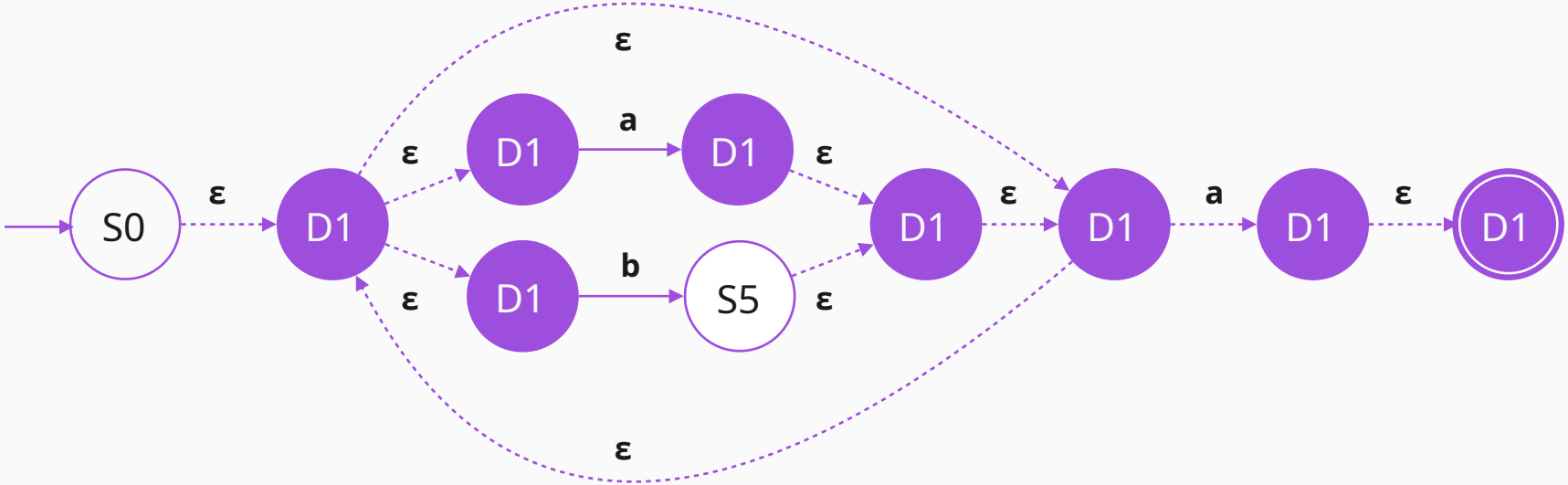
NFA 状態集合→ DFA 状態 ID のマッピング

```
def initialize_dfa
  start = @nfa.epsilon_closure(Set.new([@nfa.start]))
  start_id = 0
  @dfa_states[start] = start_id
  @queue <<< start
  @dfa = DFA.new(start_id, Set.new)
end
```

ステップ 2： 状態 D0 からの遷移を計算

D0 の各 NFA 状態から、入力 `a` で遷移できる状態の集合を求める

その集合に対して、さらに ε- 閉包を計算する



ステップ 2： 状態 D0 からの遷移を計算

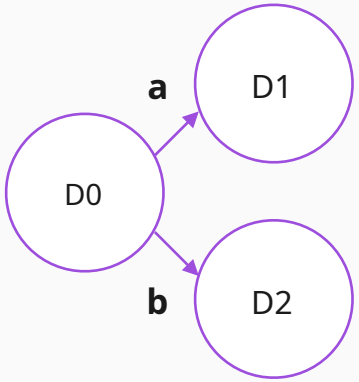
```ruby
def build_transitions(nfa_states)
  transitions = Hash.new { |h, k| h[k] = Set.new }
  nfa_states.each do |state|
    @nfa.transitions.each do |from, label, to|
      next unless from == state && !label.nil?

      transitions[label].merge(@nfa.epsilon_closure(Set[to]))
    end
  end
  transitions
end
```

D0 の各 NFA 状態を処理、 NFA の全遷移をチェック、 D0 からの遷移を計算する

ステップ 3： 新たな DFA 状態の発見と登録

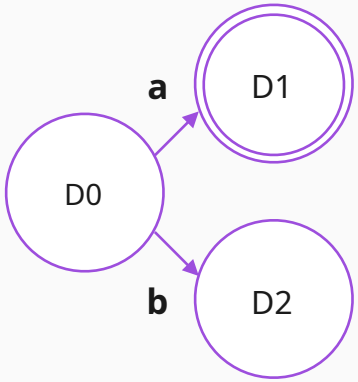
ステップ 2 で計算した遷移先の集合を、新しい DFA 状態とする




```ruby
def ensure_state(nfa_states)
  if @dfa_states.key?(nfa_states)
    return @dfa_states[nfa_states]
  end
  new_id = @dfa_states.length
  @dfa_states[nfa_states] = new_id
  @queue.push(nfa_states)
  new_id
end
```

ステップ４： 受理状態を見つける

DFA の持つ状態が内包する NFA の状態の集合に、受理状態が含まれていれば、 DFA における受理状態とする。

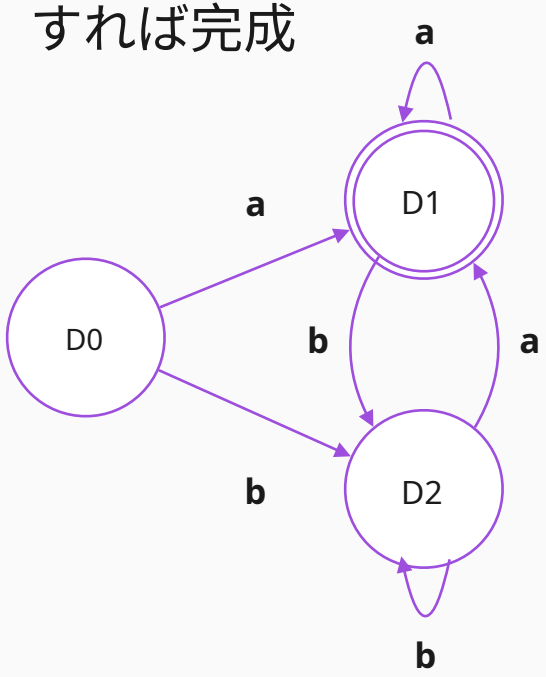



```
def mark_accept(states, id)
  return unless states.any?
    { |state| @nfa_accepts.include?(state) }

  @dfa.accept.merge([id])
end
```

ステップ 5： キューが空になるまで繰り返す

キューに追加された状態 D1 と D2 を順に取り出して、同様に遷移を計算すれば完成




```
def process_states
  while (nfa_states = @queue.shift)
    current_id = @dfa_states[nfa_states]
    mark_accept(nfa_states, current_id)
    transitions= build_transitions(nfa_states)
    process_transitions(transitions, current_id)
  end
end
```

DFA ができればマッチングの処理

```
def match?(input)
  state = @start
  input.each_char do |char|
    state = @transitions.find { |from, label, to|
      from == state && label == char
    }&.last
    return false unless state
  end
  @accept.include?(state)
end
```

現在の状態と入力文字から次の状態を探す、遷移先がなければ拒否、最終状態が受理状態なら受理、受理でないと拒否

完成！！！1

何故、 Ruby が学習に最適か

実装して理解したい、**アルゴリズムそのものに集中ができる**

Ruby での実装

“正規表現アルゴリズムそのものに集中出来る”

- **手に馴染んでいる（おだいじ）**
- **強力な組み込みデータ構造（ Set 、 Hash ）**
- **自動メモリ管理**
- **表現力豊かな構文**

他の言語（例： C 言語）

“アルゴリズムに加え、低レベルなリソース管理も必須”

- **手動でのメモリ確保・解放**
- **ポインタとアドレスの管理**
- **データ構造の自作**
- **より多くのコード行数と認知負荷**

“秋から冬にかけては正規表現エンジンの季節”

※諸説あり

作りたくなりましたよね?

迂闊に作っていきましょう

実装の参考

このトークで紹介した正規表現エンジン「鬼灯（ Hoozuki ）」の全コードは GitHub で公開されています。正規表現エンジンを作る際の参考にしてください。

https://github.com/ydah/hoozuki

Thank You!