アンダースタンディング・コンピュテーション　第 3 章
有限オートマトン

岡本 吉央
okamotoy@uec.ac.jp

電気通信大学

2019 年 6 月 14 日

最終更新：2019 年 6 月 14 日　11:58

スケジュール

2 プログラム意味論 (5 月)
3 有限オートマトン (6 月)
4 プッシュダウン・オートマトン
5 チューリング機械
6 ラムダ計算
7 万能性
8 決定可能性
9 抽象解釈／静的意味論

目次

今からやること

コンピュータの原理を学ぶために
「簡単な機能しか持たないコンピュータ」を (Ruby で) 作ってみる

- 有限オートマトン

注意：

コンピュータ (計算機)	物理的 (実体)
有限オートマトン	数学的 (仮想)

「有限オートマトン」は「コンピュータ」の簡単な部分を
理想化・単純化したもの

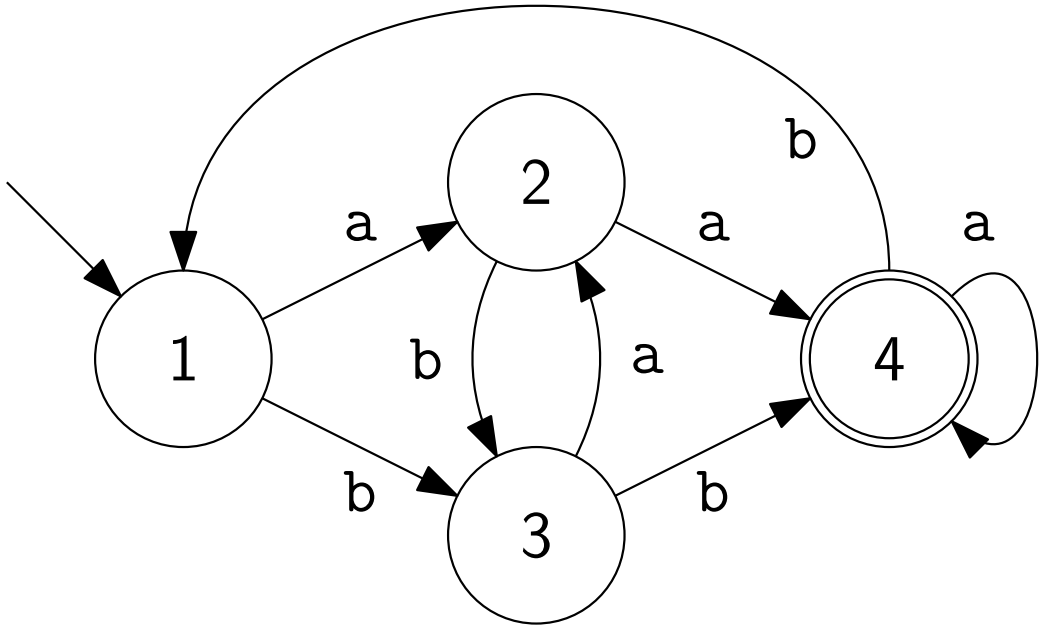
この章で登場する重要概念

- 決定性／非決定性
- 計算能力の等価性

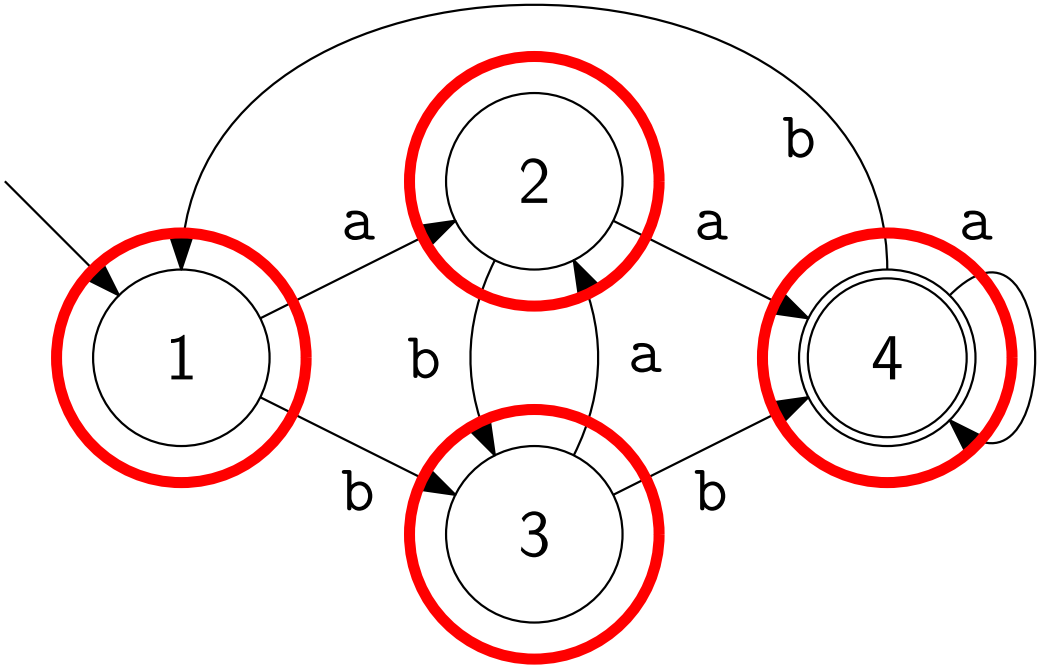
これらはこの章だけではなく，後でも重要になる

目次

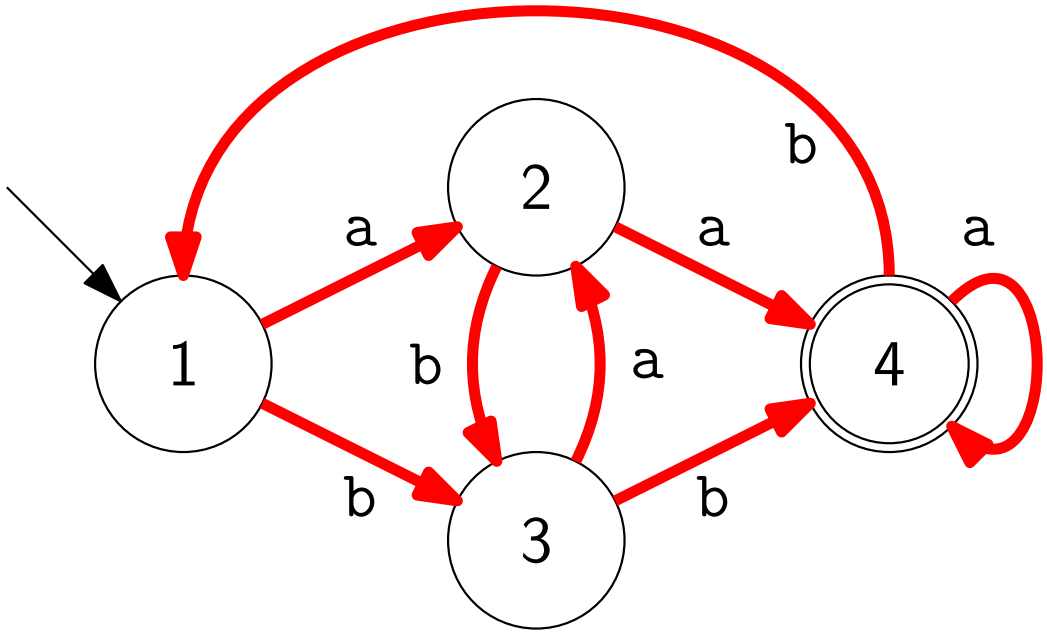
決定性有限オートマトンの図示



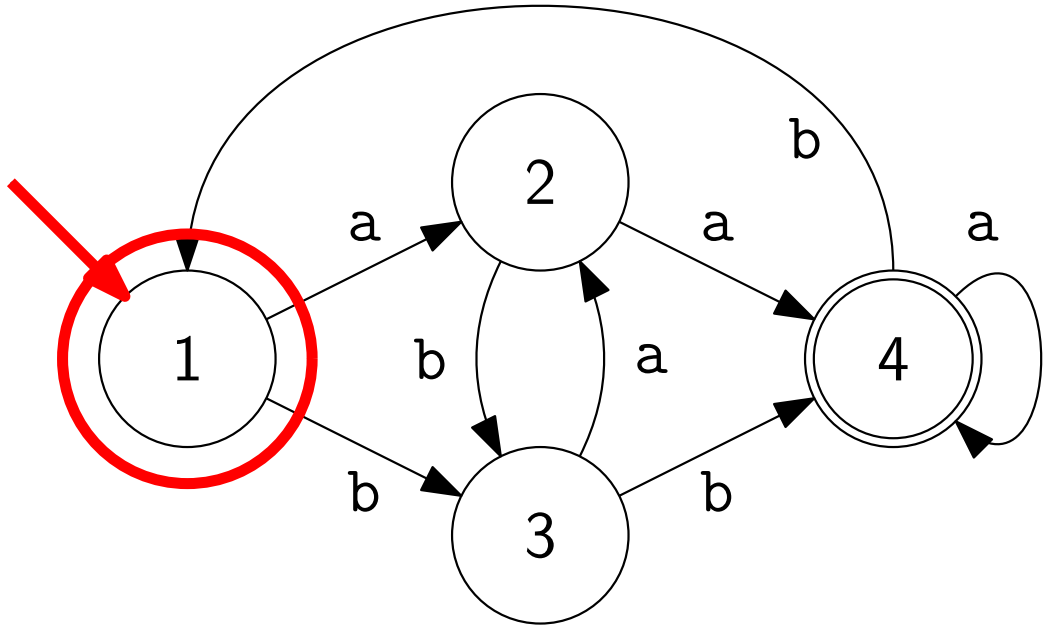
決定性有限オートマトンの構成要素：状態



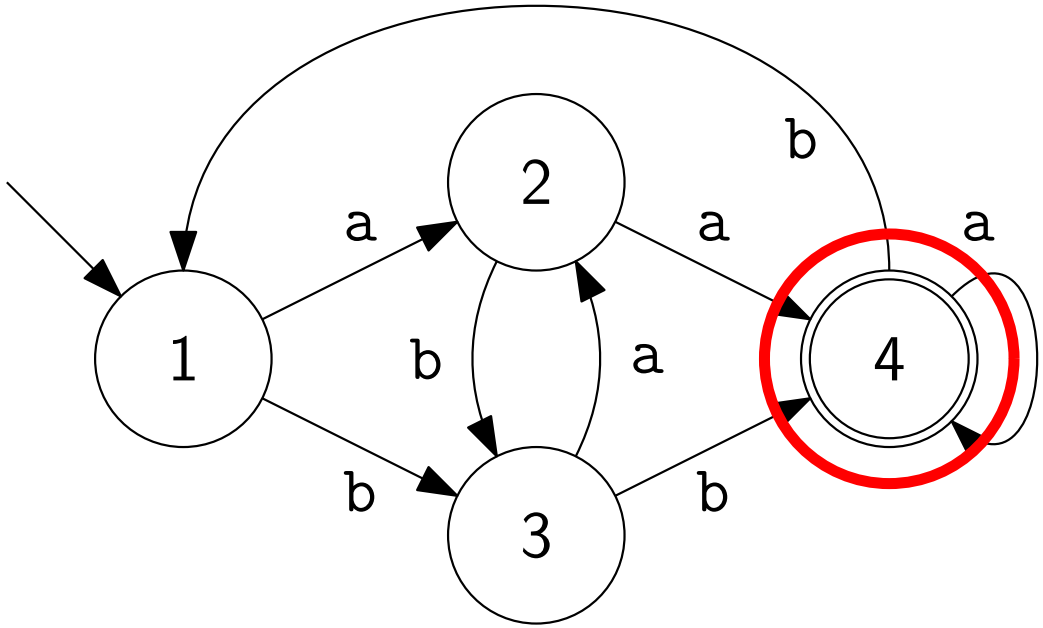
決定性有限オートマトンの構成要素：規則



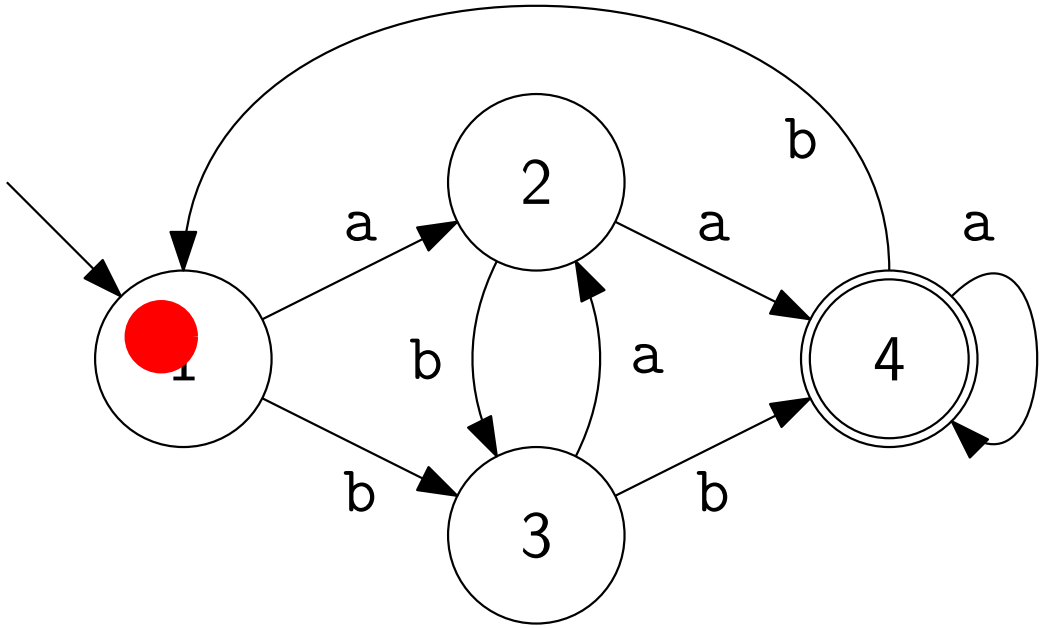
決定性有限オートマトンの構成要素：開始状態



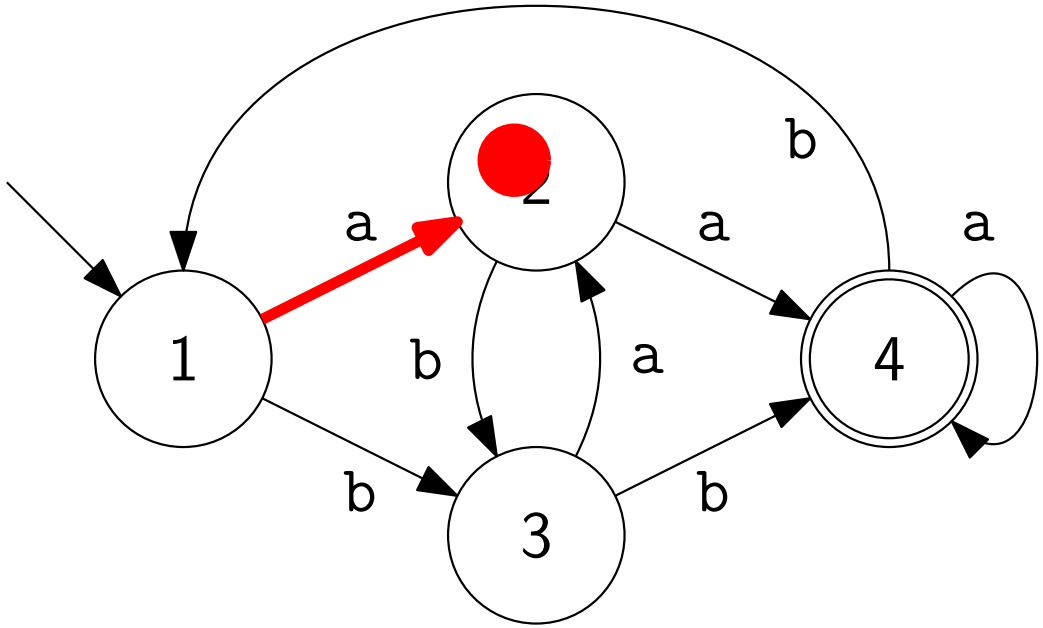
決定性有限オートマトンの構成要素：受理状態



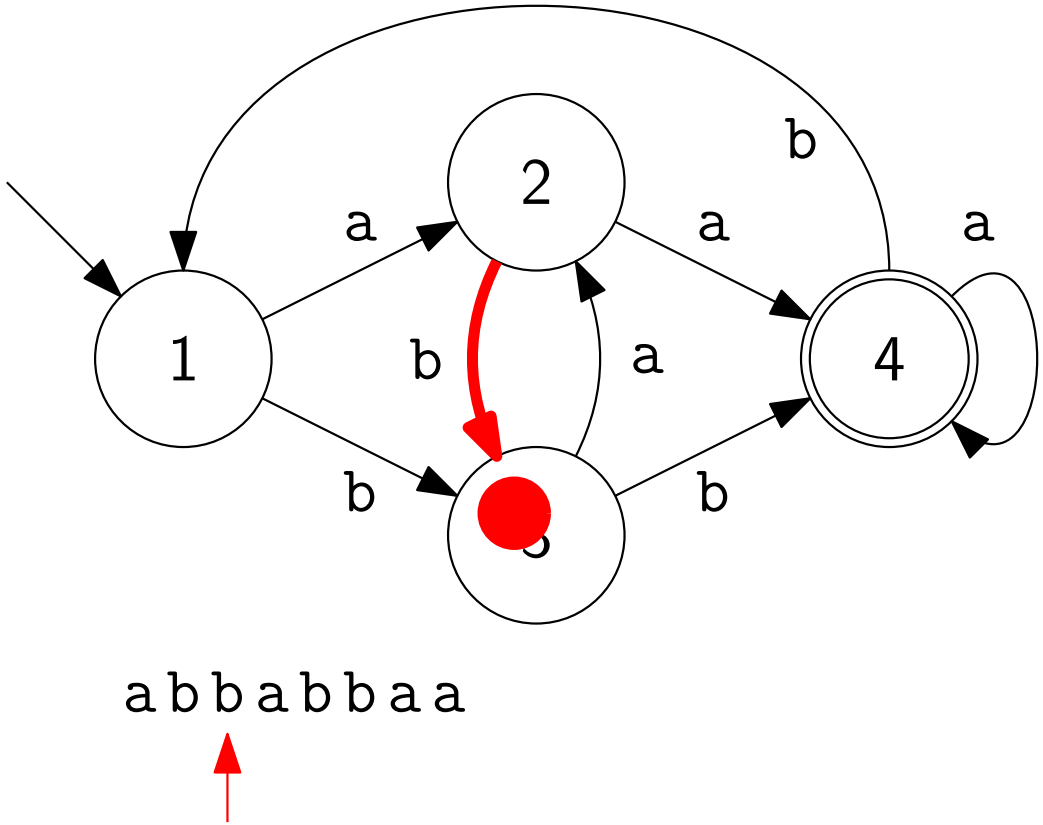
決定性有限オートマトン：文字列の受理



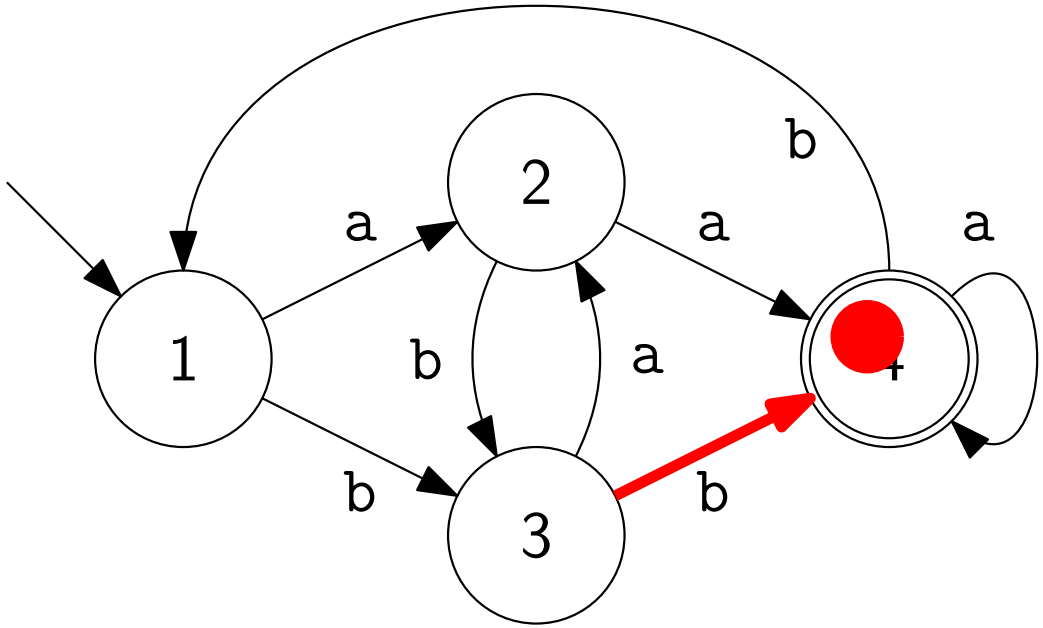
決定性有限オートマトン：文字列の受理



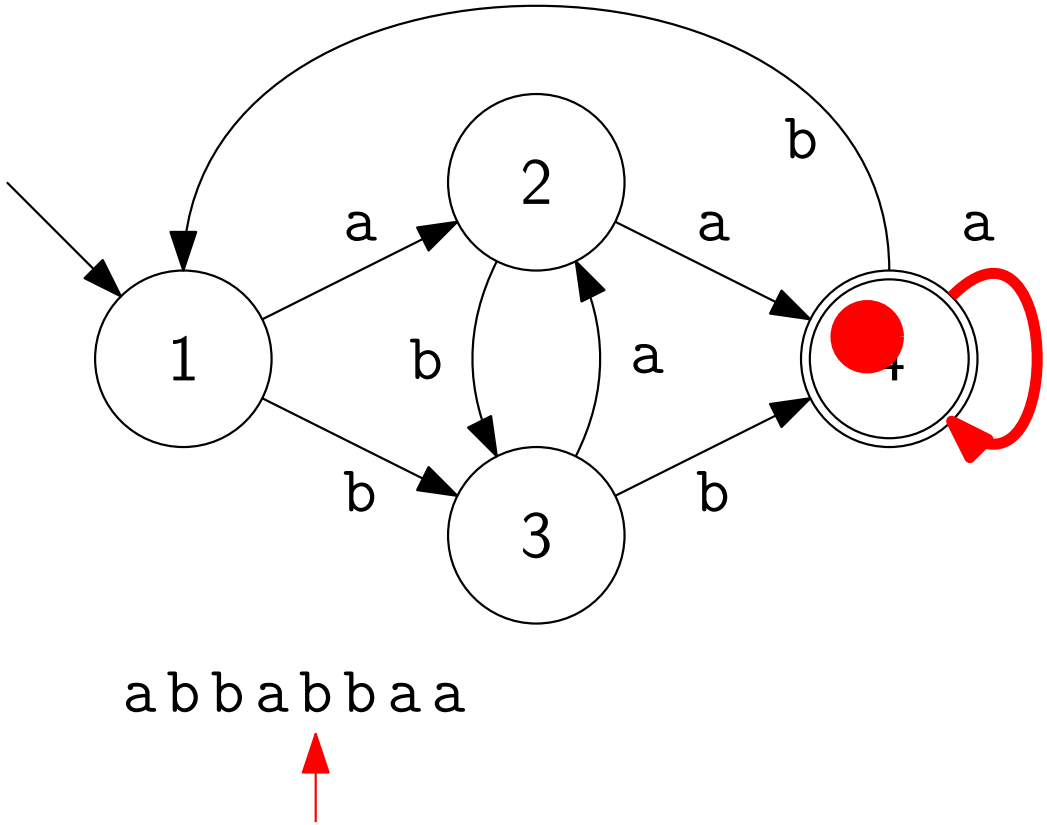
決定性有限オートマトン：文字列の受理



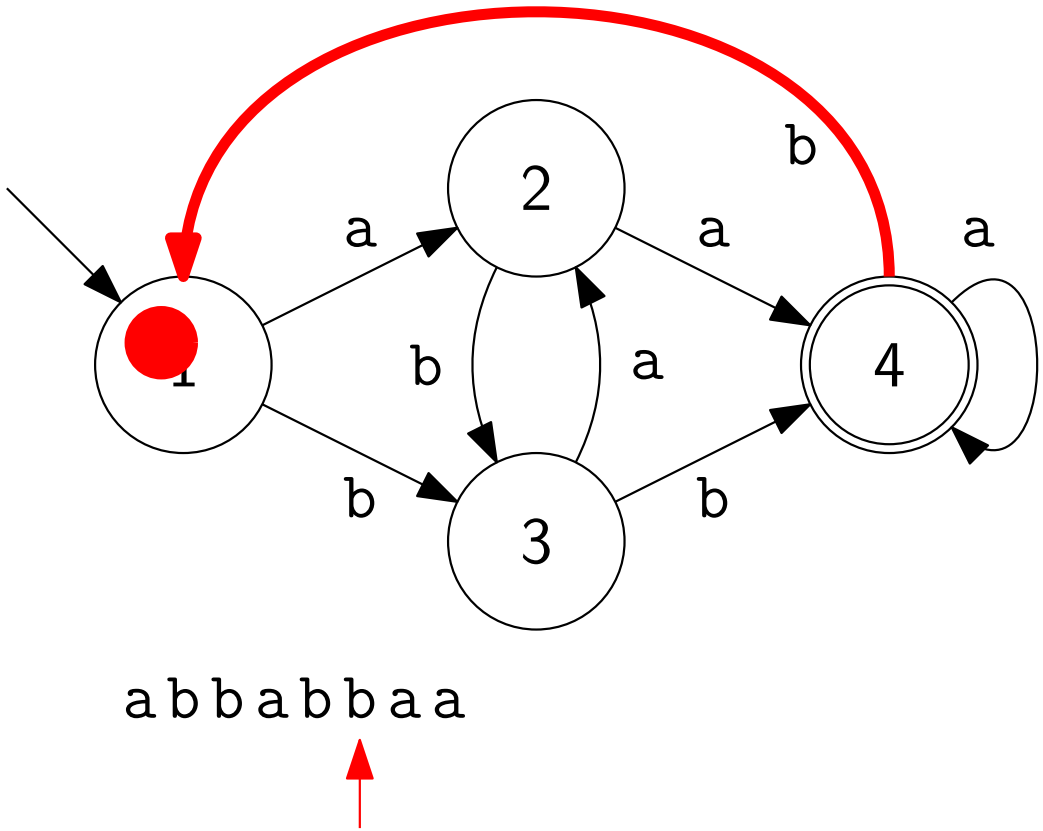
決定性有限オートマトン：文字列の受理



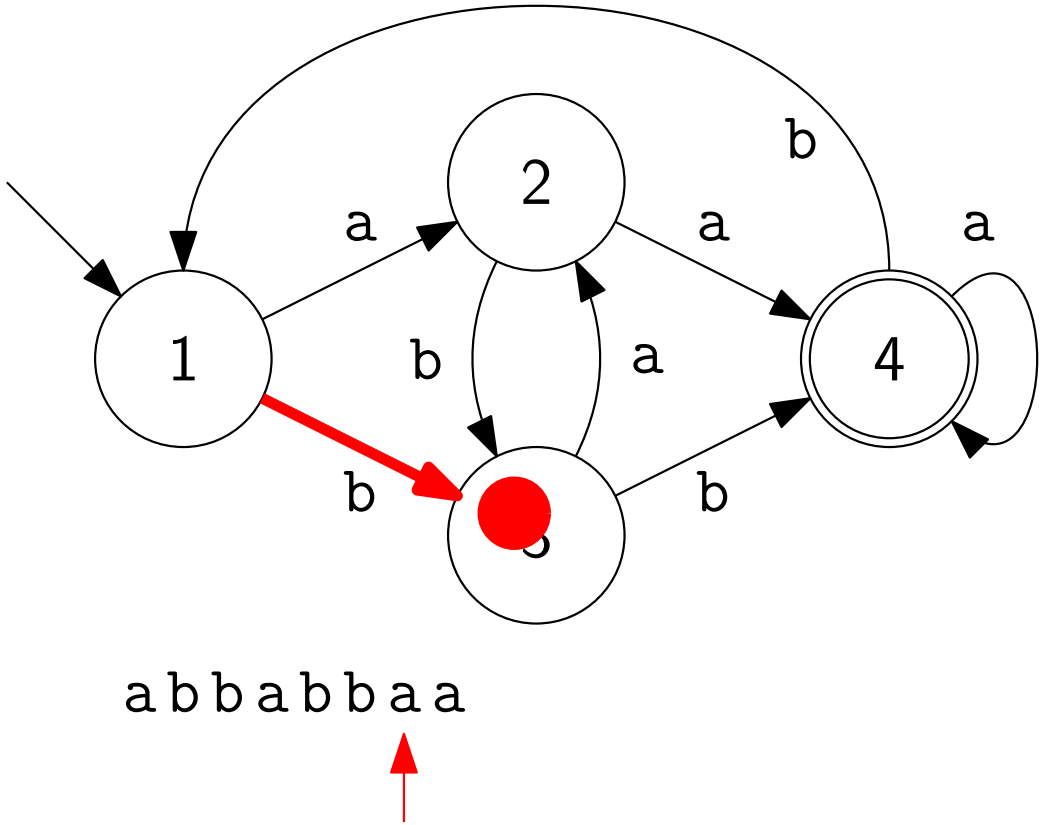
決定性有限オートマトン：文字列の受理



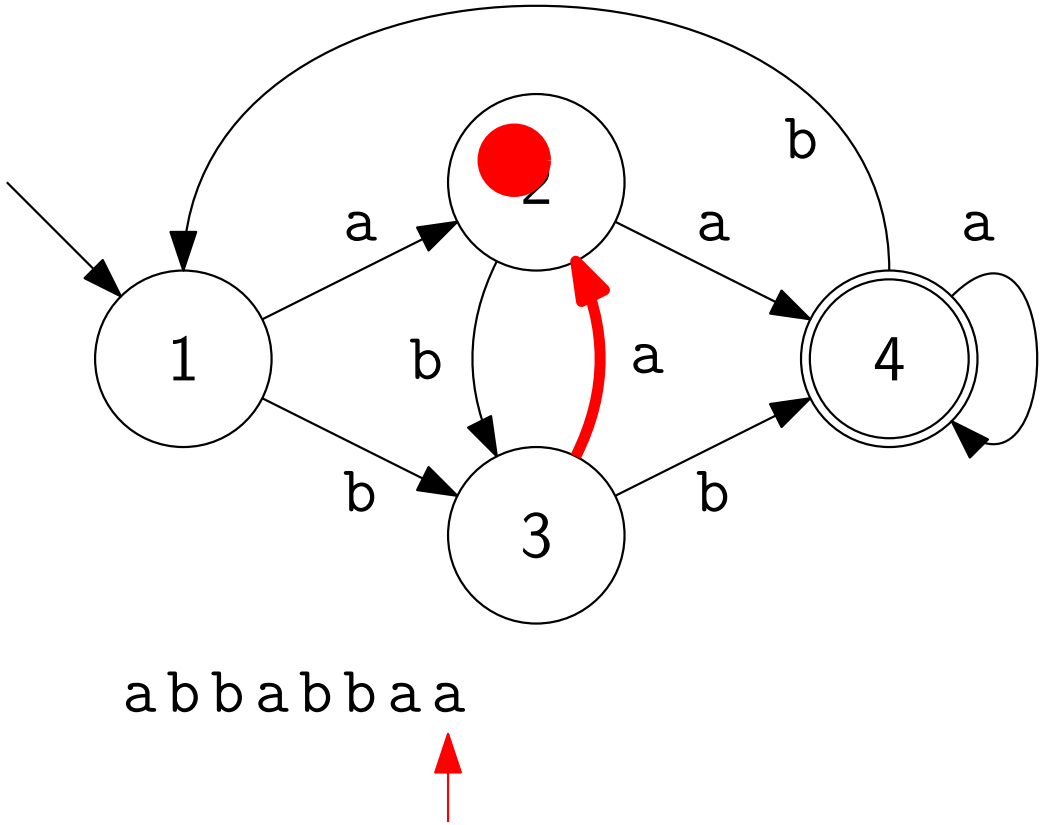
決定性有限オートマトン：文字列の受理



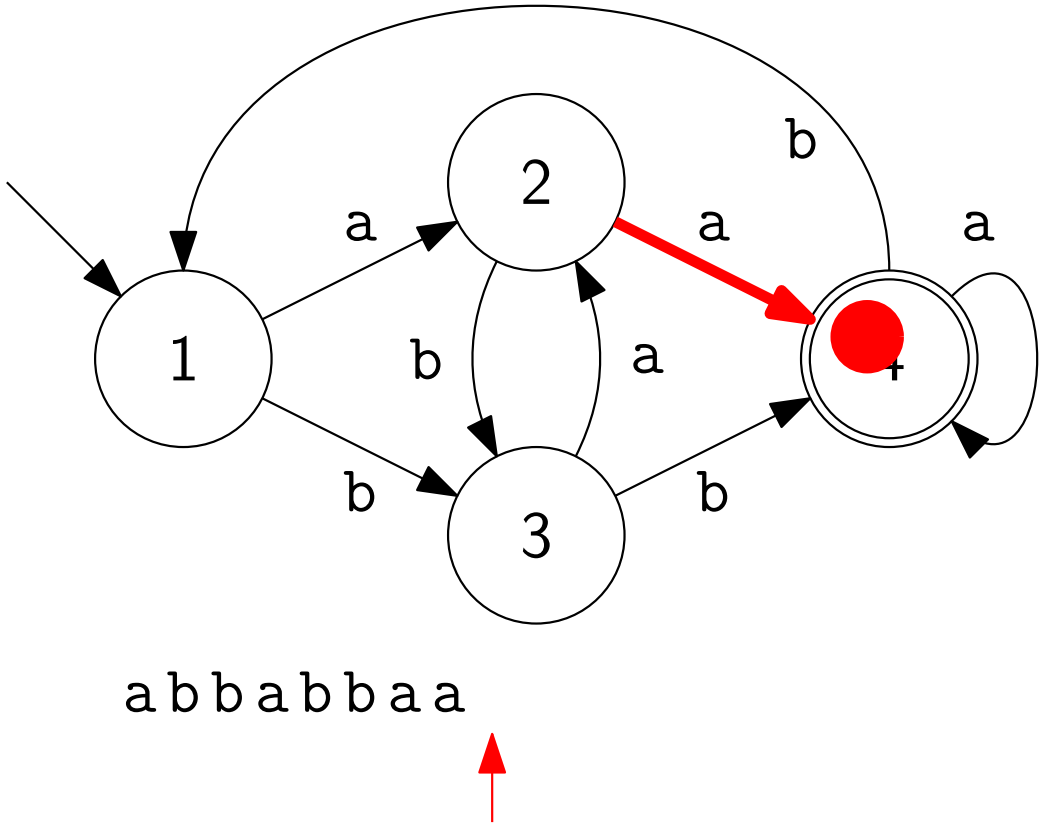
決定性有限オートマトン：文字列の受理



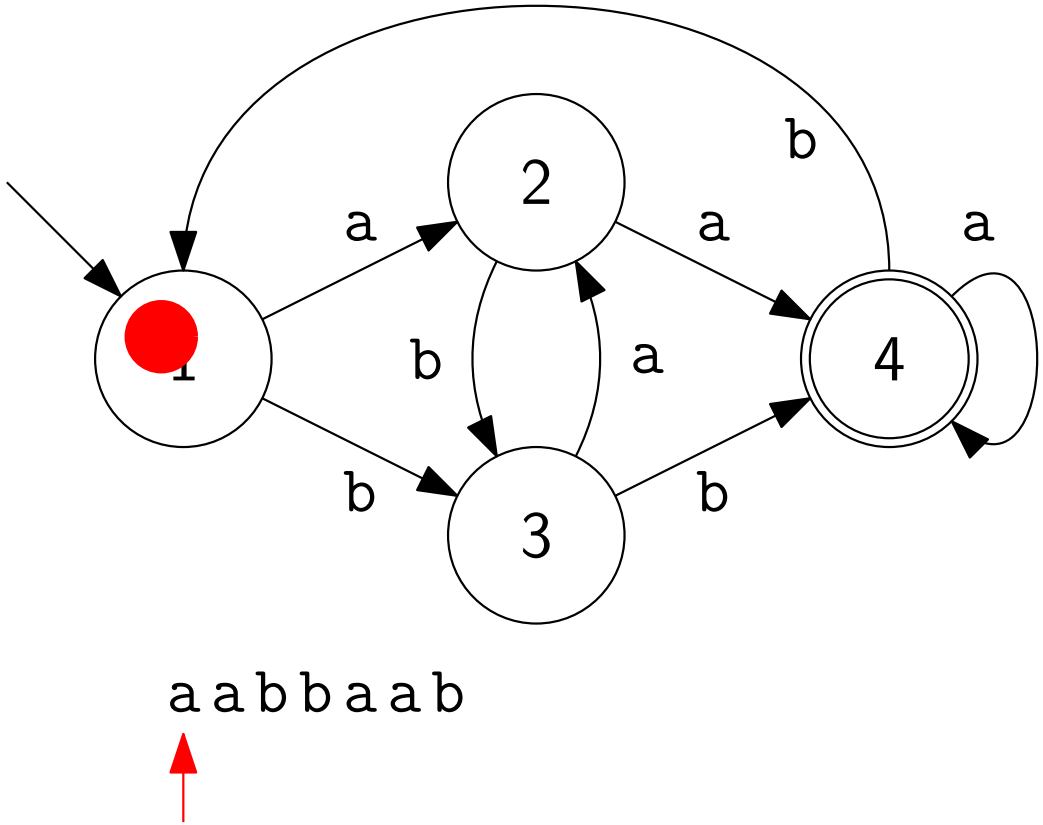
決定性有限オートマトン：文字列の受理



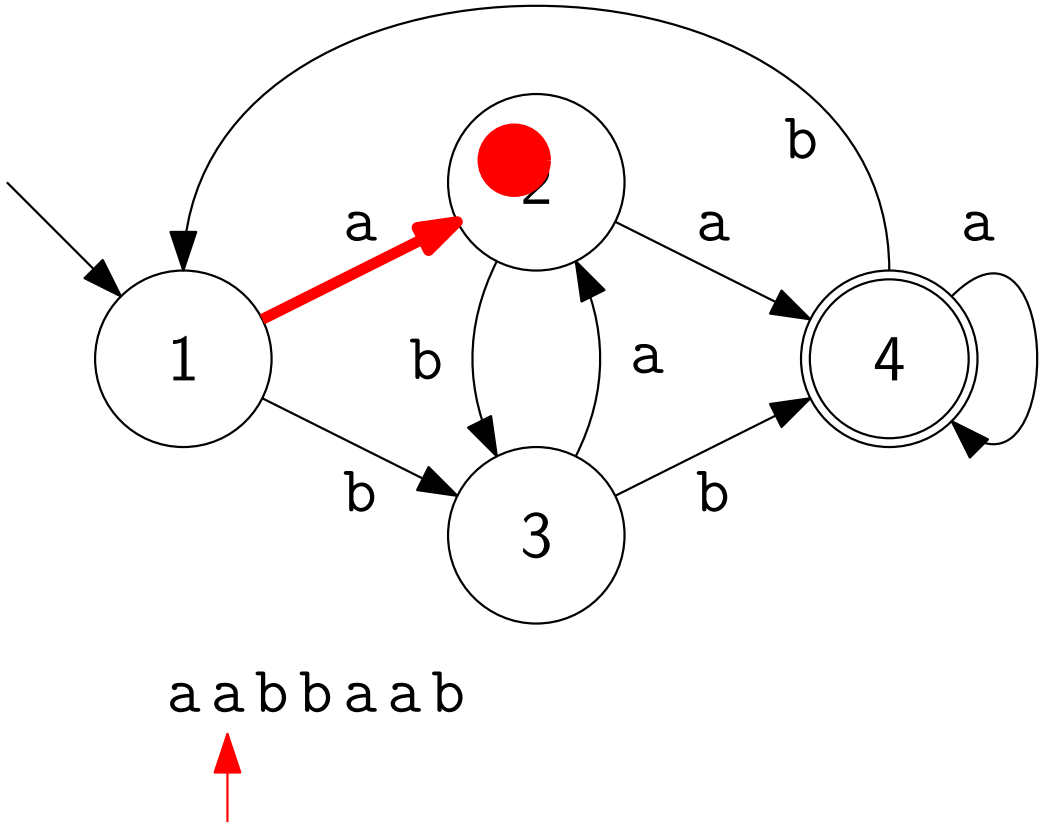
決定性有限オートマトン：文字列の受理



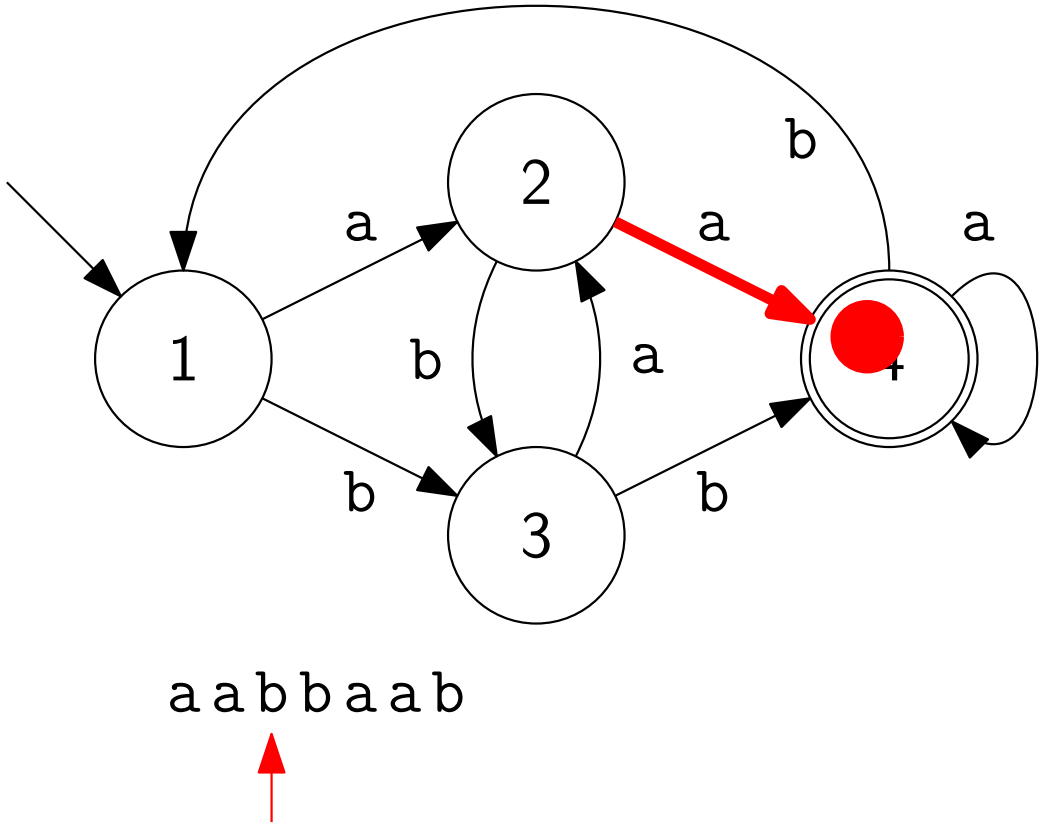
決定性有限オートマトン：文字列の棄却 (拒否)



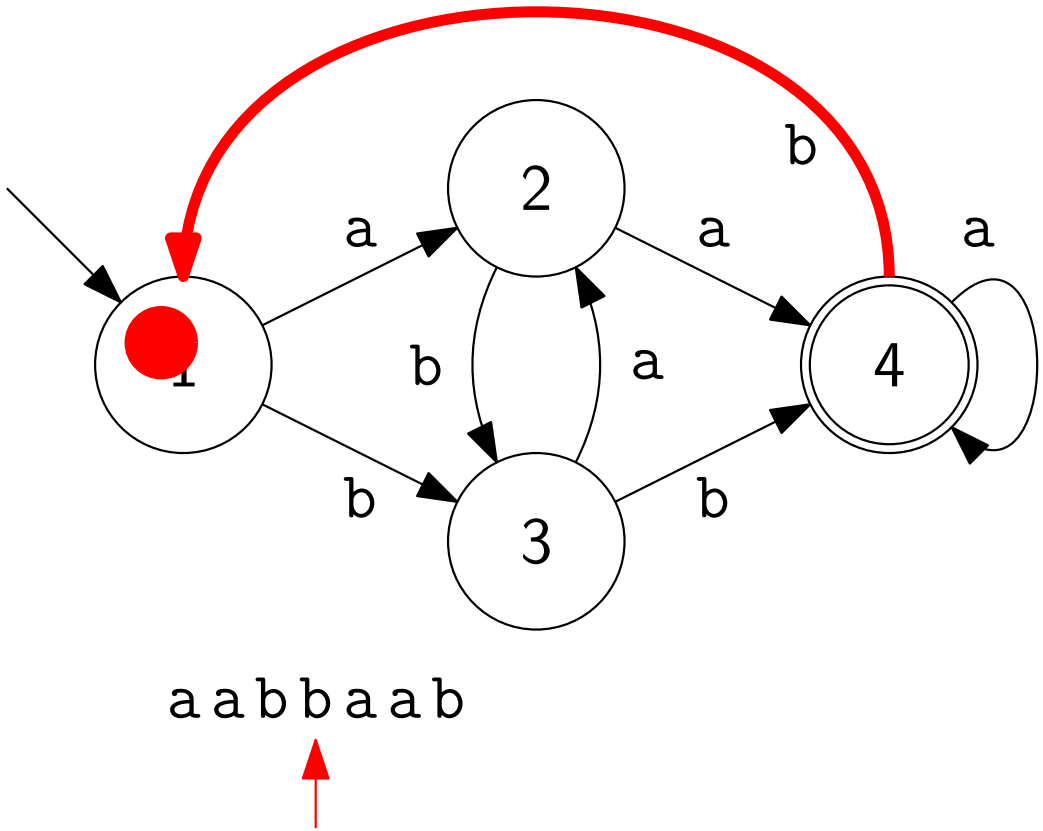
決定性有限オートマトン：文字列の棄却 (拒否)



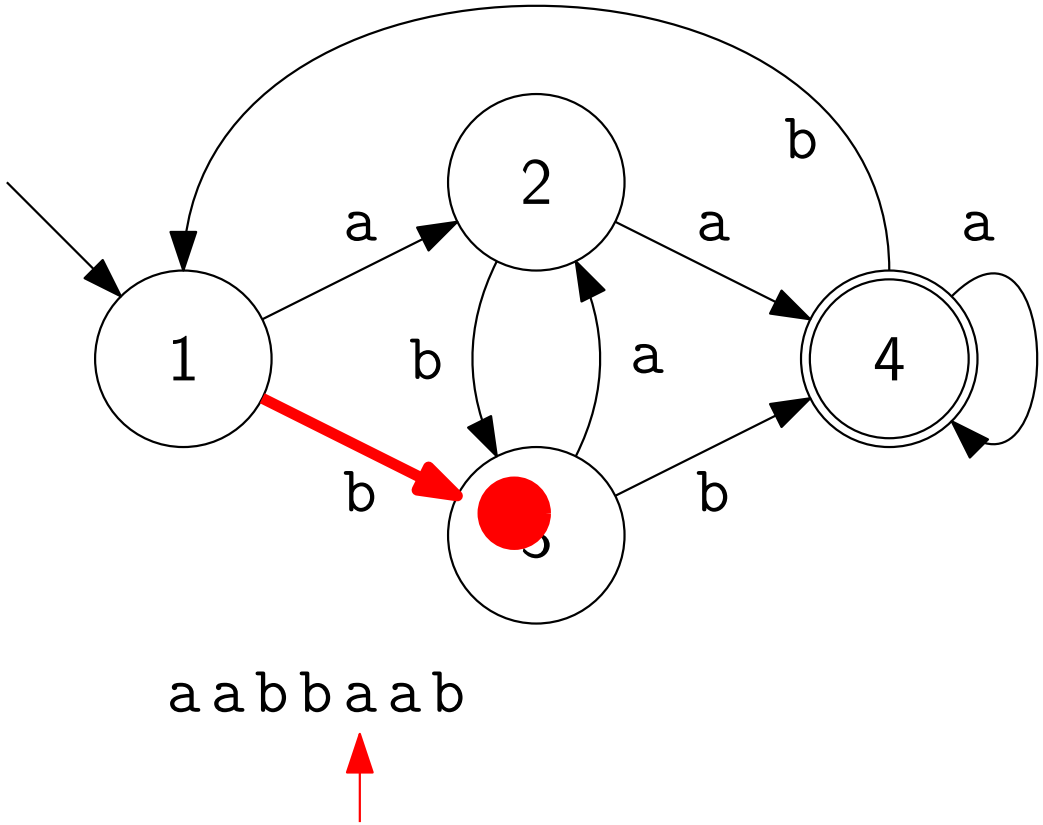
決定性有限オートマトン：文字列の棄却 (拒否)



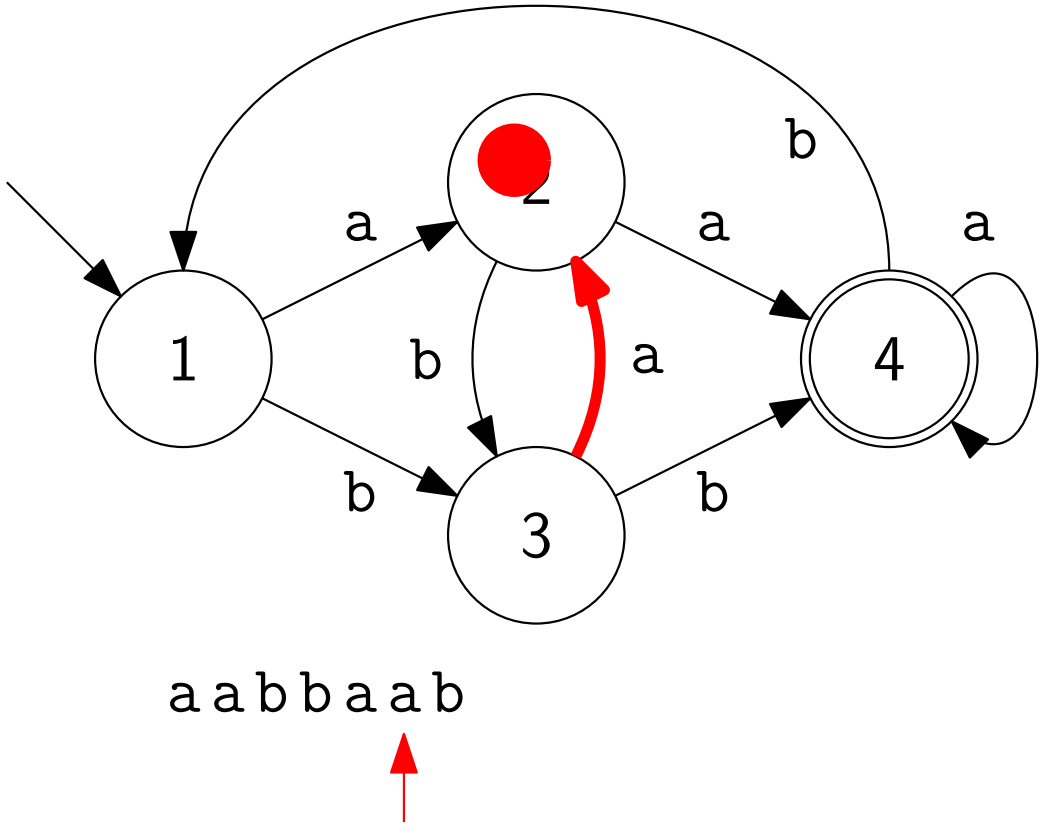
決定性有限オートマトン：文字列の棄却 (拒否)



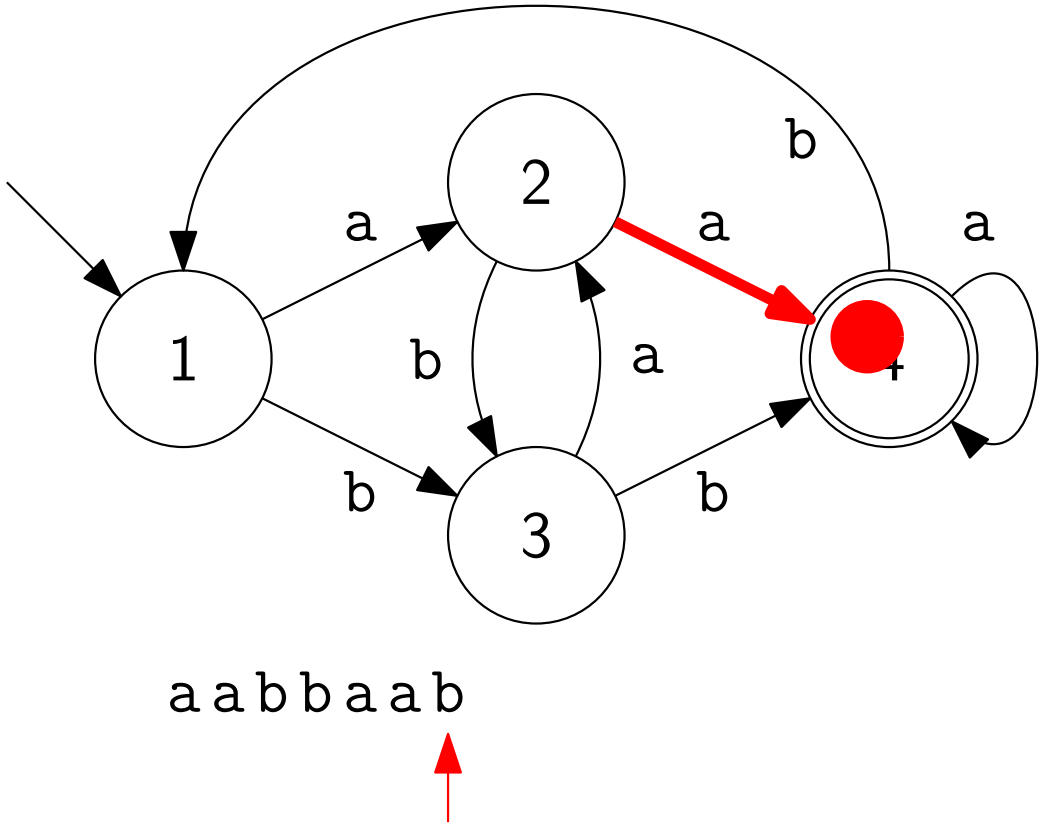
決定性有限オートマトン：文字列の棄却 (拒否)



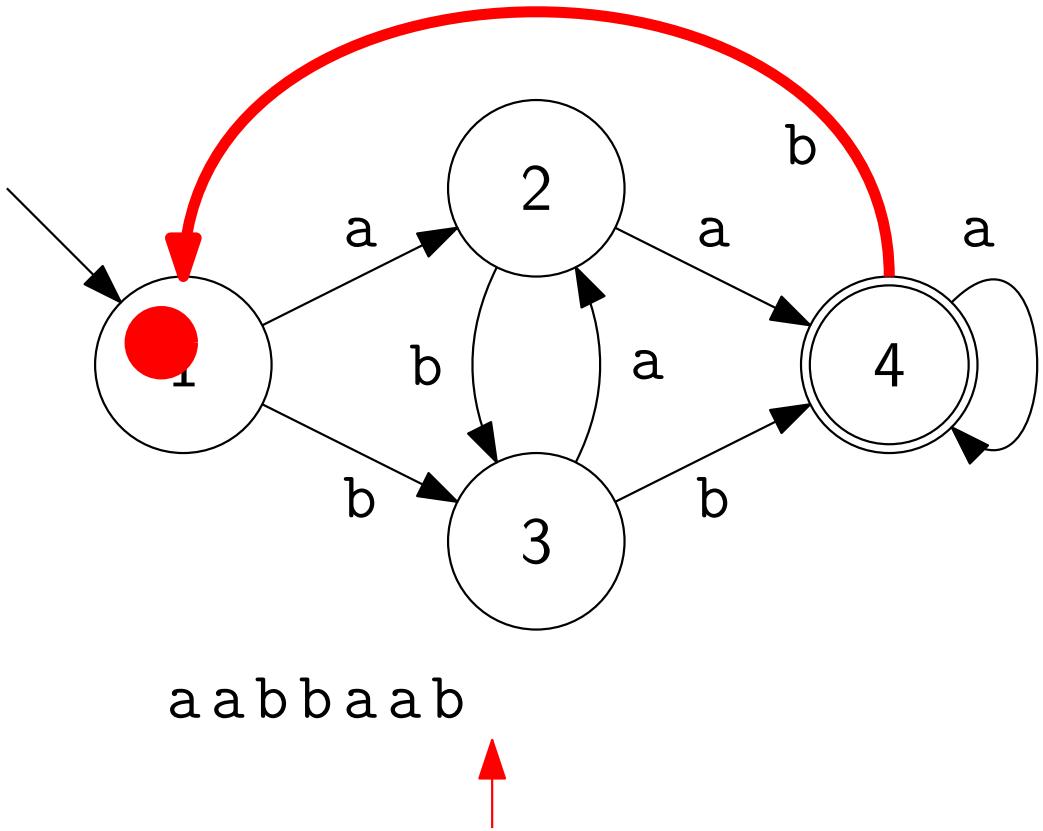
決定性有限オートマトン：文字列の棄却 (拒否)



決定性有限オートマトン：文字列の棄却 (拒否)



決定性有限オートマトン：文字列の棄却 (拒否)



計算機と決定性有限オートマトンの比較

	計算機	有限オートマトン
永続的ストレージ	HDD, SSD	なし
一時的ストレージ	RAM	現在の状態
入力	キーボードなど	文字列のストリーム
出力	ディスプレイなど	受理/棄却
プロセッサ	CPU コア	状態を変更する規則

Ruby で実行した様子

```
>> rulebook = DFARulebook.new([
     FARule.new(1, 'a', 2), FARule.new(1, 'b', 3),
     FARule.new(2, 'a', 4), FARule.new(2, 'b', 3),
     FARule.new(3, 'a', 2), FARule.new(3, 'b', 4),
     FARule.new(4, 'a', 4), FARule.new(4, 'b', 1)])
=> #<struct DFARulebook rules=[#<FARule 1 --a--> 2>, #<FARule
>> dfa_design = DFADesign.new(1, [4], rulebook)
=> #<struct DFADesign start_state=1, accept_states=[4], rulebo
>> dfa_design.accepts?('abbabbaa')
=> true
>> dfa_design.accepts?('aabbaab')
=> false
```

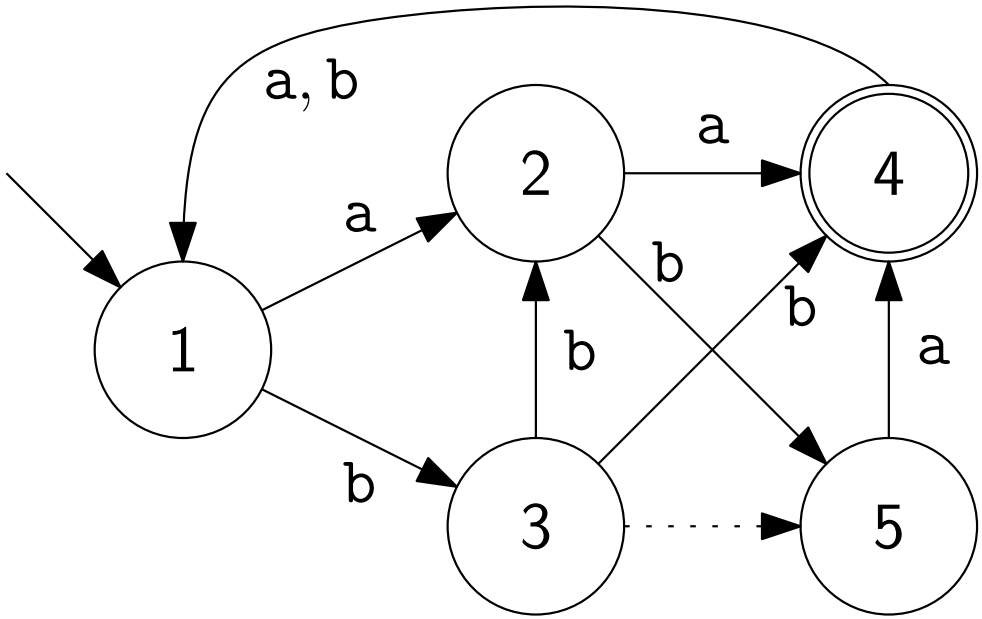
目次

非決定性有限オートマトン

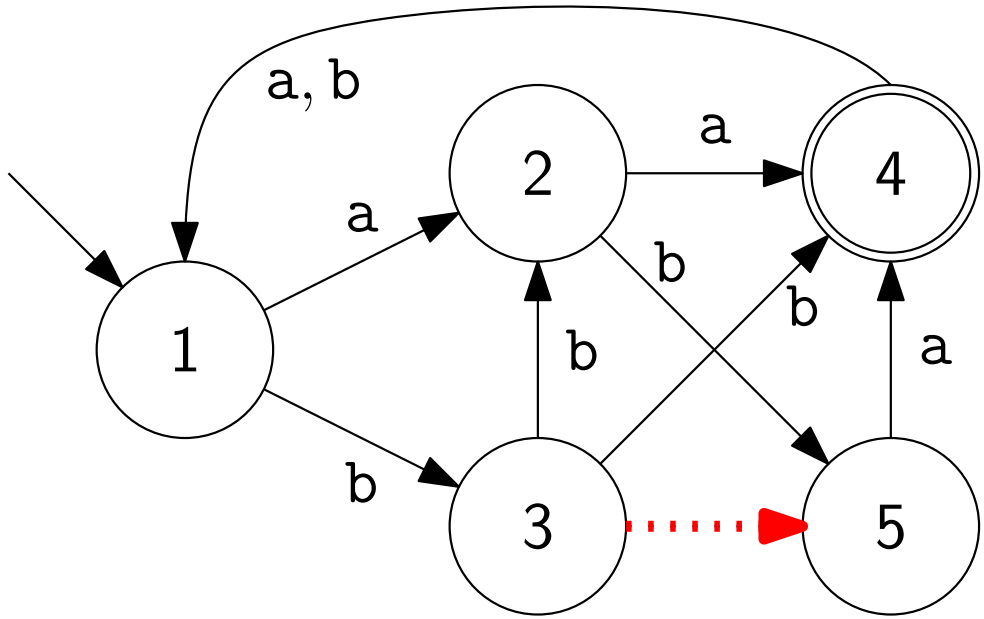
非決定性有限オートマトンは決定性有限オートマトンと次が違う

- 遷移先の状態が一意に決まらない場合がある
- 遷移先がない場合がある
- 自由移動 (ε 遷移) があるかもしれない

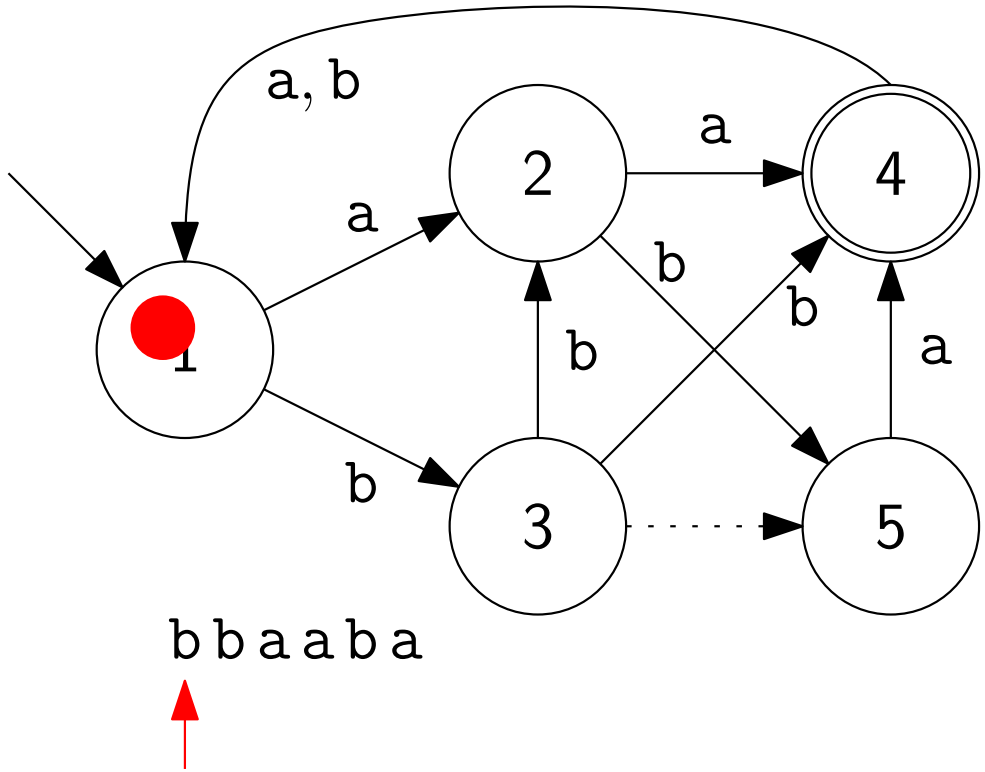
非決定性有限オートマトンの図示



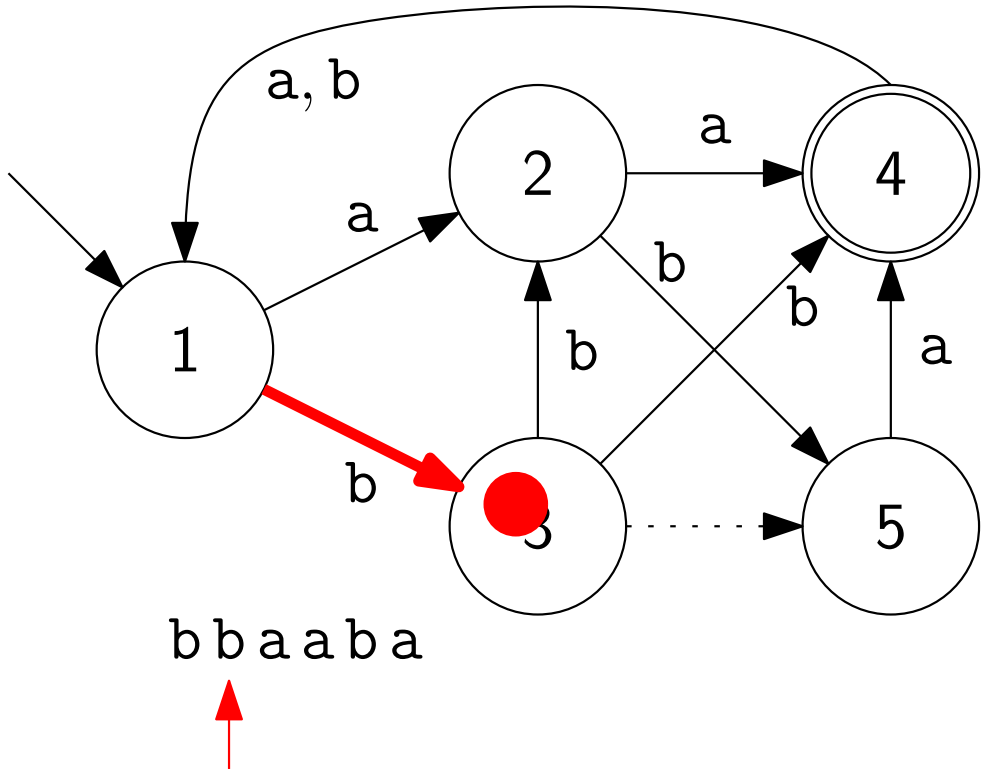
自由移動 (ε 遷移)



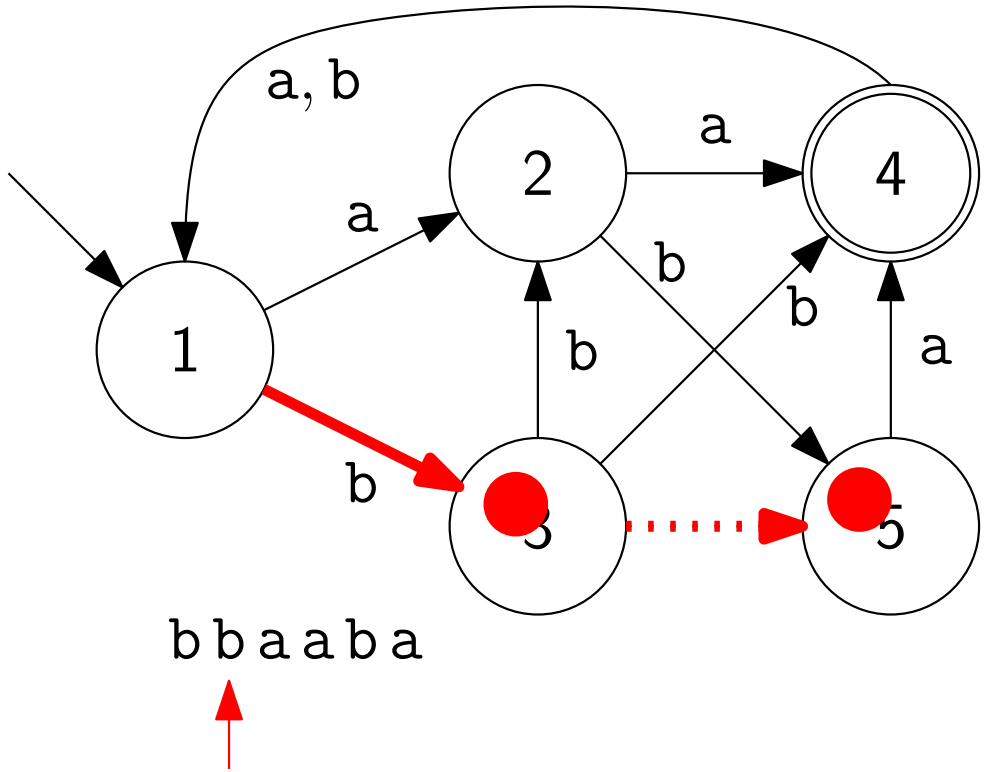
非決定性有限オートマトン：文字列の受理



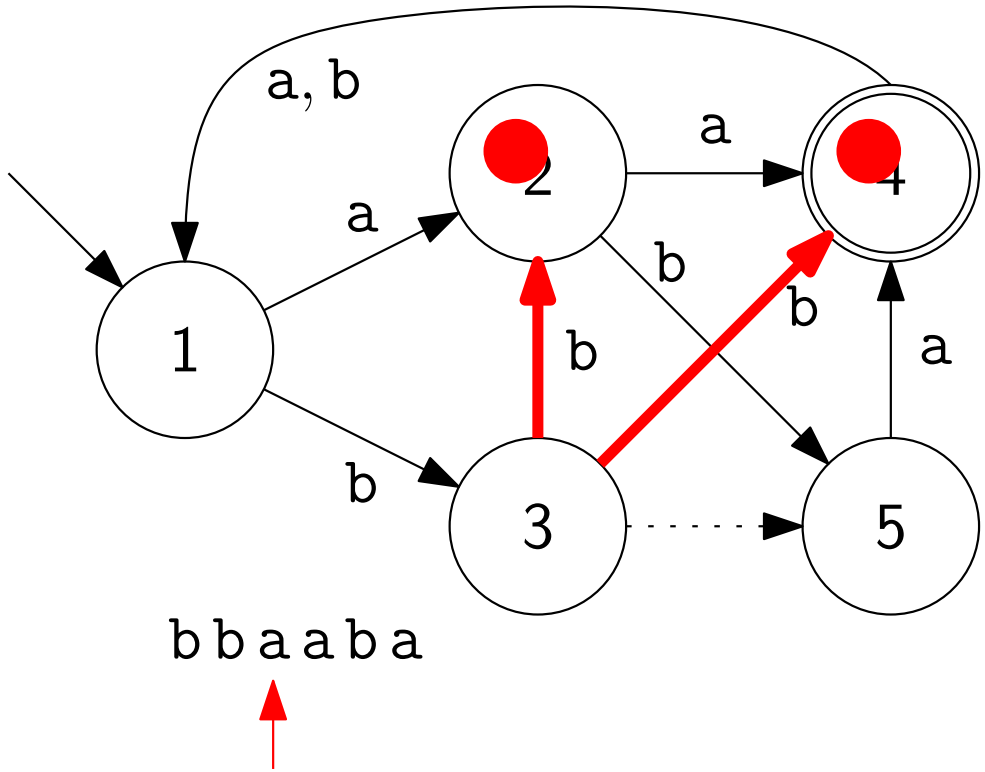
非決定性有限オートマトン：文字列の受理



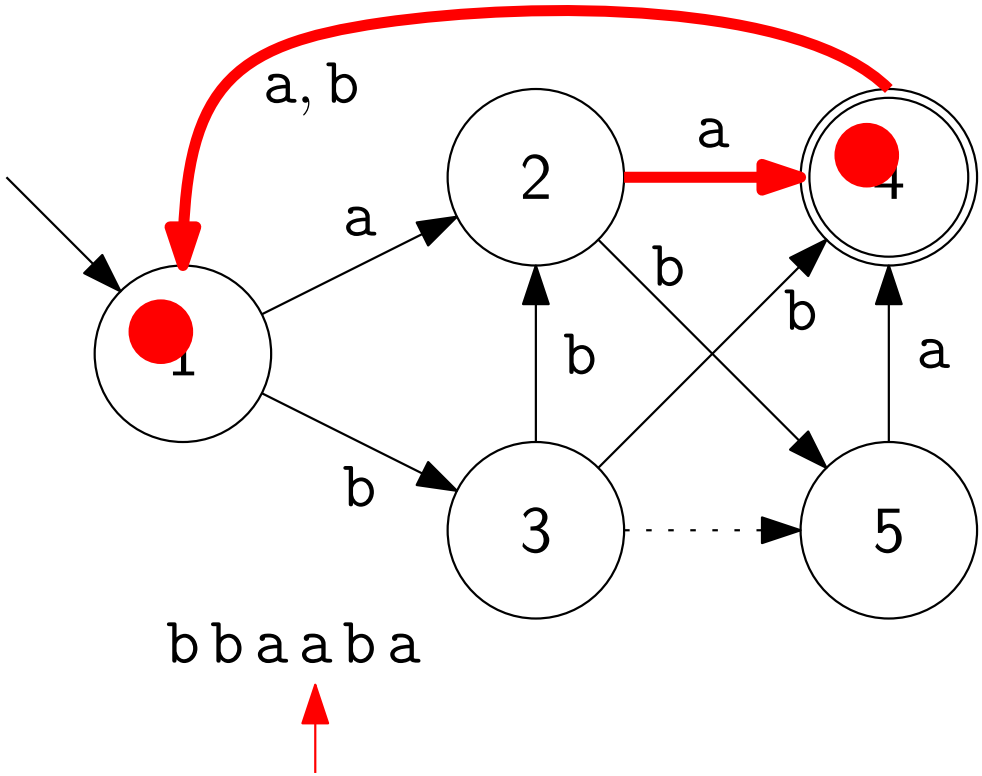
非決定性有限オートマトン：文字列の受理



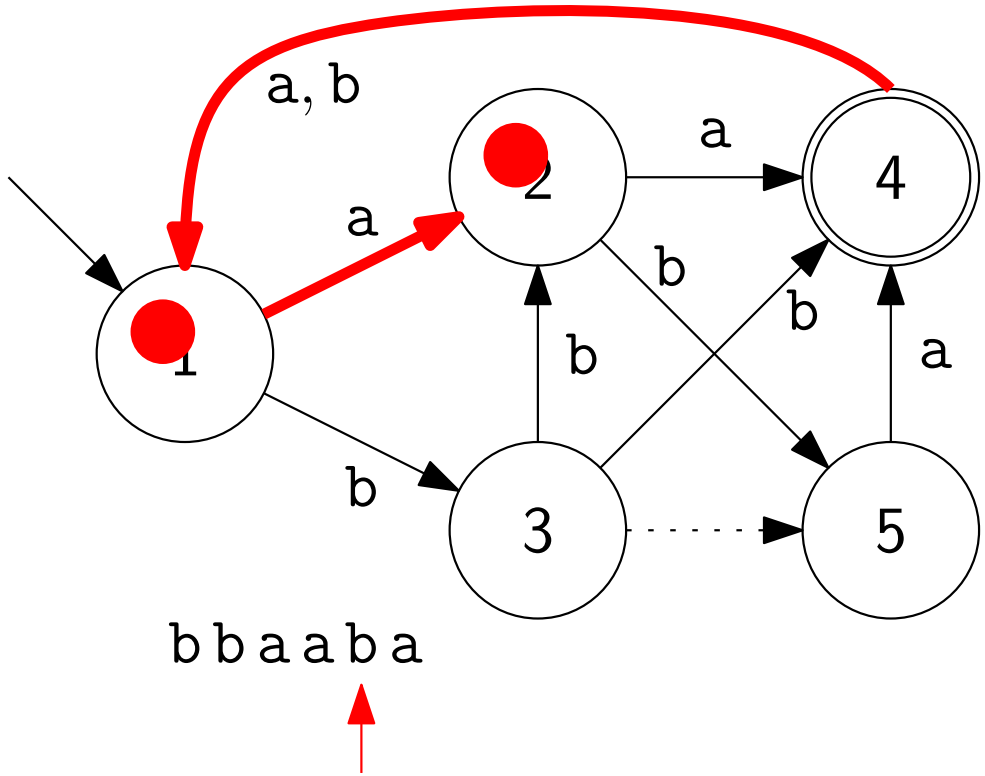
非決定性有限オートマトン：文字列の受理



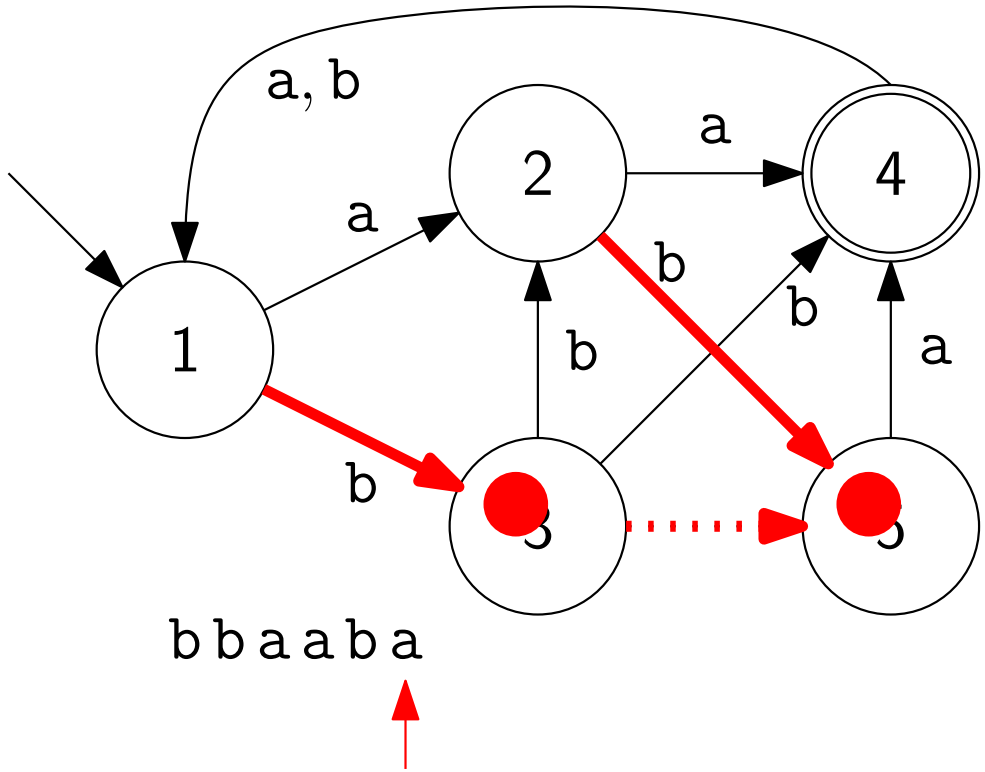
非決定性有限オートマトン：文字列の受理



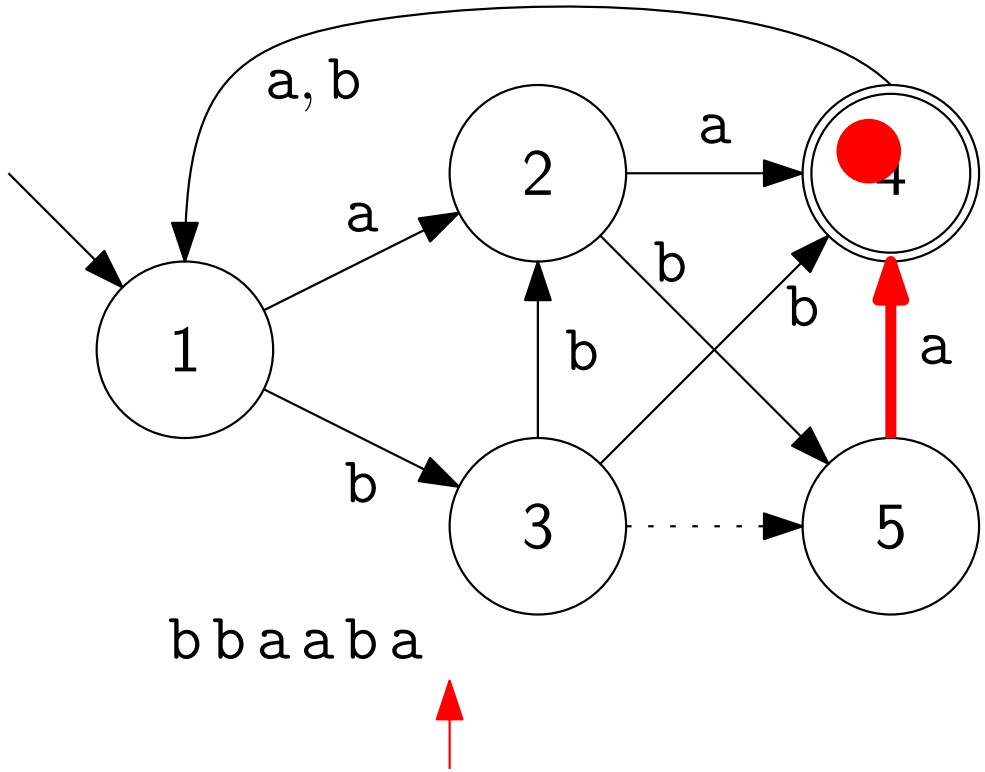
非決定性有限オートマトン：文字列の受理



非決定性有限オートマトン：文字列の受理



非決定性有限オートマトン：文字列の受理



Ruby で実行した様子

```
>> rulebook = NFARulebook.new([
     FARule.new(1, 'a', 2), FARule.new(1, 'b', 3),
     FARule.new(2, 'a', 4), FARule.new(2, 'b', 5),
     FARule.new(3, nil, 5), FARule.new(3, 'b', 2),
     FARule.new(3, 'b', 4), FARule.new(4, 'a', 1),
     FARule.new(4, 'b', 1), FARule.new(5, 'a', 4)])
=> #<struct NFARulebook rules=[#<FARule 1 --a--> 2>, #<FARule
>> nfa_design = NFADesign.new(1, [4], rulebook)
=> #<struct NFADesign start_state=1, accept_states=[4], rulebo
>> nfa_design.accepts?('bbaaba')
=> true
>> nfa_design.accepts?('bbbab')
=> false
```

目次

等価性

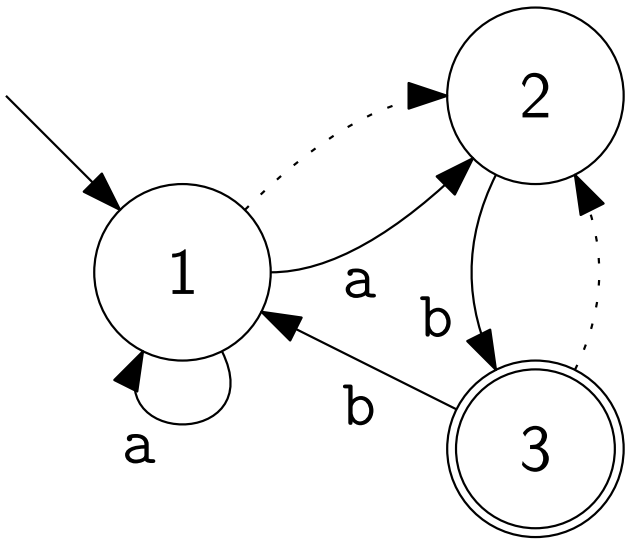
重要な事実

文字列の集合 L (言語) に対して，次の 3 つは等価 (同値)

1. L の文字列のみを受理する決定性有限オートマトンがある
2. L の文字列のみを受理する非決定性有限オートマトンがある
3. L の文字列のみにマッチする正規表現がある

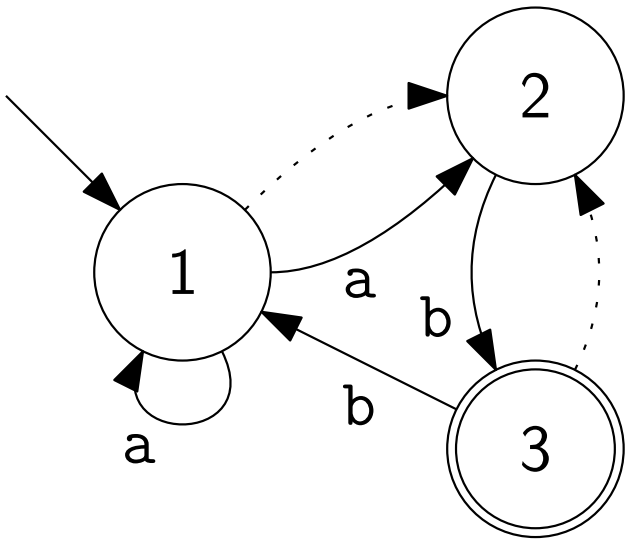
『 2 $\Rightarrow$ 1』の証明は，
非決定性有限オートマトンから決定性有限オートマトンを構成する
(「部分集合構成法」や「冪集合構成法」と呼ばれる)
(subset construction) (powerset construction)

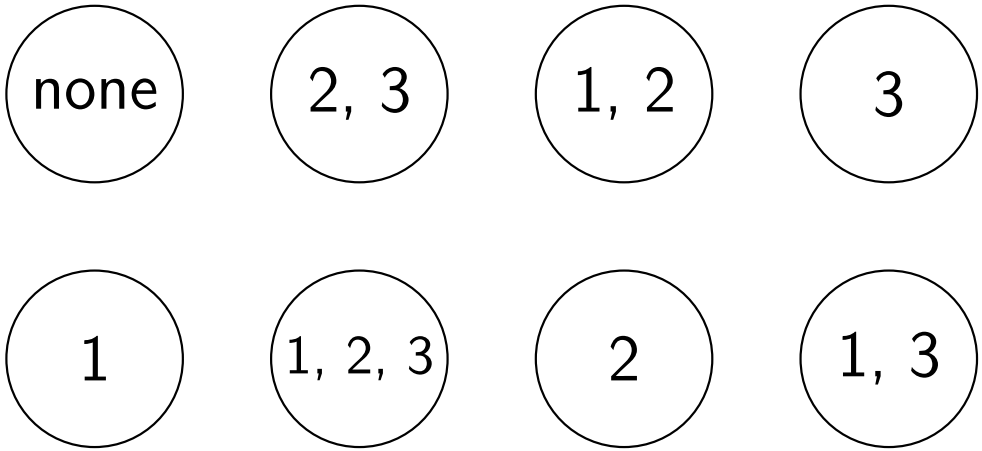
部分集合構成法



考える非決定性有限オートマトン

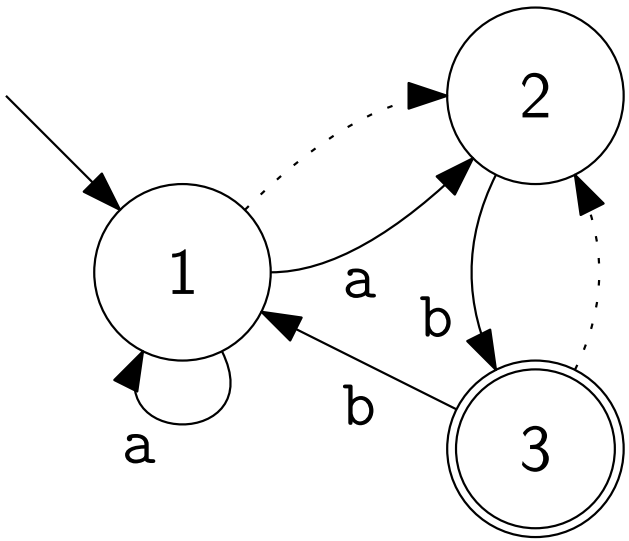
部分集合構成法

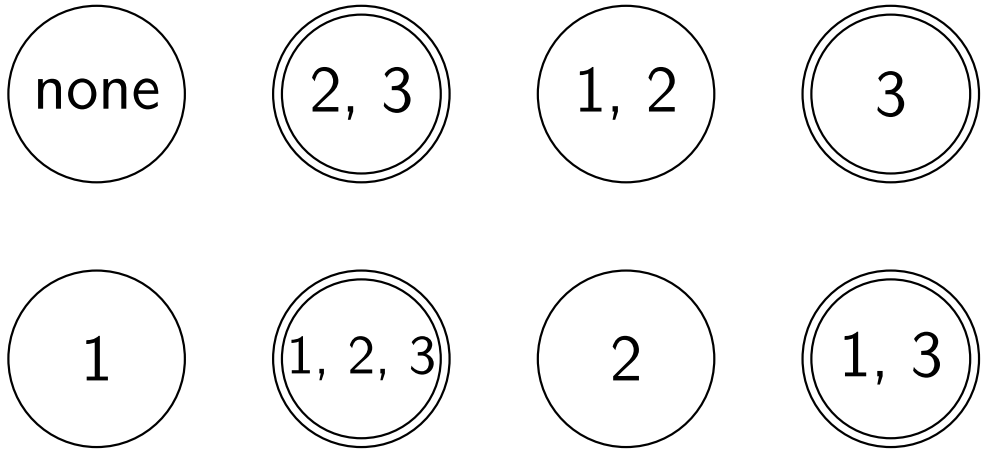




状態のすべての組合せを考えて，新たな状態とする

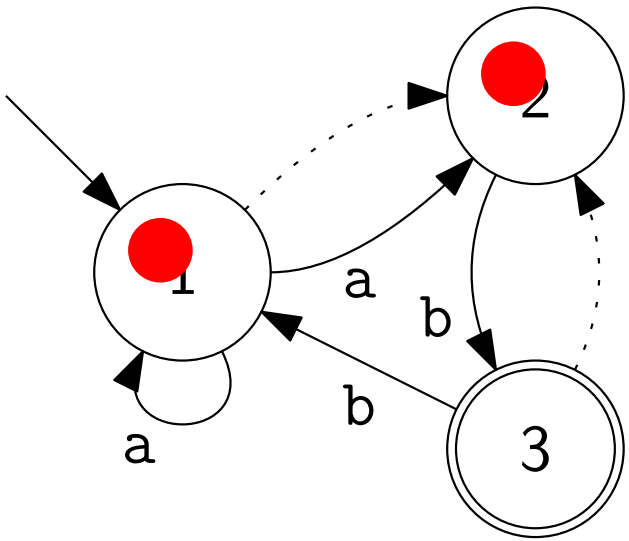
部分集合構成法

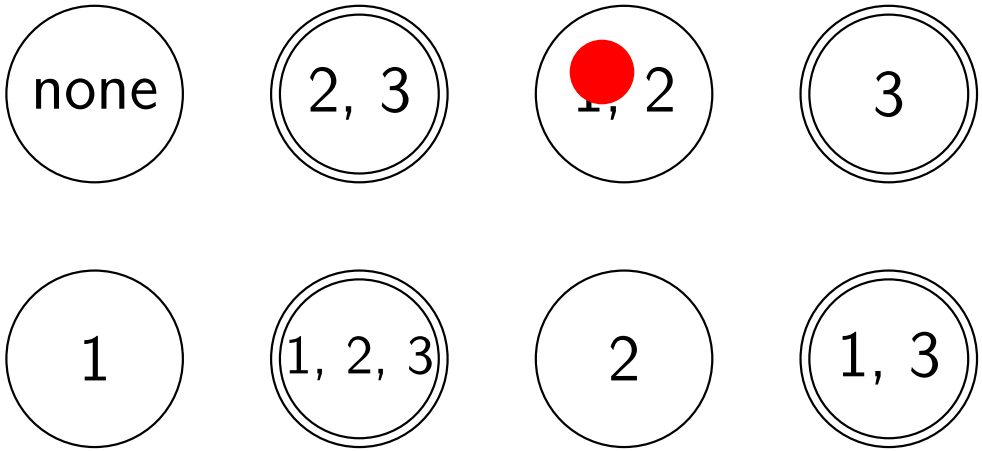




左のオートマトンの受理状態を含む状態を受理状態とする

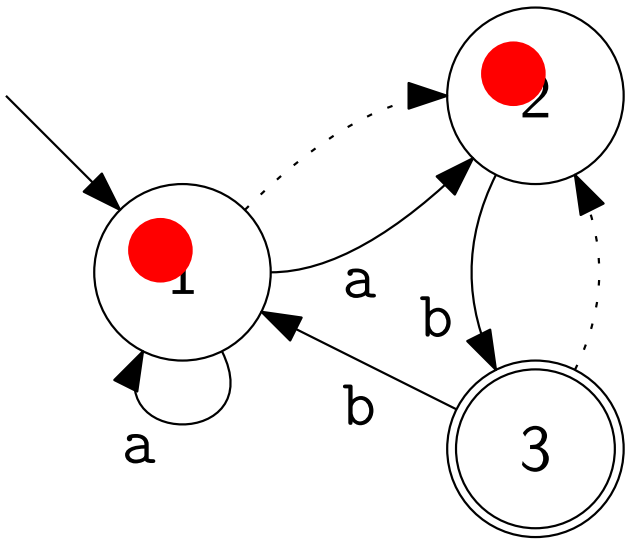
部分集合構成法

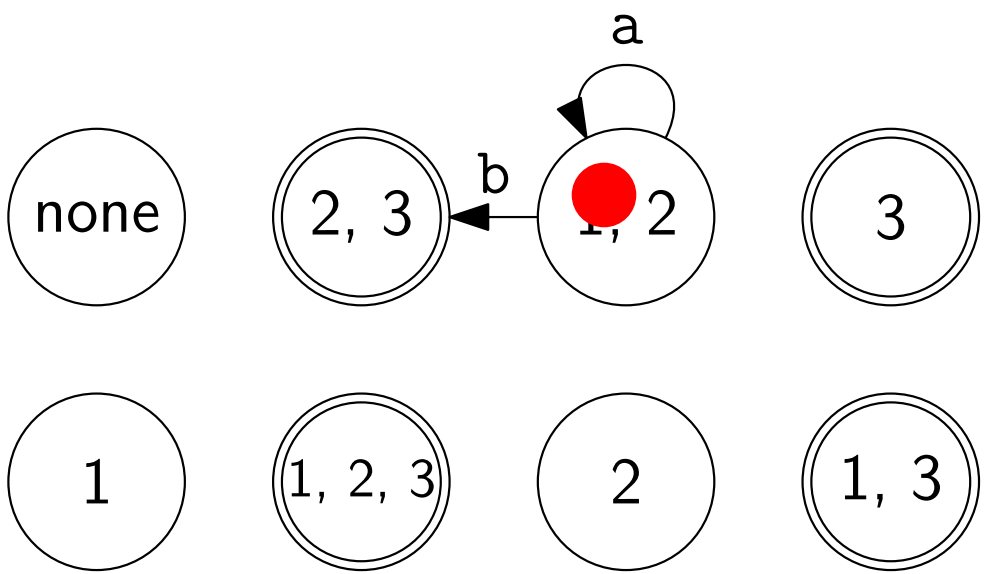




状態「$\{1, 2\}$」を考える

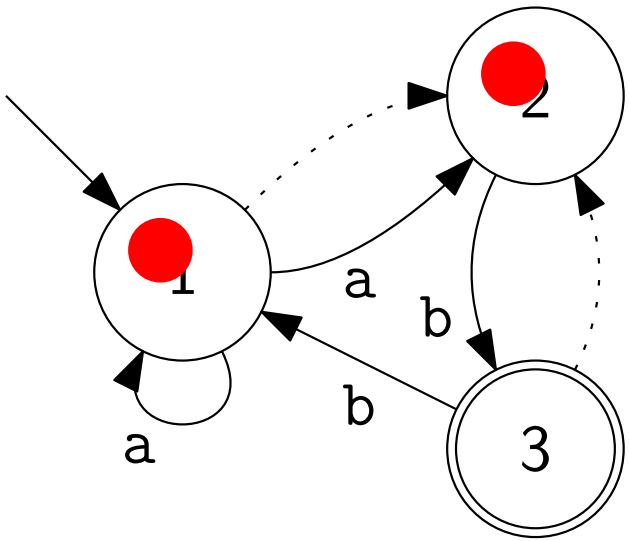
部分集合構成法

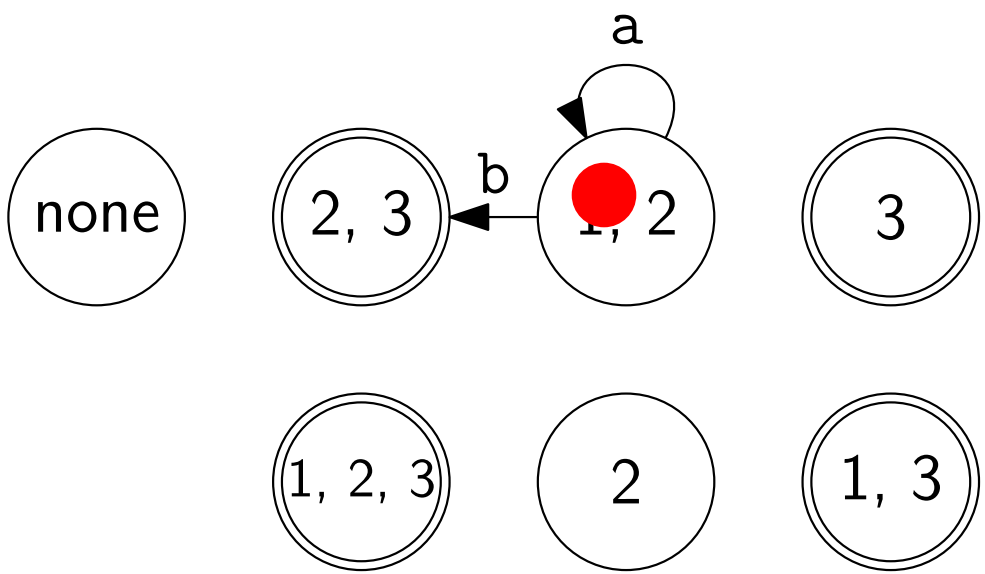




a，b を読んだときの移動を考える

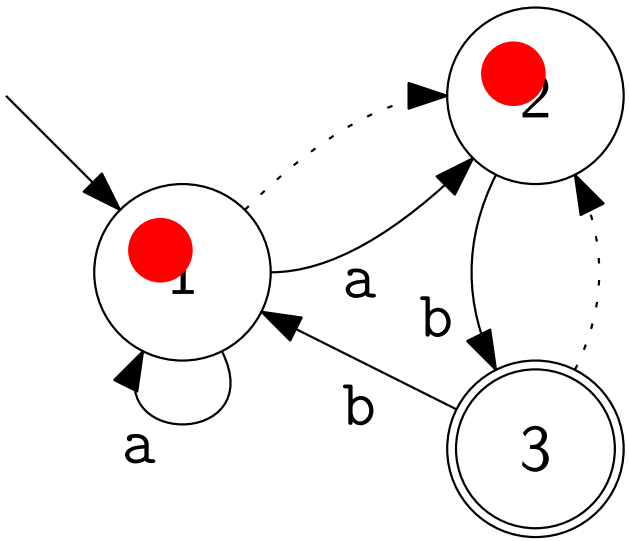
部分集合構成法

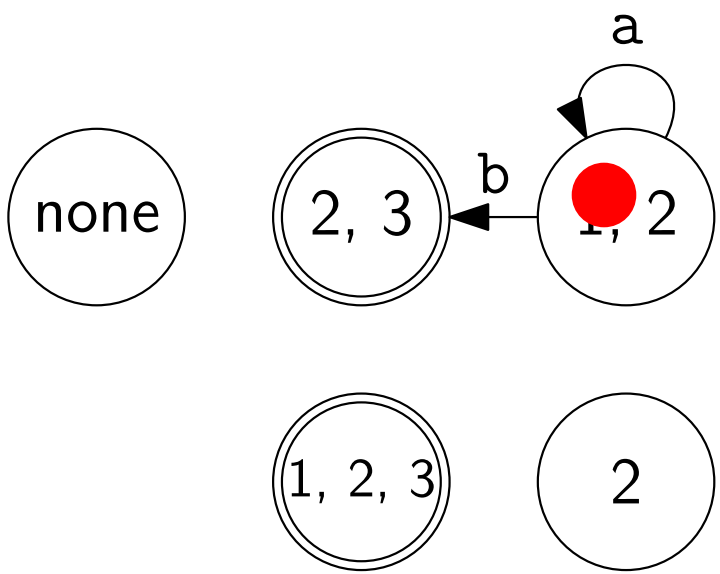




状態「$\{1\}$」は実現できないので，削除する　　(自由移動のため)

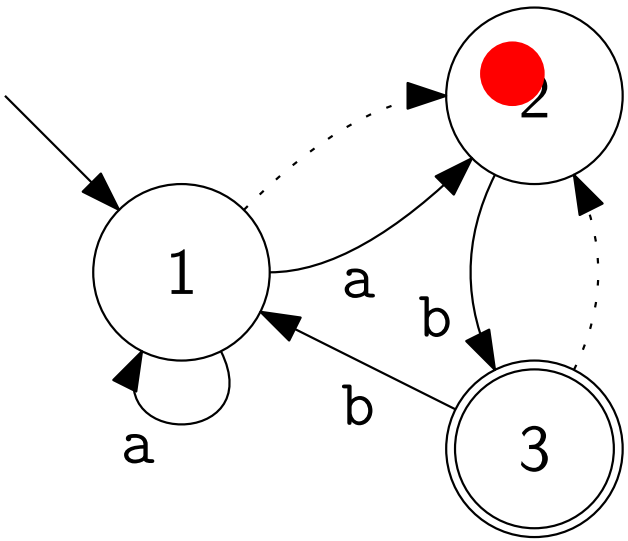
部分集合構成法

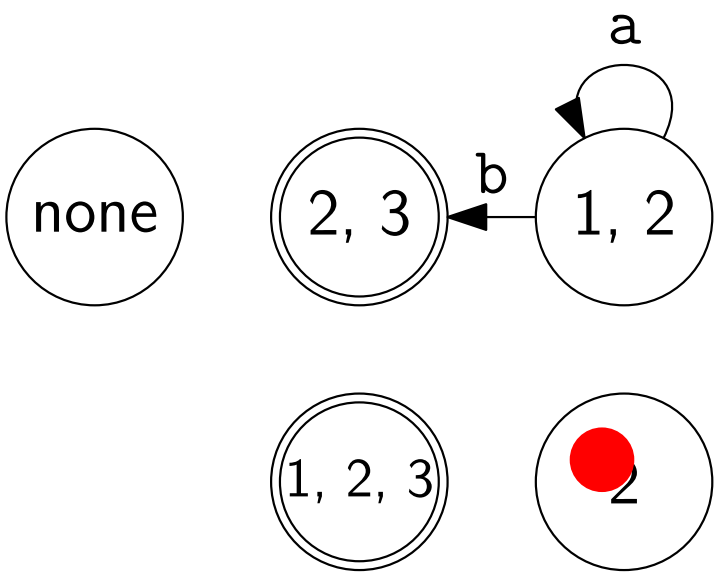




状態「$\{3\}$」,「$\{1,3\}$」も実現できないので，削除する (自由移動のため)

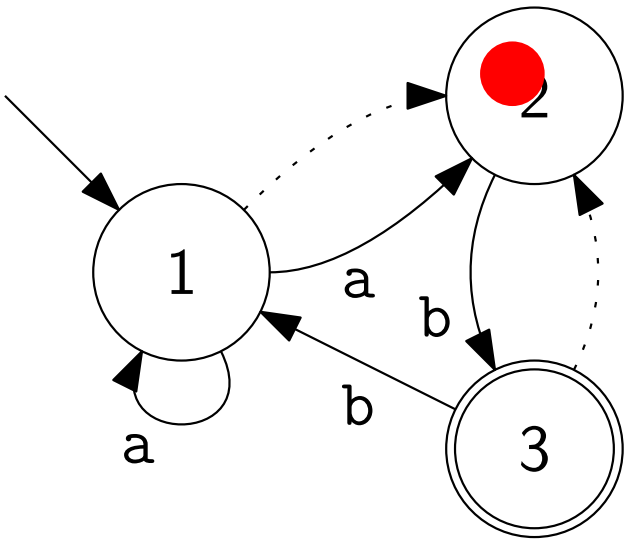
部分集合構成法

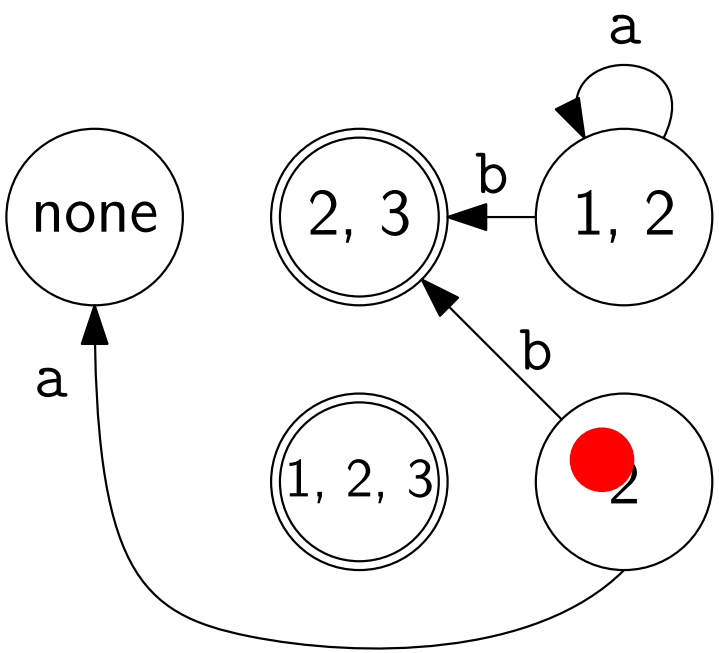




他の状態からの移動も同様に考える

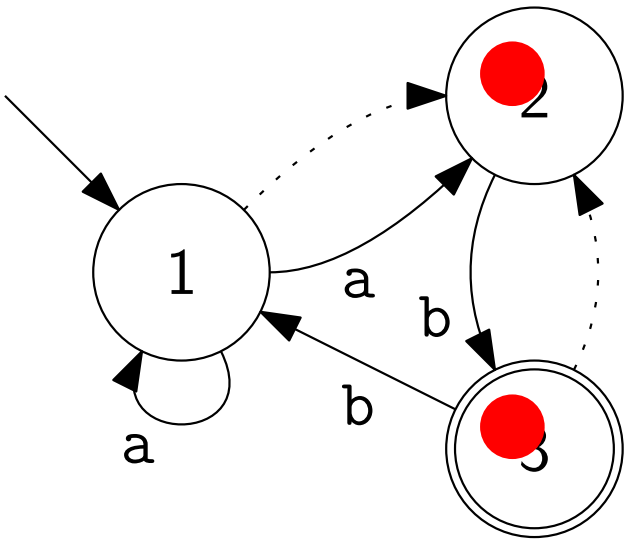
部分集合構成法

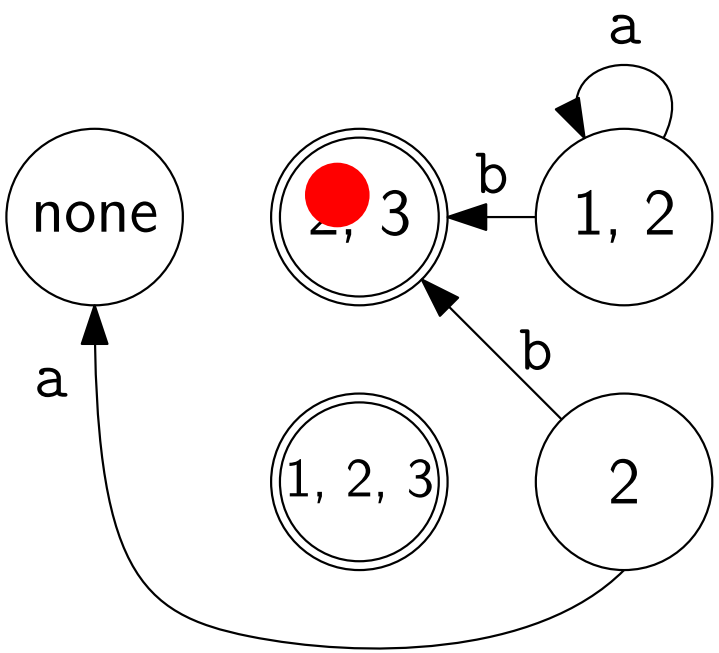




他の状態からの移動も同様に考える

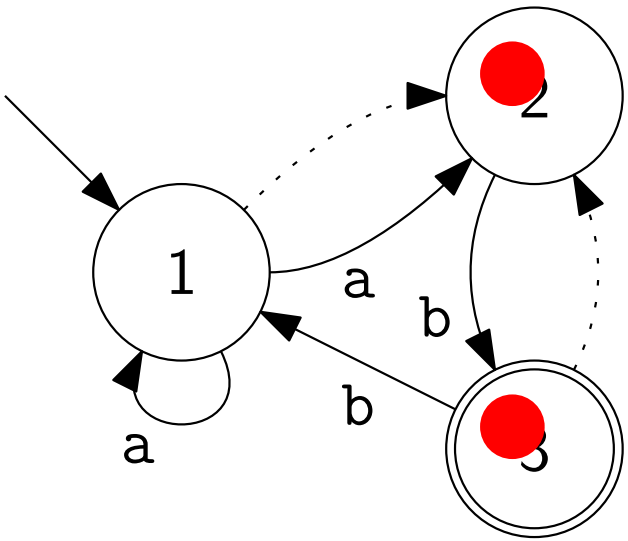
部分集合構成法

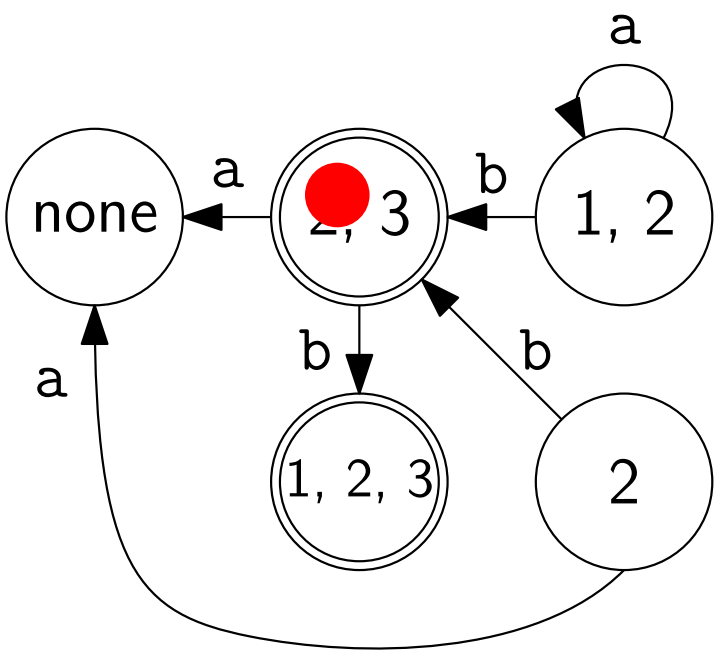




他の状態からの移動も同様に考える

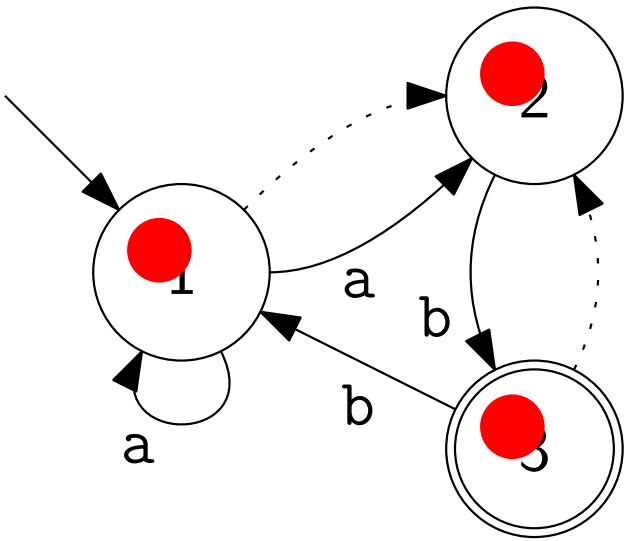
部分集合構成法

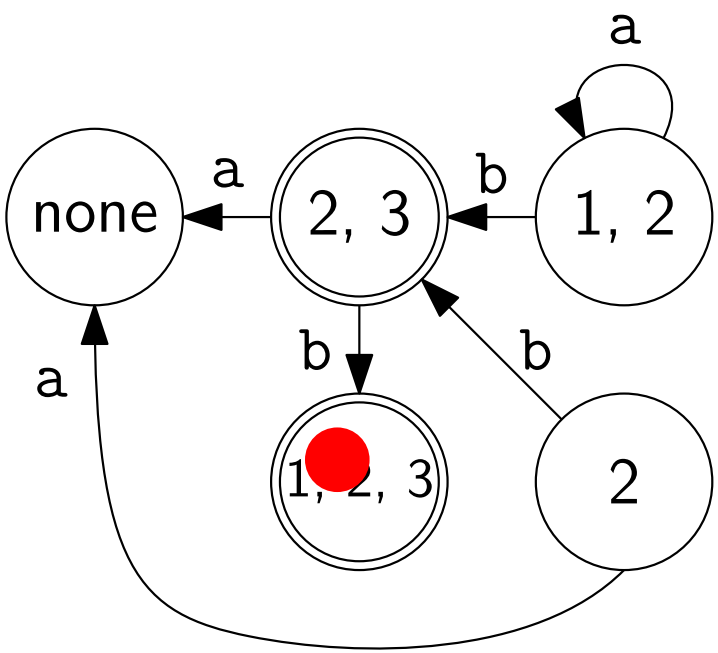




他の状態からの移動も同様に考える

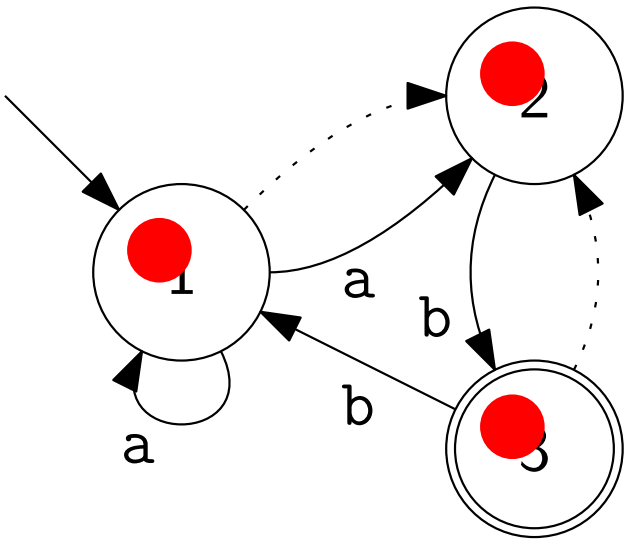
部分集合構成法

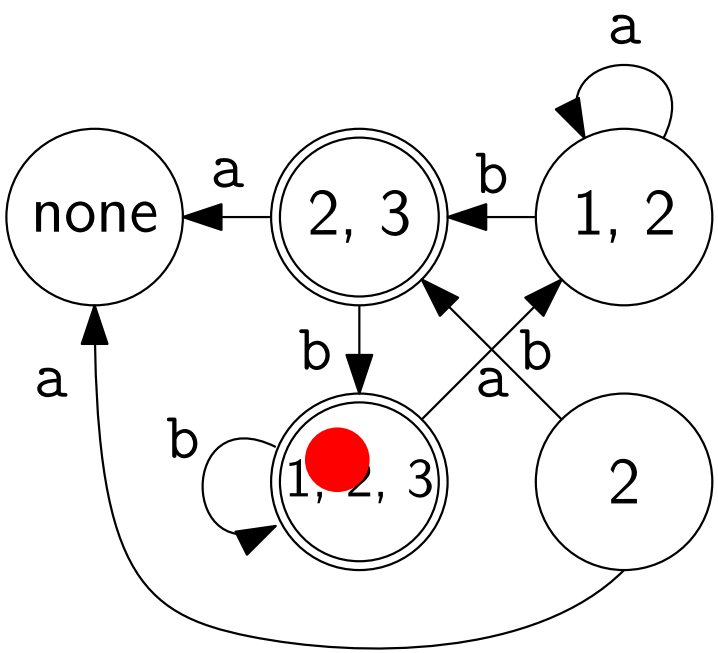




他の状態からの移動も同様に考える

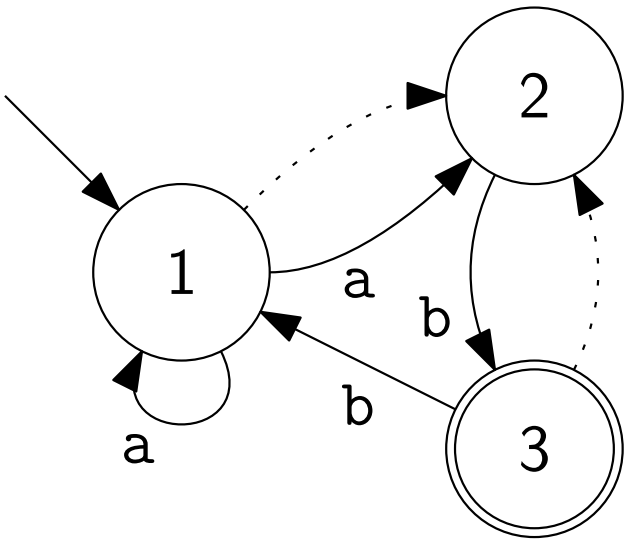
部分集合構成法

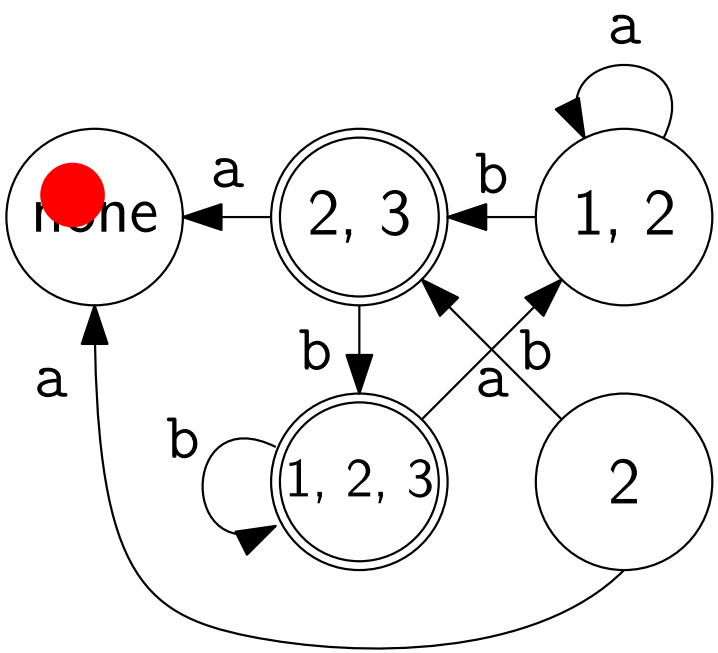




他の状態からの移動も同様に考える

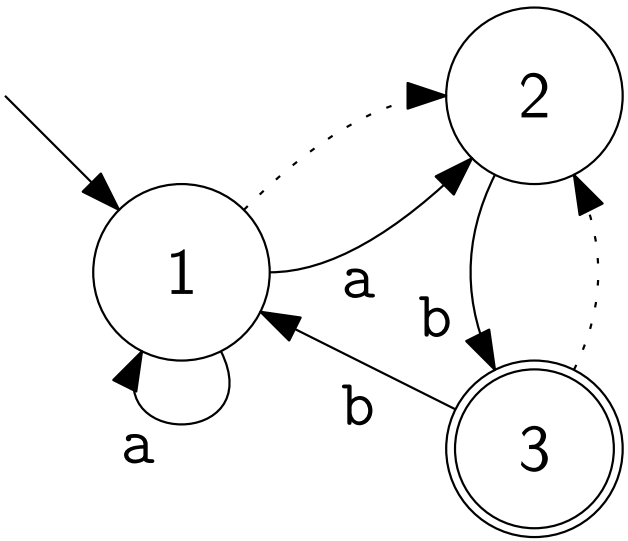
部分集合構成法

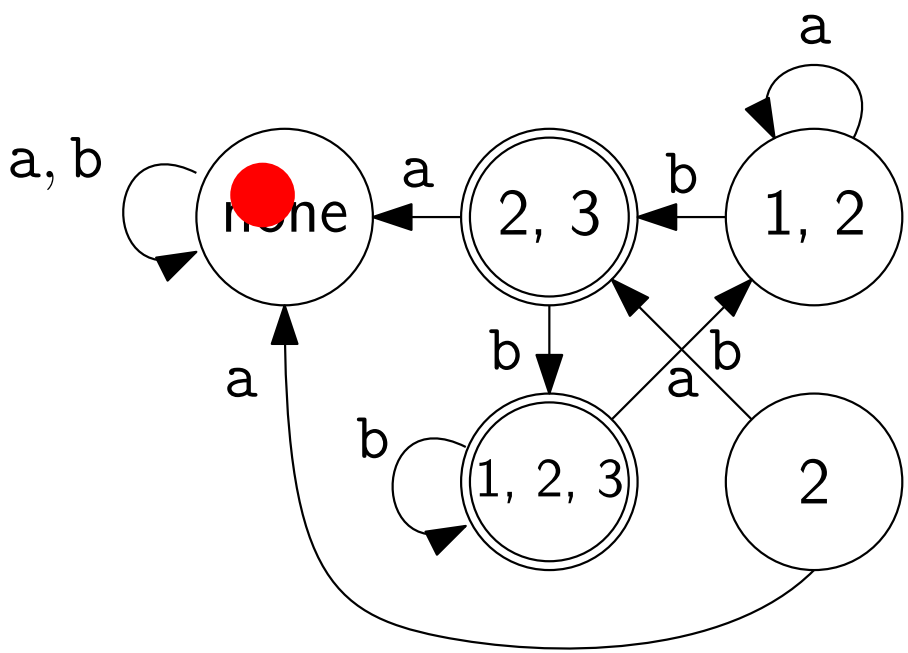




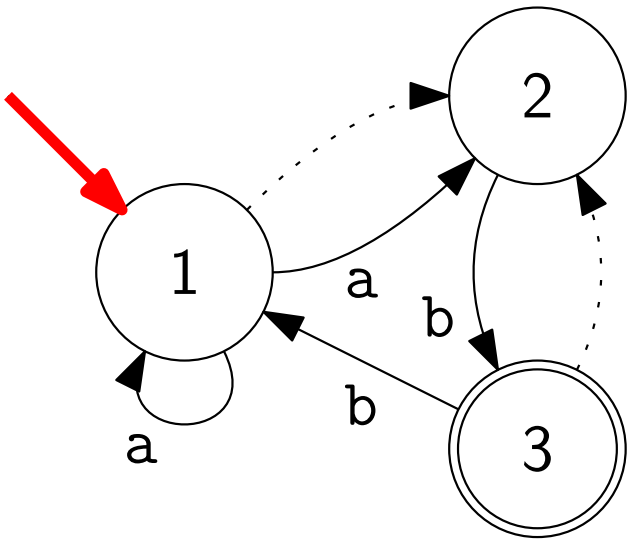
他の状態からの移動も同様に考える

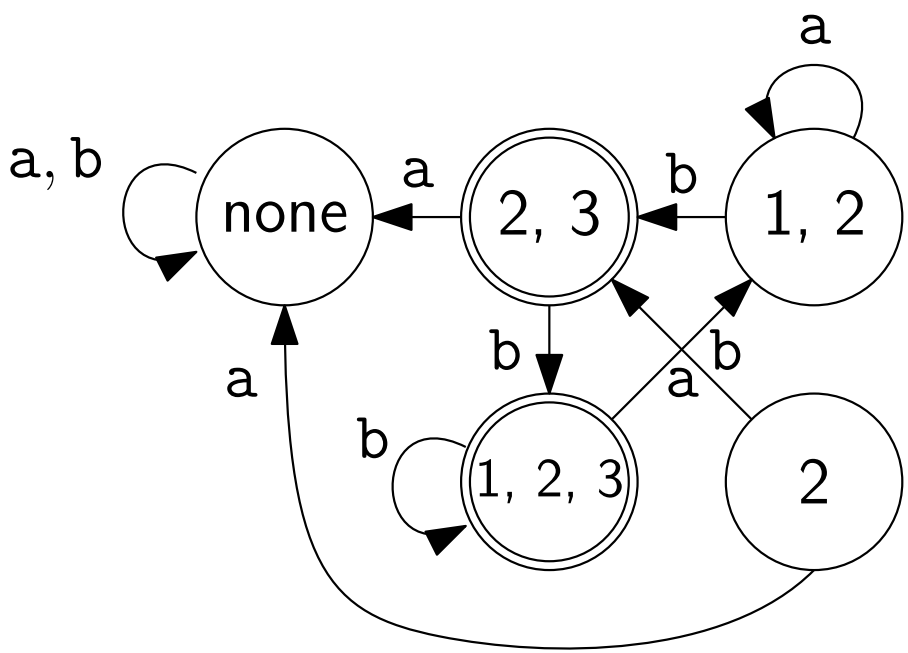
部分集合構成法





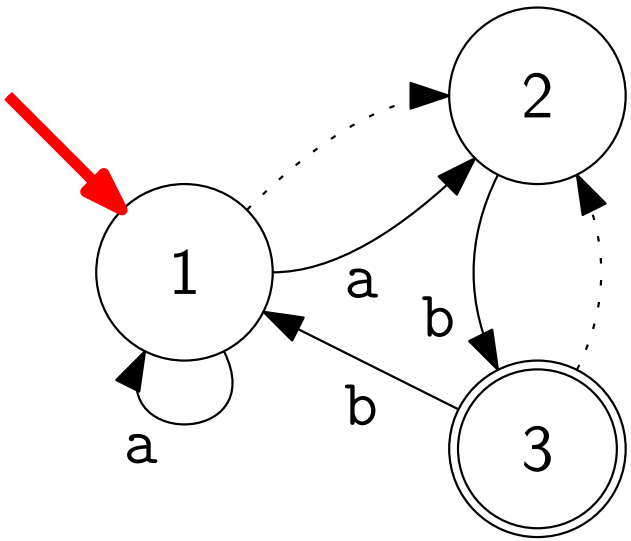
他の状態からの移動も同様に考える

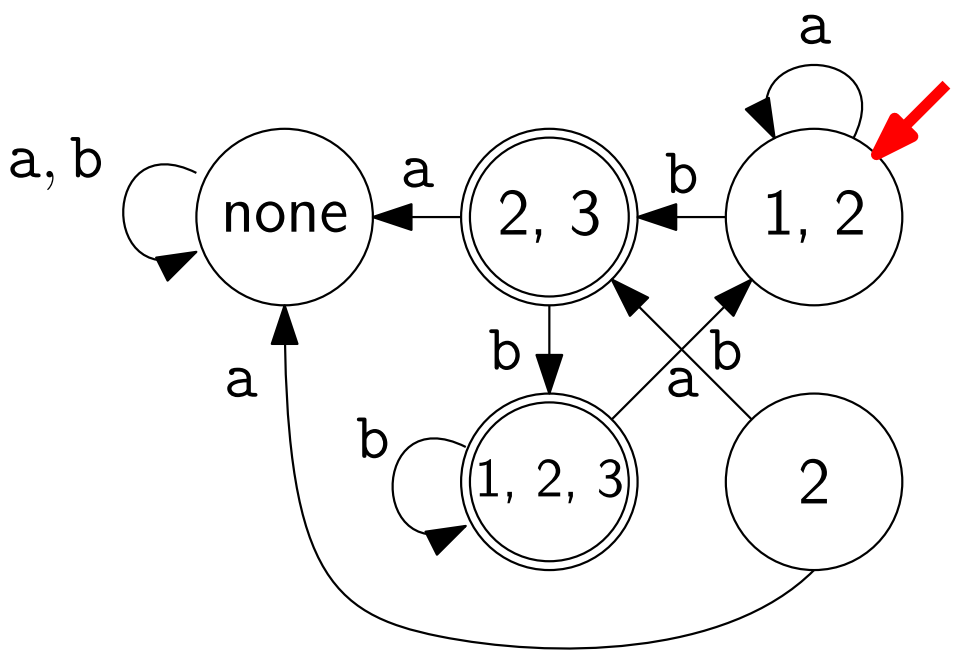




開始状態を定める

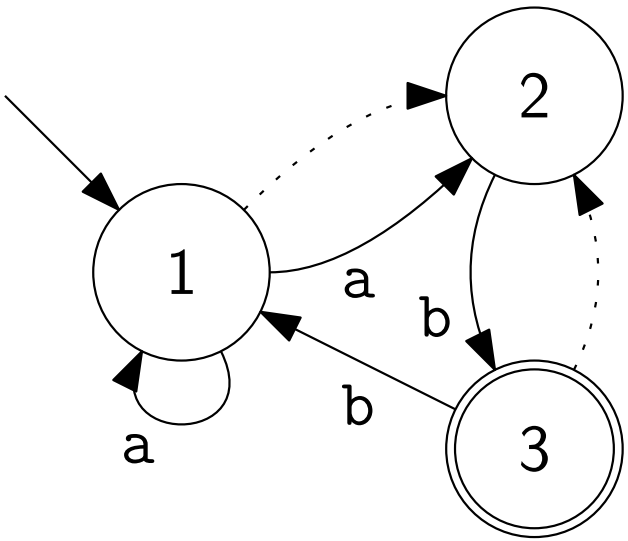
部分集合構成法

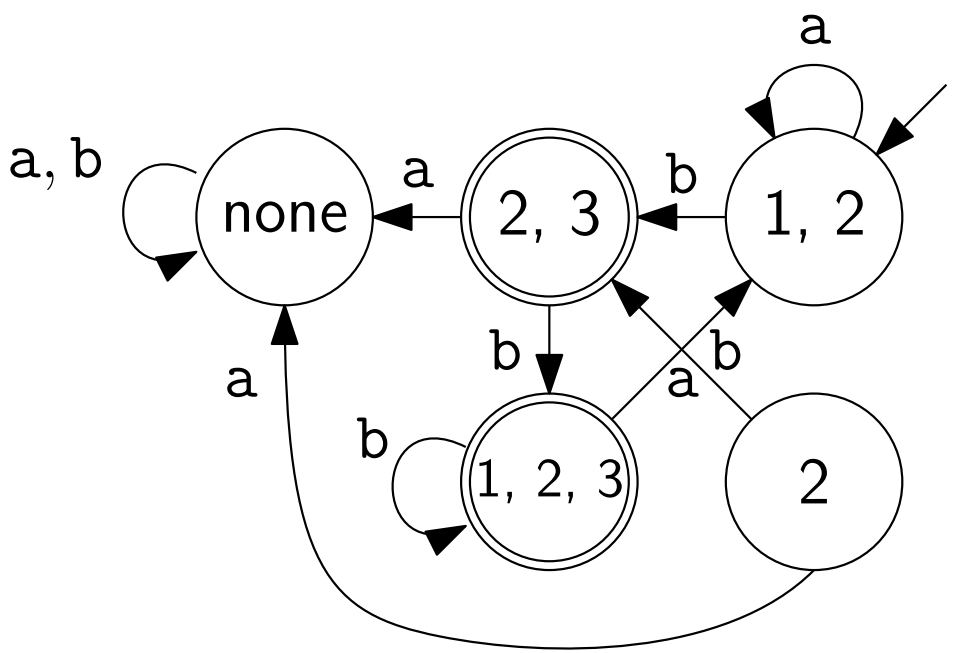




開始状態を定める

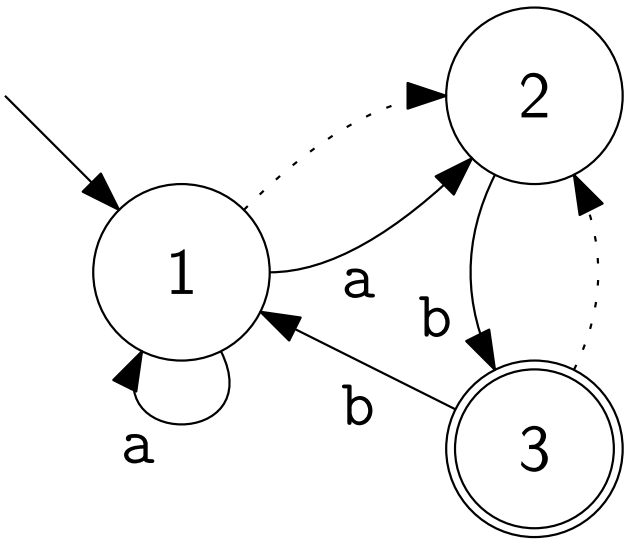
部分集合構成法

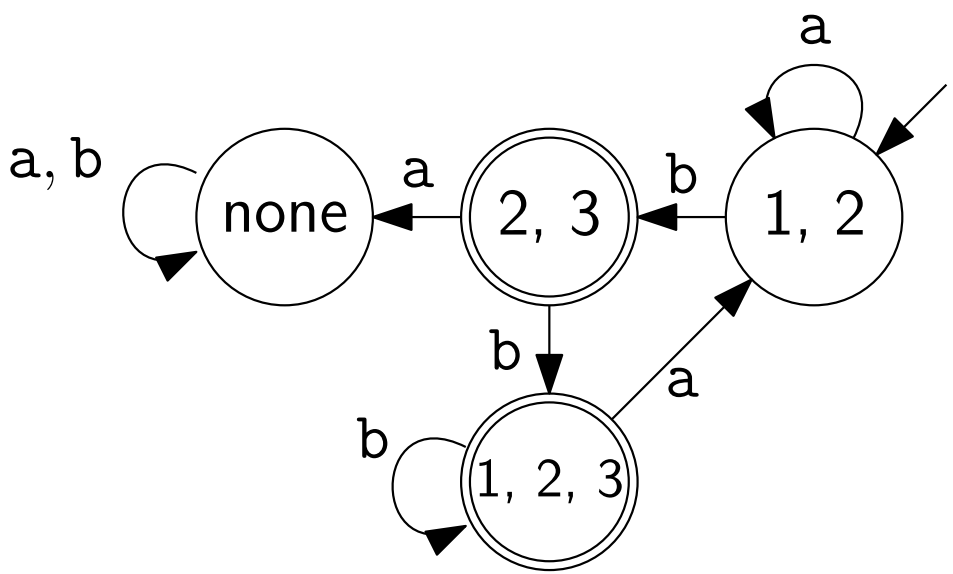




状態「{2}」に移動する状態がないので，状態「{2}」は不要

部分集合構成法





決定性有限オートマトンが完成

決定性と非決定性：一長一短

決定性有限オートマトン

- ○ 移動が分かりやすい
- ○ 実装しやすい
- × 設計しにくい
- × 状態数が大きくなりがち

非決定性有限オートマトン

- × 移動が分かりにくい
- × 実装しにくい
- ○ 設計しやすい
- ○ 状態数が小さくなりがち

部分集合構成法では，非決定性有限オートマトンの状態数が n のとき，
そこから作られる決定性有限オートマトンの状態数は $\leq 2^n$ で，
最悪の場合，2^n になりうる

```
>> simul = NFASimulation.new(nfa_design)
=> #<struct NFASimulation nfa_design=#<struct NFADesign start_
>> dfa_design = simul.to_dfa_design
=> #<struct DFADesign
     start_state=#<Set: {1}>,
     accept_states=[#<Set: {4}>, #<Set: {2, 4}>,
       #<Set: {4, 1}>, #<Set: {3, 2, 4, 5}>,
       #<Set: {2, 4, 5, 1}>, #<Set: {4, 1, 2}>],
     rulebook=#<struct DFARulebook rules=[
       #<FARule #<Set: {1}> --a--> #<Set: {2}>>,
       #<FARule #<Set: {1}> --b--> #<Set: {3, 5}>>,
       #<FARule #<Set: {2}> --a--> #<Set: {4}>>,
       #<FARule #<Set: {2}> --b--> #<Set: {5}>>,
       #<FARule #<Set: {3, 5}> --a--> #<Set: {4}>>,
       #<FARule #<Set: {3, 5}> --b--> #<Set: {2, 4}>>,
       #<FARule #<Set: {4}> --a--> #<Set: {1}>>,
       #<FARule #<Set: {4}> --b--> #<Set: {1}>>,
```

```
#<FARule #<Set: {5}> --a--> #<Set: {4}>>,
#<FARule #<Set: {5}> --b--> #<Set: {}>>,
#<FARule #<Set: {2, 4}> --a--> #<Set: {4, 1}>>,
#<FARule #<Set: {2, 4}> --b--> #<Set: {5, 1}>>,
#<FARule #<Set: {}> --a--> #<Set: {}>>,
#<FARule #<Set: {}> --b--> #<Set: {}>>,
#<FARule #<Set: {4, 1}> --a--> #<Set: {1, 2}>>,
#<FARule #<Set: {4, 1}> --b--> #<Set: {1, 3, 5}>>,
#<FARule #<Set: {5, 1}> --a--> #<Set: {4, 2}>>,
#<FARule #<Set: {5, 1}> --b--> #<Set: {3, 5}>>,
#<FARule #<Set: {1, 2}> --a--> #<Set: {2, 4}>>,
#<FARule #<Set: {1, 2}> --b--> #<Set: {3, 5}>>,
#<FARule #<Set: {1, 3, 5}> --a--> #<Set: {2, 4}>>,
#<FARule #<Set: {1, 3, 5}> --b--> #<Set: {3, 2, 4, 5}>>
#<FARule #<Set: {3, 2, 4, 5}> --a--> #<Set: {4, 1}>>,
#<FARule #<Set: {3, 2, 4, 5}> --b--> #<Set: {2, 4, 5, 1
```

```
#<FARule #<Set: {2, 4, 5, 1}> --a--> #<Set: {4, 1, 2}>>
#<FARule #<Set: {2, 4, 5, 1}> --b--> #<Set: {5, 1, 3}>>
#<FARule #<Set: {4, 1, 2}> --a--> #<Set: {1, 2, 4}>>,
#<FARule #<Set: {4, 1, 2}> --b--> #<Set: {1, 3, 5}>>]>>
```

目次

個人プロジェクト案の例

- クラス `DFA`，`NFA` にメソッドを追加する
  - 文字列を入力して，状態の移動を一文字ずつ追い，到達した状態を画面に逐一出力するメソッド `trace(string)` を実装する
- DFA 最小化アルゴリズムを実装する
  - Brzozowski のアルゴリズム
  - Hopcroft のアルゴリズム，Moore のアルゴリズム
- 「有限オートマトンの類似概念」について調べて，実装してみる
  - 例えば，Moore 機械や Mealy 機械なら，できそうか？
  - たぶん，確率的オートマトンや木オートマトンは難しそう？