MATHEMATICAL MACHINES

Computing

- Most computing consists in actions that modify data:
  - The data is textual
  - The actions are discrete: well-defined and single-step.

Computing

- Most computing consists in actions that modify data:
  - The data is textual
  - The actions are discrete: well-defined and single-step.
- The data is textual because discrete data has textual representation. (Though not all computing is discrete, eg Analog Computing is not.)

Acceptors

- ***What do algorithms do?***

Acceptors

- ***What do algorithms do?***
- Two main options: acceptors and transducers.
- An ***acceptor*** is an algorithm that takes a textual input
  (representing input data)
  and upon termination may or may not issue ***accept*** as output.

Acceptors

- ***What do algorithms do?***
- Two main options: acceptors and transducers.
- An ***acceptor*** is an algorithm that takes a textual input (representing input data) and upon termination may or may not issue ***accept*** as output.
- An acceptor that terminates for all input is a ***decider.***
- When a decider terminate for an input without accepting we say that it ***rejects*** the input.
- A decider is thus a solution for a decision problem.

Transducers

- A ***transducer*** is an algorithm that takes strings as input, and upon termination yields a string as output.

Transducers

- A ***transducer*** is an algorithm that takes strings as input, and upon termination yields a string as output.
- A transducer computes a ***partial-function*** (i.e. univalent mapping).

Transducers

- A ***transducer*** is an algorithm that takes strings as input, and upon termination yields a string as output.
- A transducer computes a ***partial-function*** (i.e. univalent mapping).
- An acceptor can be viewed as a transducer with ***accept*** as the only possible output;

  and a decider as a total transducer with ***accept*** and ***reject*** as the only possible outputs.

The simplest devices

- ***What is the simplest possible mathematical machine:***
  - Transducer, or acceptor?

The simplest devices

- ***What is the simplest possible mathematical machine:***
  - Transducer, or acceptor?
  - Fixed, or expandable external memory?

The simplest devices

- ***What is the simplest possible mathematical machine:***
  - ▸ Transducer, or acceptor?
  - ▸ Fixed, or expandable external memory?
  - ▸ Random-access, or sequential reading?

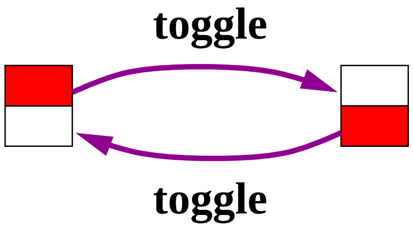
The simplest devices

- ***What is the simplest possible mathematical machine:***
    - Transducer, or acceptor?
    - Fixed, or expandable external memory?
    - Random-access, or sequential reading?
- We start with the ***automaton,***
  an acceptor with no external memory that reads its input sequentially!

The simplest devices

- ***What is the simplest possible mathematical machine:***
  - Transducer, or acceptor?
  - Fixed, or expandable external memory?
  - Random-access, or sequential reading?
- We start with the ***automaton,***
  an acceptor with no external memory that reads its input sequentially!
- This model captures the behavior of
  many familiar physical devices.
  Let’s look at a couple of very simple ones.

The electric switch

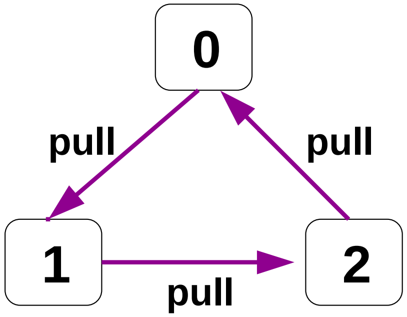


- The position of the switch is inverted
  after an odd number of toggles,
  and remains unchanged after an even number.

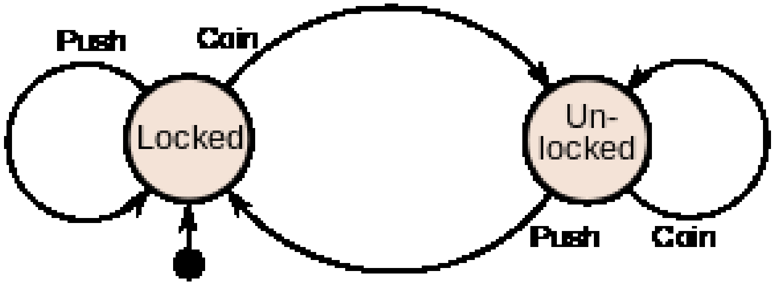
The ceiling fan

- A ceiling fan with manual cord-controlled:

  The speed is incremented (mod 2) with each pull.



The toll-turnstile



- The turnstile can be in one of two states: locked or unlocked.
- The action *insert token*
  changes the state *locked* into *unlocked*.
- The action *push and pass*
  changes the state *unlocked* into *locked*.

States

- A core concept of mathematical machines is the ***state.***
- E.g. a state of an elevator might consist of
  its position, motion (up, down, rest), upcoming destinations, time idle, etc.
- States are often labeled, for convenience, but don’t have to be.

States

- A core concept of mathematical machines is the ***state.***
- E.g. a state of an elevator might consist of
  its position, motion (up, down, rest), upcoming destinations, time idle, etc.
- States are often labeled, for convenience, but don't have to be.
- Given a practical problem, deciding what are the relevant "states"
  often requires careful analysis.
- But once a mathematical model is distilled,
  the ***states*** become an abstraction,
  which we can represent graphically, e.g. by a circle.

Transitions

- A ***transition-rule*** is a mapping from states to states. We label each transition-rule by an identifier.

Transitions

- A ***transition-rule***

  is a mapping from states to states. We label each transition-rule by an identifier.

- We focus for now on transitions that are ***functions***,

  i.e. univalent and total.

Transitions

- A $\boxed{\textit{transition-rule}}$

  is a mapping from states to states. We label each transition-rule by an identifier.

- We focus for now on transitions that are ***functions***,

  i.e. univalent and total.

- A pair of states related by a transition-rule `a` is an $\boxed{\textit{action}}$ of `a`.

Transitions

- A ***transition-rule*** is a mapping from states to states. We label each transition-rule by an identifier.
- We focus for now on transitions that are ***functions***, i.e. univalent and total.
- A pair of states related by a transition-rule `a` is an ***action*** of `a`.
- For the toll-turnstile and the stopwatch the transition-rules are determined by certain human actions.

Textual form of transitions

- Since all finite discrete structures have simple textual codes,
  we can assume that:

1. All input data is textual
2. Each transition is coded by a single reserved letter
3. The action of the transition labeled `a`
   is the reading (i.e. consumption) of `a`,
   much like the movement of a cursor.

$$\begin{tikzcd}
\underline{\texttt{a}}\texttt{bracadabra} \arrow[d, "\texttt{a}"] \\
\underline{\texttt{b}}\texttt{racadabra}
\end{tikzcd}$$

A transition system

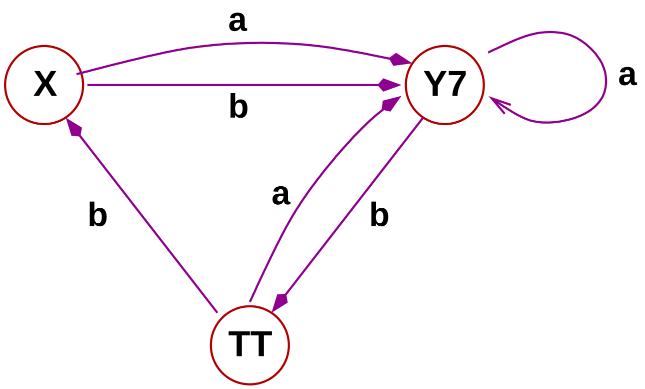
- A ***transition-system*** consists of a set of states and transition-rules over them.

A transition system

- A ***transition-system*** consists of a set of states
  and transition-rules over them.
- So a transition-system can be represented as a labeled di-graph:
  The nodes are the states,
  and the the actions are labeled edges.

A transition system

- A ***transition-system*** consists of a set of states and transition-rules over them.
- So a transition-system can be represented as a labeled di-graph: The nodes are the states, and the the actions are labeled edges.
- When all transition-rules are functions, there is exactly one edge for each state and action:

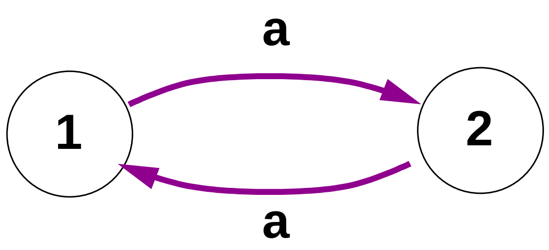


Example: Detecting an odd number of actions

- Consider the switch.

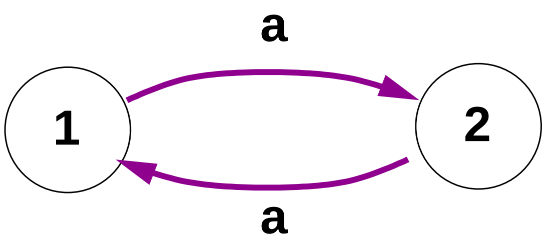
  We represent the transition “toggle” by the letter `a` ,

  and label the states as 1 and 2:



Example: Detecting an odd number of actions

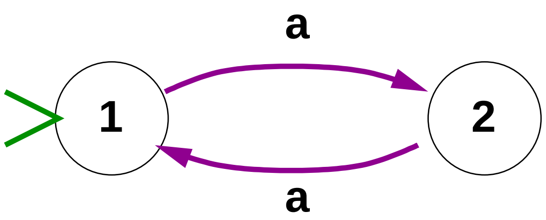
- Consider the switch.
  We represent the transition “toggle” by the letter `a` ,
  and label the states as 1 and 2:



- The device reads strings of `a`’s,
  and with each letter read it switch state.
- Reading odd number of `a` ’s leads to the opposite state.
- The physical nature of the toggle action is no longer present,
  and is indeed irrelevant.

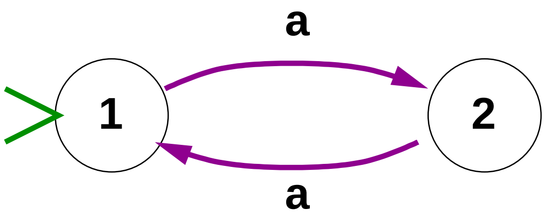
Start state and accepting states

- We intend to start at a particular state, so we single out one state as the ***initial*** (starting) state, indicated graphically by an incoming arrow.



Start state and accepting states

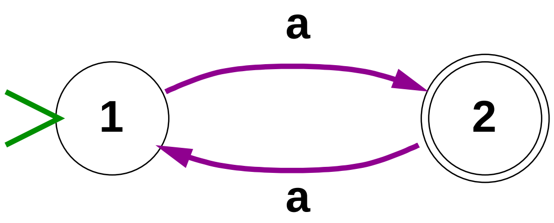
- We intend to start at a particular state,
  so we single out one state as the ***initial*** (starting) state,
  indicated graphically by an incoming arrow.



Where do the strings of length 1,3,... odd n lead?

Start state and accepting states

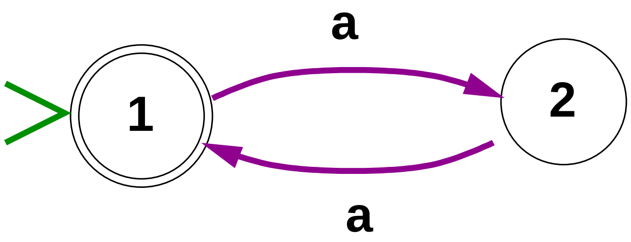
- We intend to start at a particular state,
  so we single out one state as the ***initial*** (starting) state,
  indicated graphically by an incoming arrow.



- The strings of odd length leads to state 2,
  so to accept just those strings we'd set 2
  as the unique accepting state.
- We do this graphically by doubling the contour of state 2.
- In general there can be several accepting states.

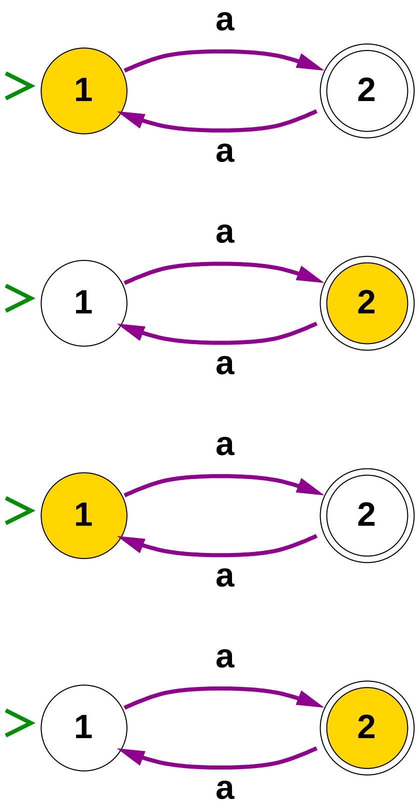
Initial state can be accepting

- It is possible that the initial state is accepting.
- To accept the strings of even length
  set **1** as the only accepting state:



The device in action

- Device accepting odd length:



READING

a

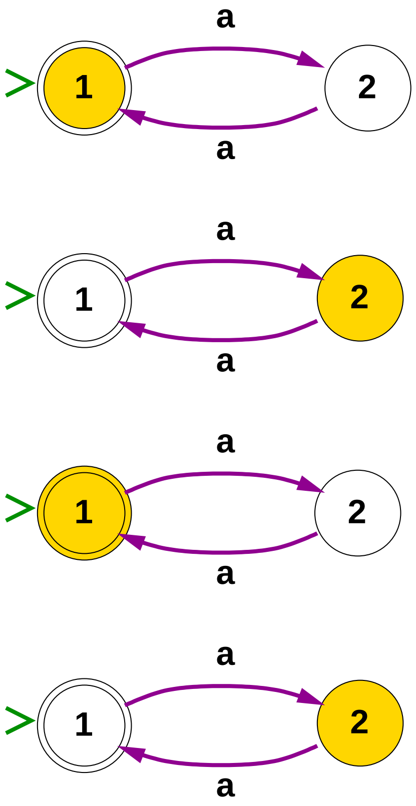
aa

aaa

string accepted IFF has odd #a
aaa accepted

The device in action

- Device accepting even length:



READING

a

aa

aaa

string accepted IFF has even #a
aaa not accepted

Definition of automata

- An **automaton,** aka ***deterministic finite automaton (DFA)*** consists of
  - An alphabet Σ.

Definition of automata

- An ***automaton,*** aka ***deterministic finite automaton (DFA)*** consists of
  - An alphabet Σ.
  - A non-empty finite set Q of objects called ***states***.
  - One state $s \in Q$ singled out as ***initial-state*** (or ***initial-state***).
  - A set $A \subseteq S$ of states singled out as ***accepting states***.

Definition of automata

- An **automaton,** aka ***deterministic finite automaton (DFA)*** consists of
  - An alphabet Σ .
  - A non-empty finite set Q of objects called **states**.
  - One state $s \in Q$ singled out as **initial-state** (or ***initial-state***).
  - A set $A \subseteq S$ of states singled out as **accepting states**.
  - A **transition function** $\delta : Q \times \Sigma \to Q$.
    Given state $q \in Q$ and input-symbol σ
    $\delta(q, \sigma)$ is the new (target) state.
- We also write $q \xrightarrow{\sigma} p$ for $\delta(q, \sigma) = p$.
  Note: p may be the same as q.

Comments on the definition

- Formally, M above is a tuple $(\Sigma, Q, s, A, \delta)$ of its components.

Comments on the definition

- Formally, M above is a tuple $(\Sigma, Q, s, A, \delta)$ of its components.
- M is ***over the alphabet*** Σ.
  We don't mention Σ when irrelevant or clear.

Comments on the definition

- Formally, M above is a tuple $(\Sigma, Q, s, A, \delta)$ of its components.
- M is ***over the alphabet*** Σ.
  We don't mention Σ when irrelevant or clear.
- *Automaton* is of Greek origin:

  auto = self, *matos* = move.

  Plural: *automata* or *automatons*. *Automat***a** is never singular.

Comments on the definition

- Formally, M above is a tuple $(\Sigma, Q, s, A, \delta)$ of its components.
- M is ***over the alphabet*** Σ.
  We don't mention Σ when irrelevant or clear.
- *Automaton* is of Greek origin:
  auto = self, *matos* = move.
  Plural: *automata* or *automatons*. *Automat***a** is never singular.
- Since automata play a central role,
  they've acquired over time several alternative names, in particular *deterministic finite automaton (DFA)*.which we'll frequently use.

Some practical applications of automata

Textual applications

- Pattern matching, search engines
- Lexical analysis for compilation
- Data compression
- Automatic translation

Some practical applications of automata

Software systems

- Cyber-security
- System planning
- Information streaming
- Bio-informatics

Some practical applications of automata

Hardware systems

- Circuit design
- Robotics

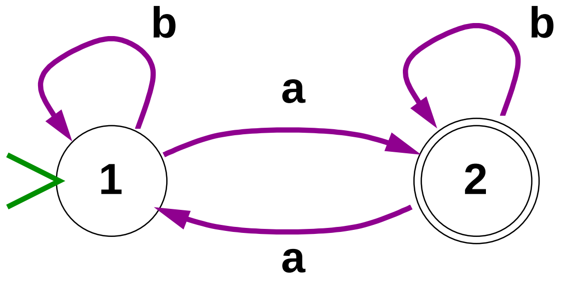
Some practical applications of automata

Verification

- System modeling
- Verification of communication protocols
- Verification of embedded systems
- Model checking

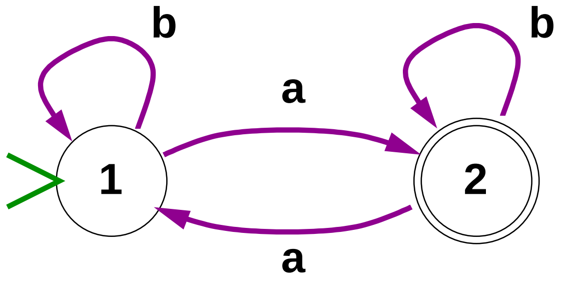
Example of a formal description

- Here's an automaton M over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ that accepts strings with an odd number of $\mathtt{a}$'s (and no others).



Example of a formal description

- Here's an automaton M over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ that accepts strings with an odd number of $\mathtt{a}$'s (and no others).



- Its formal definition: $M = (\Sigma, Q, s, A, \delta)$ where
  - ⋆ $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
  - ⋆ $Q = \{1, 2\}$
  - ⋆ $s = 1$
  - ⋆ $A = \{2\}$

Operational semantics: How automata function

- Intuitively, an automaton reads successive input symbols
  starting with the initial state, and
  updating the state according to the transition function δ.

Operational semantics: How automata function

- Intuitively, an automaton reads successive input symbols
  starting with the initial state, and
  updating the state according to the transition function δ.

- The steps of an automaton change just the state,
  and the implicit move to the next input symbol.

- Since the transition mapping of an automaton is a function,
  there is exactly one next-state for each symbol read.

Operational semantics: How automata function

- Intuitively, an automaton reads successive input symbols
  starting with the initial state, and
  updating the state according to the transition function δ.
- The steps of an automaton change just the state,
  and the implicit move to the next input symbol.
- Since the transition mapping of an automaton is a function,
  there is exactly one next-state for each symbol read.
- Computation terminates iff the end of the input string is reached.

Operational semantics: How automata function

- Intuitively, an automaton reads successive input symbols
  starting with the initial state, and
  updating the state according to the transition function δ.
- The steps of an automaton change just the state,
  and the implicit move to the next input symbol.
- Since the transition mapping of an automaton is a function,
  there is exactly one next-state for each symbol read.
- Computation terminates iff the end of the input string is reached.
- The essence of a DFA is in its being an ***online acceptor***.

Traces

- If $w = \sigma_1 \cdots \sigma_n$ then we write $q \xrightarrow{\sigma_1 \cdots \sigma_n} p$ to state that $q \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} r_2 \quad \cdots \quad r_{n-1} \xrightarrow{\sigma_n} p$ for some states $r_1, \ldots, r_{n-1}$.

Traces

- If $w = \sigma_1 \cdots \sigma_n$ then we write $q \xrightarrow{\sigma_1 \cdots \sigma_n} p$ to state that $q \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} r_2 \quad \cdots \quad r_{n-1} \xrightarrow{\sigma_n} p$ for some states $r_1, \ldots, r_{n-1}$.
- The sequence of states $q, \; r_1, \; r_2, \; \cdots \; r_{n-1}, \; p$ is a ***state-trace*** of the automaton.

Inductive definition of traces

- The ternary relation $q \xrightarrow{w} p$ can be defined inductively, by recurrence on w :
  - ► $q \xrightarrow{\varepsilon} q$
  - ► If $\delta(q,\sigma) = p$ that is $q \xrightarrow{\sigma u} r$, and $p \xrightarrow{u} r$ then $p \xrightarrow{\sigma} q$.

Inductive definition of traces

- The ternary relation $q \xrightarrow{w} p$ can be defined inductively,
  by recurrence on w :
  - $q \xrightarrow{\varepsilon} q$
  - If $\delta(q,\sigma) = p$ that is $q \xrightarrow{\sigma u} r$,
    and $p \xrightarrow{u} r$ then $p \xrightarrow{\sigma} q$.
- This definition invokes no auxiliary data
  that might be modified during execution.
- No mathematical machine we'll encounter (except NFAs)
  has such a definition:
  They all are based on a notion of ***configuration***,
  which combines the machine's states with modifiable data.

Accepted strings, recognized languages

- For $A \subseteq Q$ let's write $q \xrightarrow{w} A$

  when $q \xrightarrow{w} p$ for some $p \in A$.

- M $\boxed{\textit{accepts}}$ w when $s \xrightarrow{w} A$.

Accepted strings, recognized languages

- For $A \subseteq Q$ let's write $q \xrightarrow{w} A$
  when $q \xrightarrow{w} p$ for some $p \in A$.
- M $\boxed{\textit{accepts}}$ w when $s \xrightarrow{w} A$.
- The language $\boxed{\textit{recognized}}$ by M is

$$\begin{aligned} \mathcal{L}(M) &= \{w \in \Sigma^* \mid M \text{ accepts } w\} \\ &= \{w \in \Sigma^* \mid s \xrightarrow{w} A\} \end{aligned}$$

- We re-use here the notation $\mathcal{L}(\cdots)$ that we used for regular expressions.

Accepted strings, recognized languages

- For $A \subseteq Q$ let's write $q \xrightarrow{w} A$
  when $q \xrightarrow{w} p$ for some $p \in A$.
- M ***accepts*** w when $s \xrightarrow{w} A$.
- The language ***recognized*** by M is

$$\begin{aligned} \mathcal{L}(M) &= \{ w \in \Sigma^* \mid M \text{ accepts } w \} \\ &= \{ w \in \Sigma^* \mid s \xrightarrow{w} A \} \end{aligned}$$

- We re-use here the notation $\mathcal{L}(\cdots)$ that we used for regular expressions.
- Two automata are ***equivalent*** if they recognize the same language.

Automata are strictly regimented

1. Automata are acceptors: they produce no output.

Automata are strictly regimented

1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).

Automata are strictly regimented

1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.

Automata are strictly regimented

1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.
4. Scanning at a single point (i.e. computation is ***on-line***).

Automata are strictly regimented

1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.
4. Scanning at a single point (i.e. computation is ***on-line***).
5. Exactly one move exists for each state and symbol.

Automata are strictly regimented

1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.
4. Scanning at a single point (i.e. computation is ***on-line***).
5. Exactly one move exists for each state and symbol.
6. Computation stops when the input's end is reached.

Automata are strictly regimented

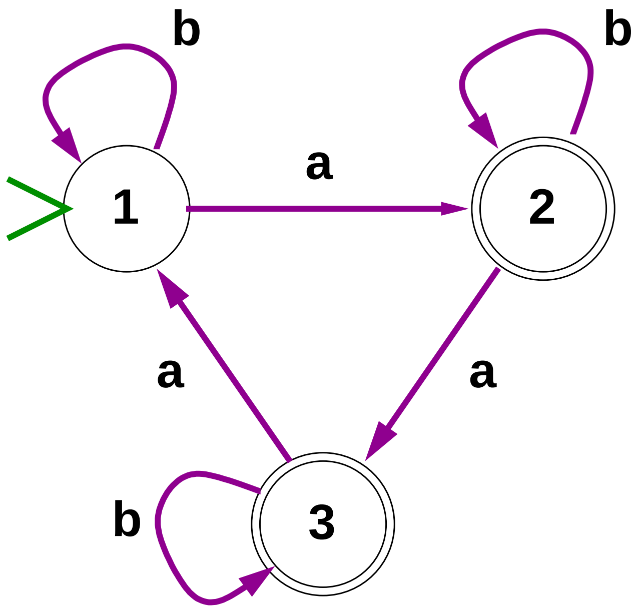
1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.
4. Scanning at a single point (i.e. computation is ***on-line***).
5. Exactly one move exists for each state and symbol.
6. Computation stops when the input’s end is reached.
7. No auxiliary memory or devices.

Automata are strictly regimented

Only two are crucial: violating them changes computing's nature:

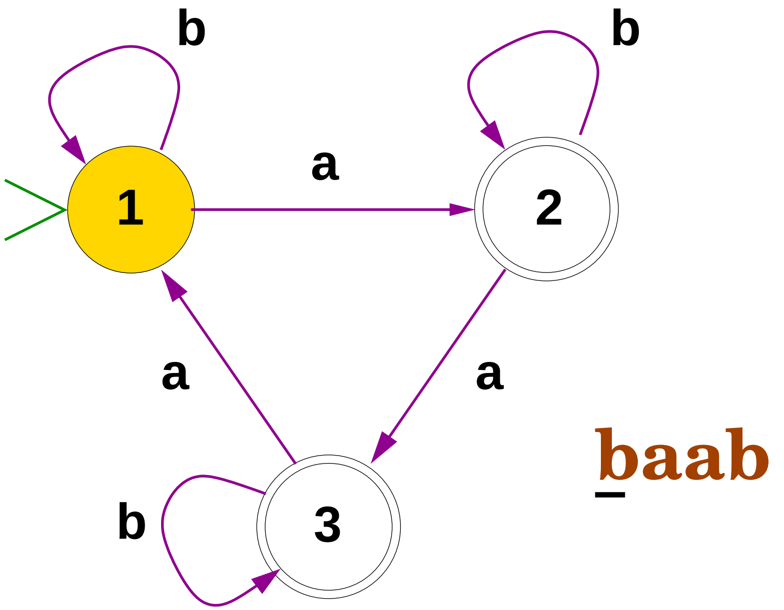
1. Automata are acceptors: they produce no output.
2. The input must be lexical (strings over a fixed alphabet).
3. Scanning forward: no backtracking or repositioning.
4. Scanning at a single point (i.e. computation is ***on-line***).
5. Exactly one move exists for each state and symbol.
6. Computation stops when the input's end is reached.
7. No auxiliary memory or devices.

Example: An automaton for Mod 3



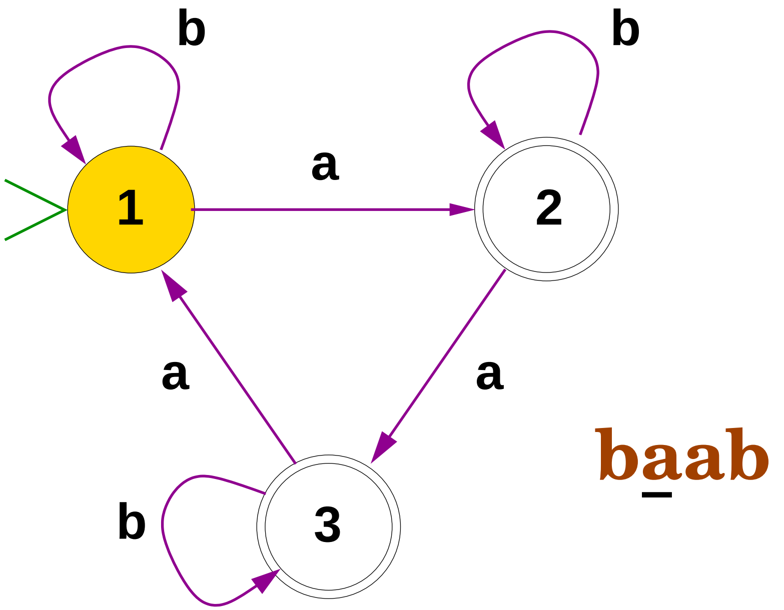
- $w \in \{\mathtt{a},\mathtt{b}\}^*$ accepted iff $\#_a(w) \neq 0 \pmod 3$

Example of an accepted string



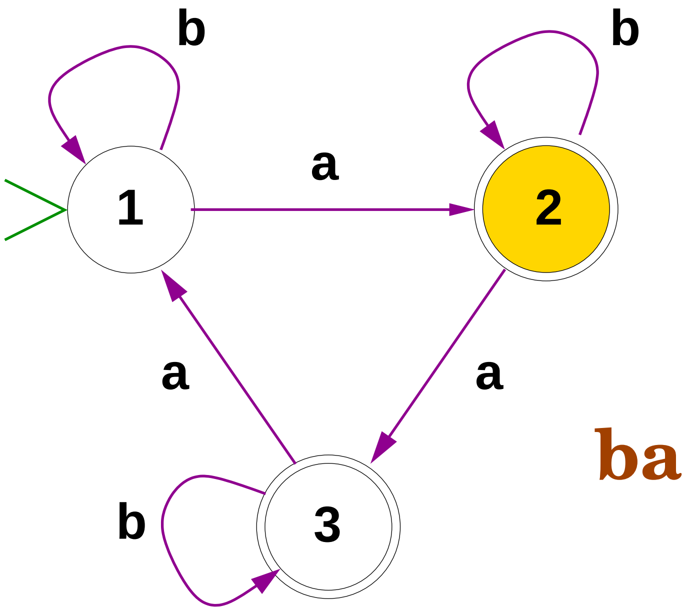
- State 1 (initial). Nothing read yet.

An accepted string



- Still state 1. Initial b read.

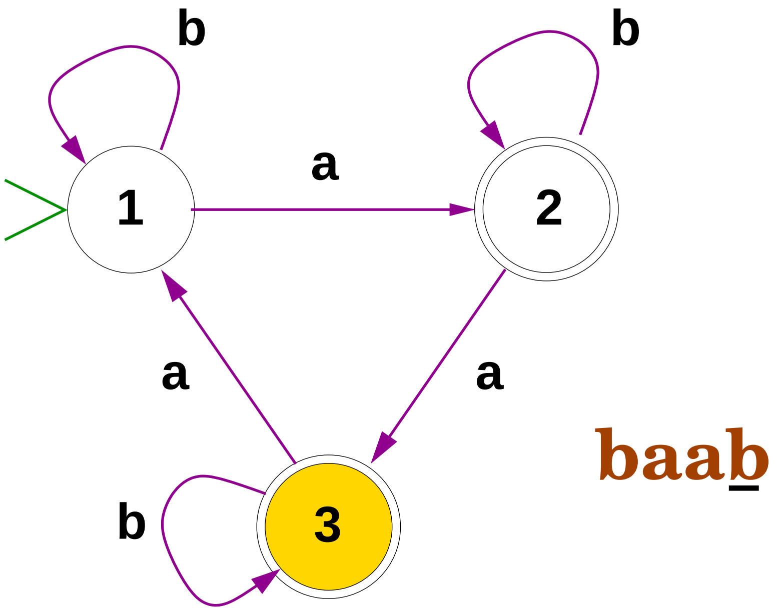
An accepted string



baab

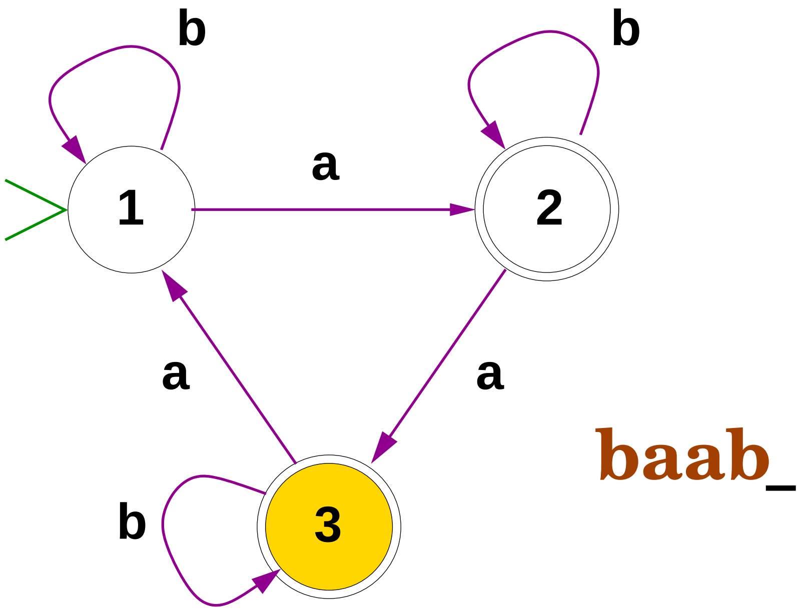
- Read `ba`, state 2.

An accepted string



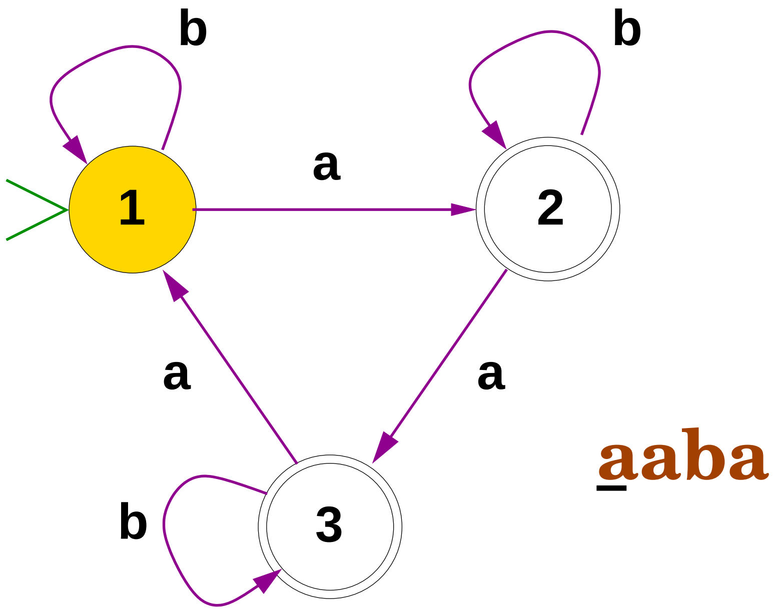
- Read baa, state 3.

An accepted string



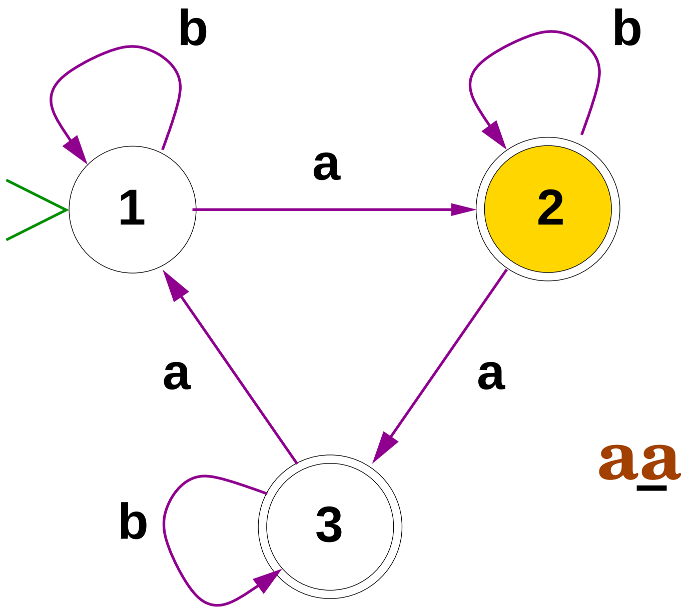
- Finished reading *baab*, state 3, accepted.

A non-accepted string



- State 1 (initial). Nothing read yet.

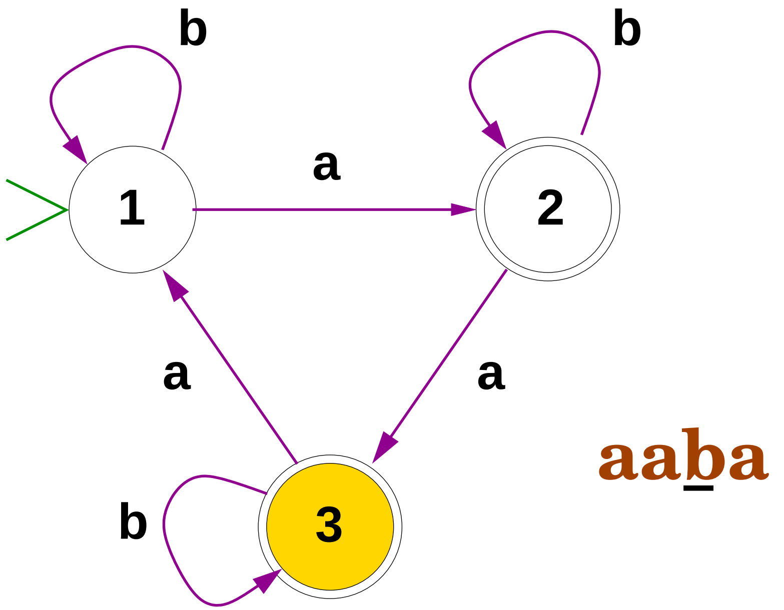
A non-accepted string



$\mathbf{a\underline{a}ba}$

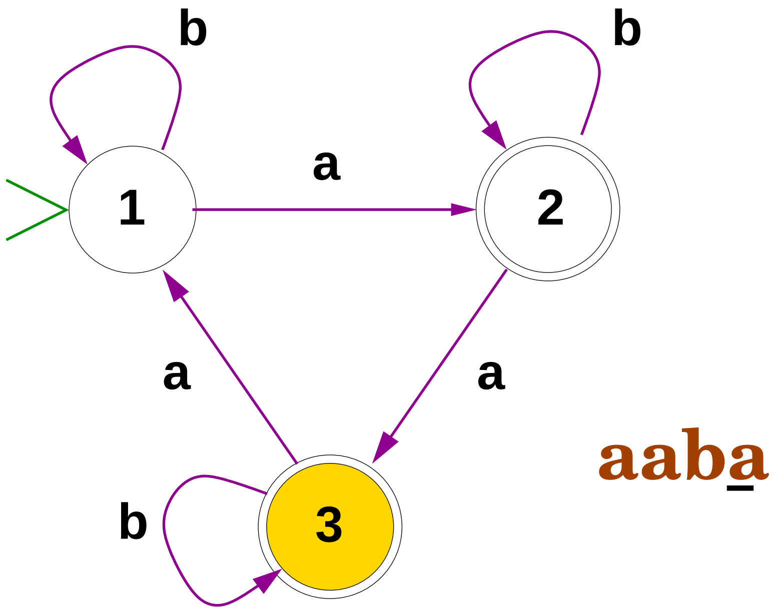
- Read a, State 2.

A non-accepted string



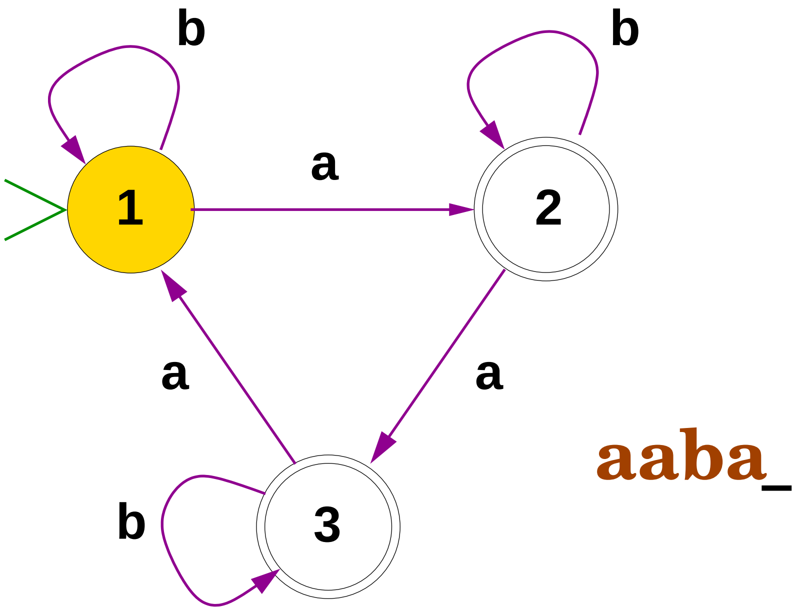
- Read `aa`, state 3.

A non-accepted string



- Read aab, state 3.

A non-accepted string



- Finished reading aaba, state 1, not accepted.

A computation trace

- For our example above, the computation for the string **`baab`** is

  $$1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} 2 \xrightarrow{\texttt{a}} 3 \xrightarrow{\texttt{b}} 3.$$

  Abbreviated notation: $1 \xrightarrow{\texttt{baab}} 3$

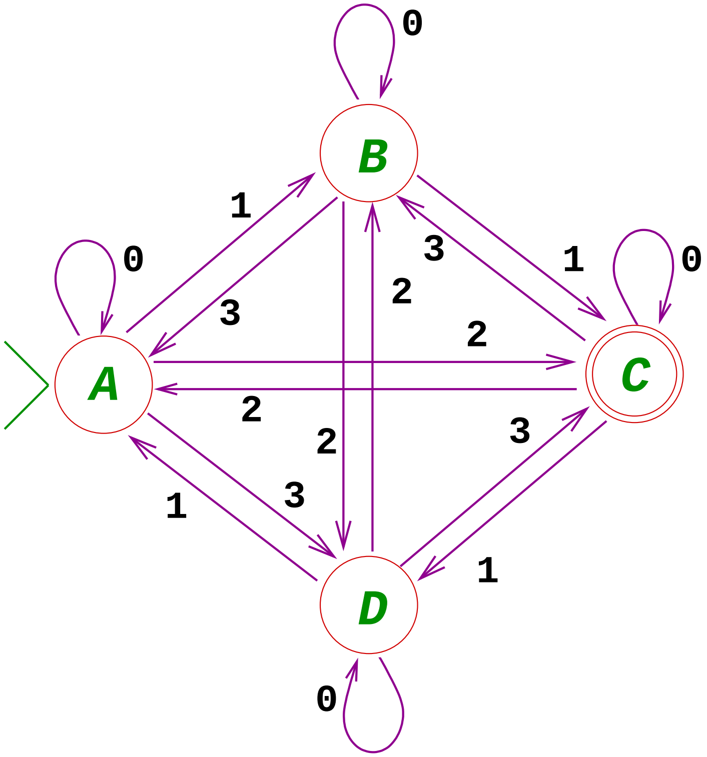
- The computation for the string **`aaba`** is

  $$1 \xrightarrow{\texttt{a}} 2 \xrightarrow{\texttt{a}} 3 \xrightarrow{\texttt{b}} 3 \xrightarrow{\texttt{a}} 1.$$

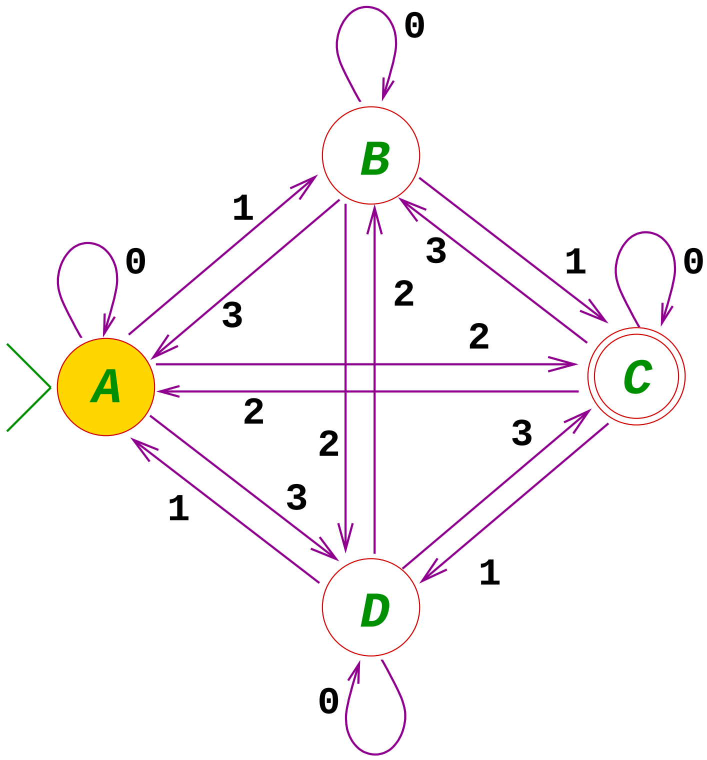
  Abbreviated notation: $1 \xrightarrow{\texttt{aaba}} 3$

Example: Addition mod 4

- The following automaton is over the alphabet {0, 1, 2, 3}
- It accept a string of digits iff they add up to 2 modulo 4.

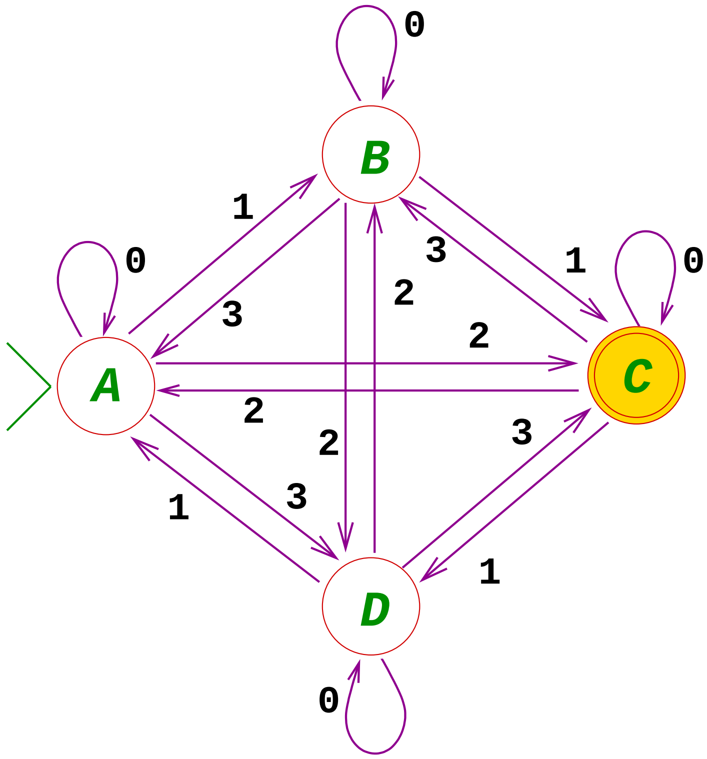


- Reading input 21032 from initial state A:



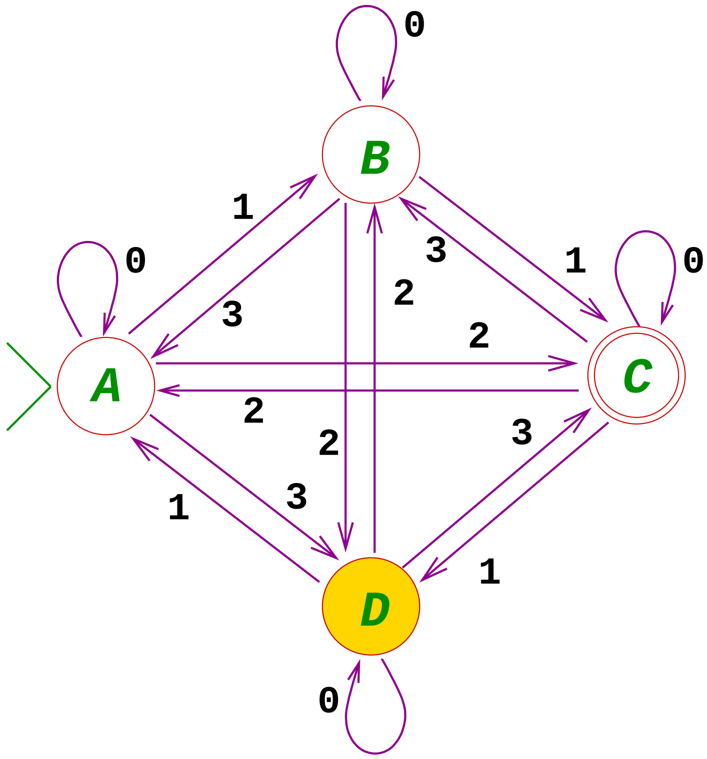
A 21032

- Reads remaining string 1032:



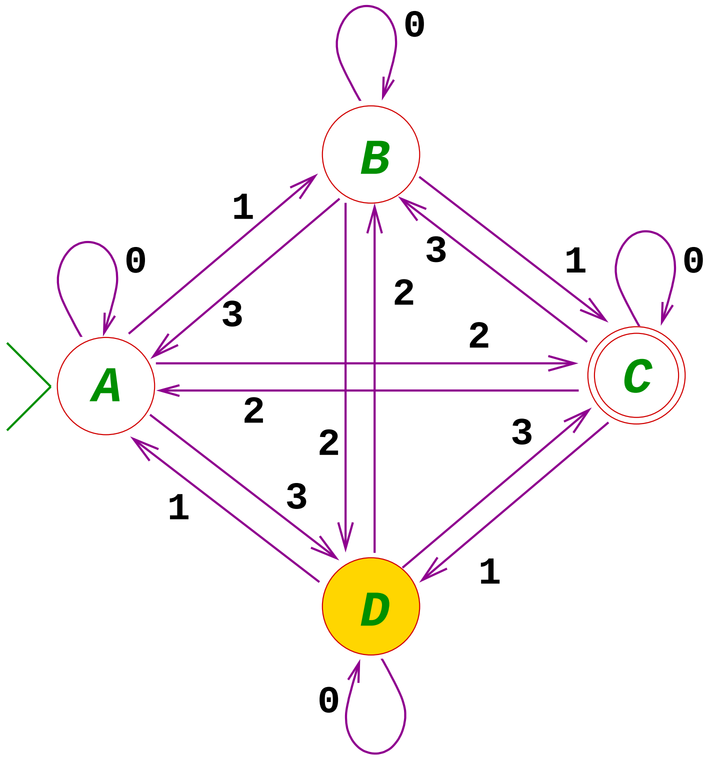
C **1032**

- Reads remaining string 032:



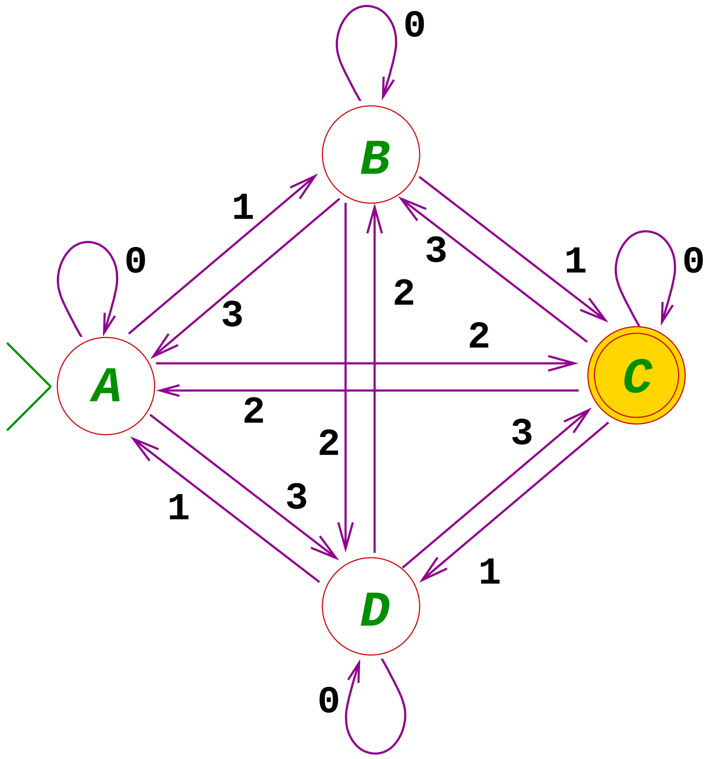
D 032

- Reads remainder 32:



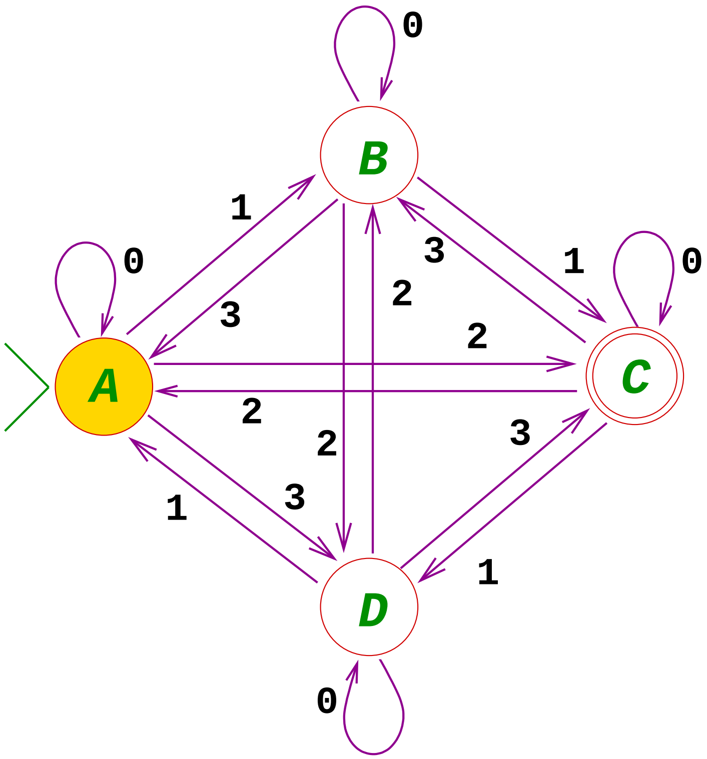
D 32

- Reads remainder 2:



C 2

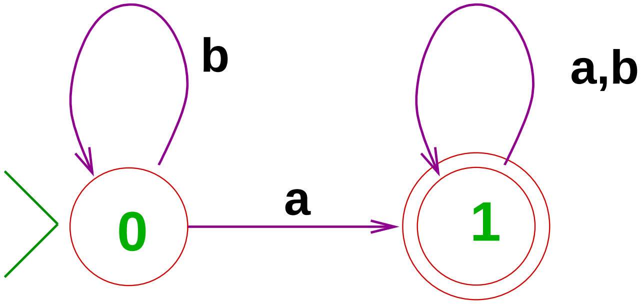
- Reads remainder ε (empty string):



A ε

- Ends reading. A not an accept-state, 21032 not accepted.

Additional examples

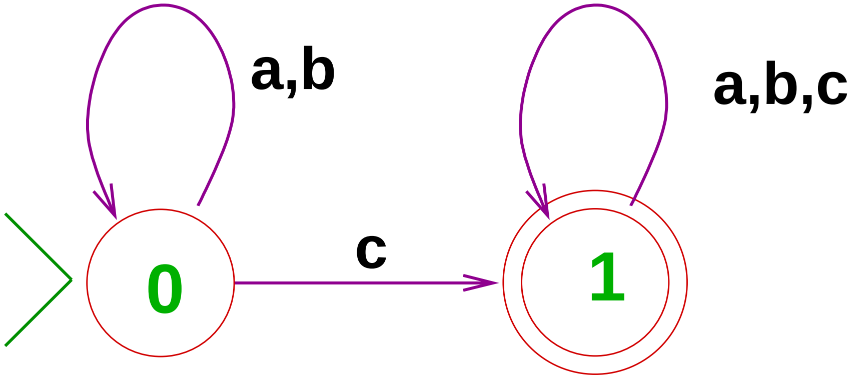


$$0 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} 1$$

$$0 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{b}} 0 \xrightarrow{\texttt{b}} 0$$

What is the language recognized?

Three letter example

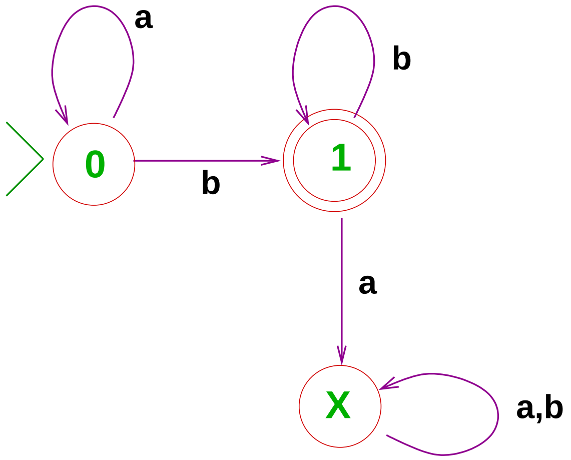


$$0 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{b}} O \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{c}} 1 \xrightarrow{\texttt{b}} 1$$

$$0 \xrightarrow{\texttt{c}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} 1$$

What are the language accepted?

An automaton with a sink



$0 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} 1$

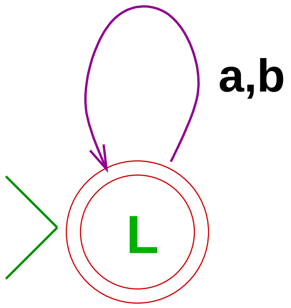
$0 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} X \xrightarrow{\texttt{b}} X \xrightarrow{\texttt{a}} X$

Note: Every state has exactly one arrow for every $\sigma \in \Sigma$.

- A ***sink*** is a non-accepting state with all outgoing transitions pointing to itself.

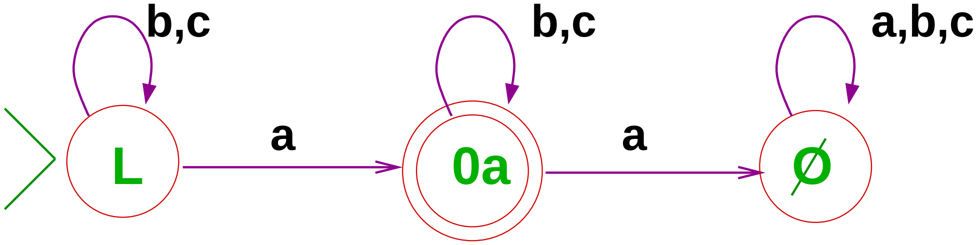
Example

Here is a trivial automaton with a single state:



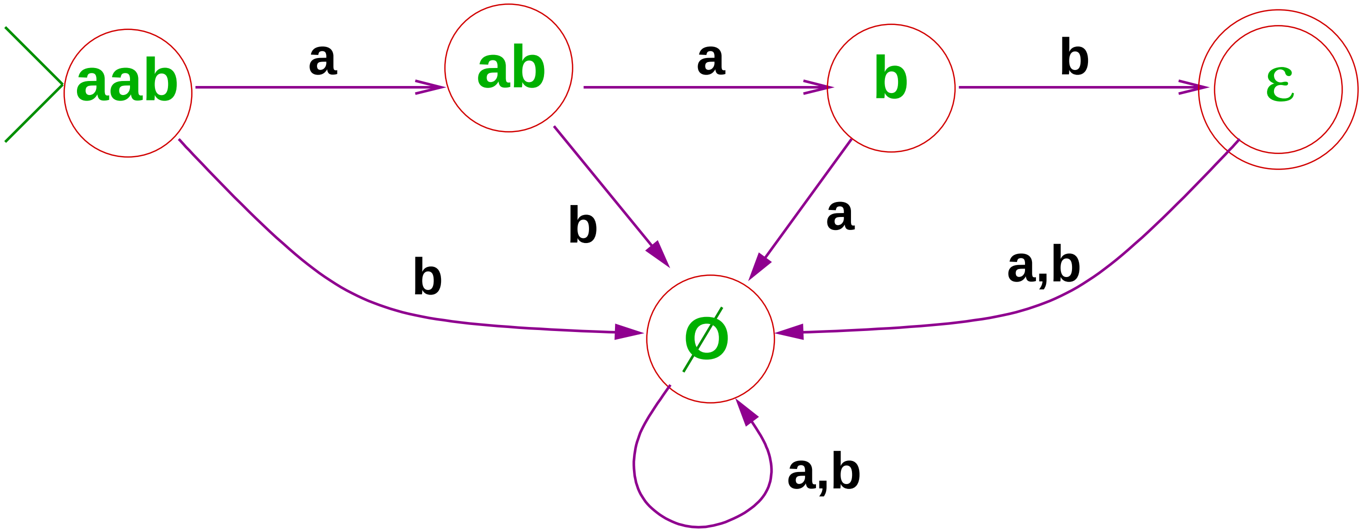
What strings are accepted?

Example



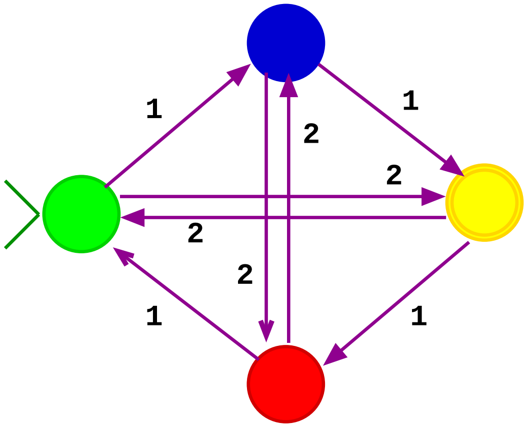
accepts the strings with exactly one a , and no other.

Example

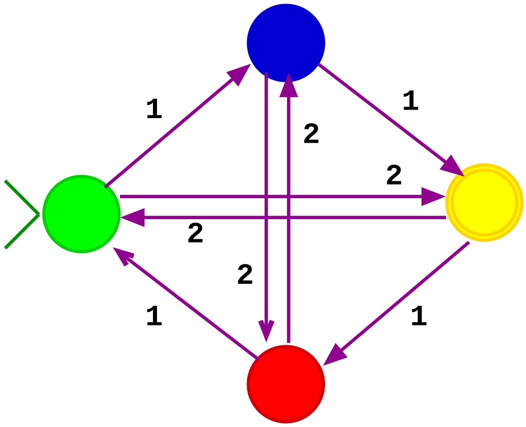


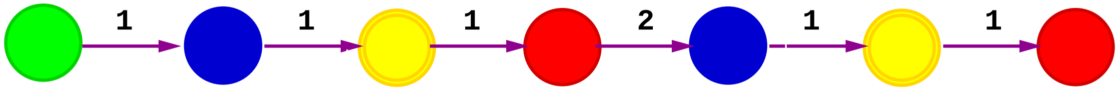
accepts the string `aab` and no other.

AUTOMATA ARE REPETITIVE

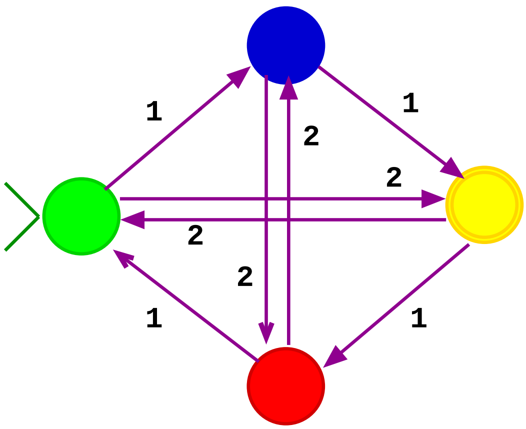


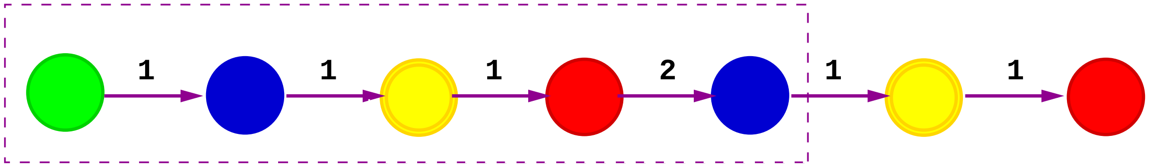
- Here's an automaton that accepts a string $w \in \{1,2\}^*$ iff the sum of the digits in w is $2 \mod (4)$.



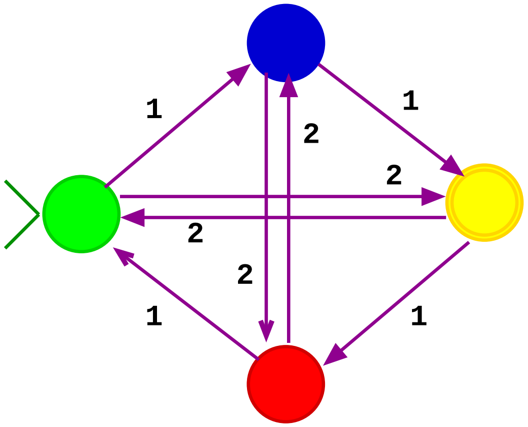


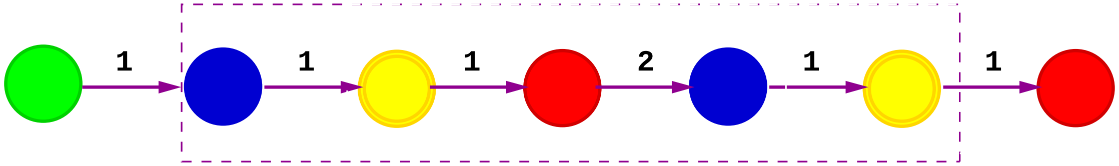
- This is its trace for input 111212.
  The input has 6 symbols, so the trace lists 7 states.



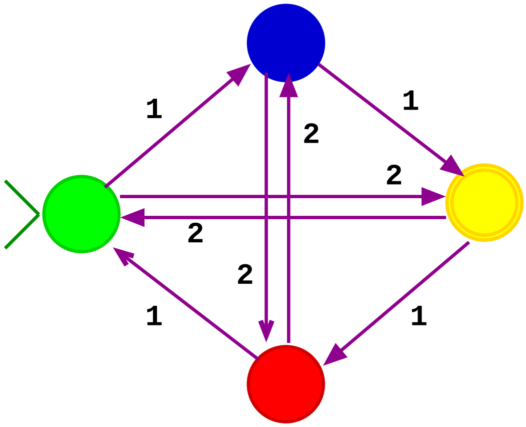


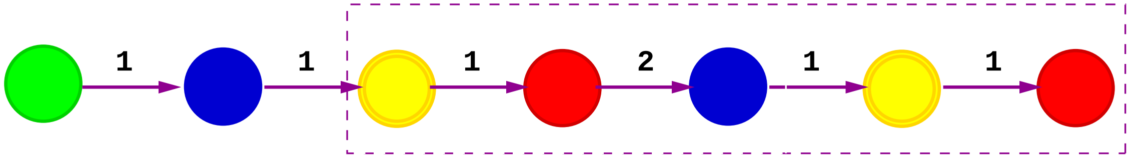
- Looking at the first 5 of the 7, we must have a state repeating, because there are only 4 states.





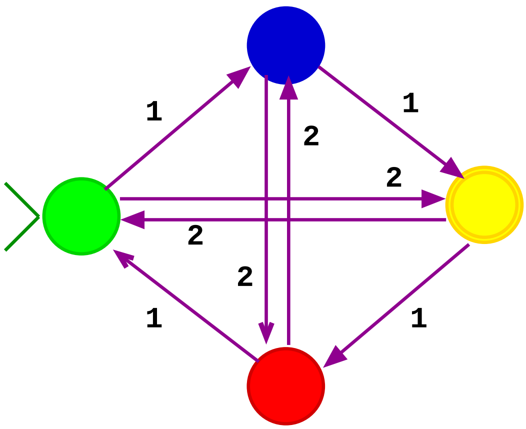
The same happens for the next stretch of 5 states (i.e. 4 input symbols)

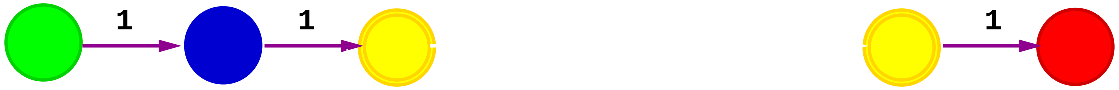




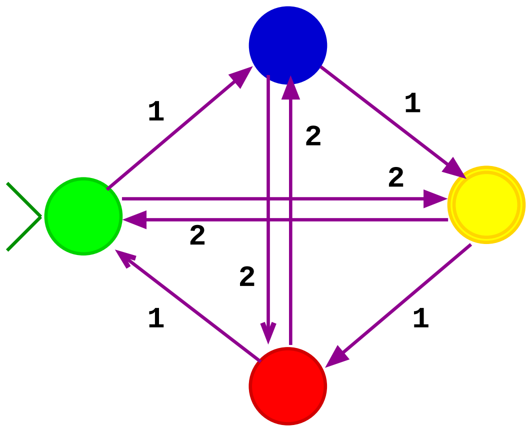
And the next one.

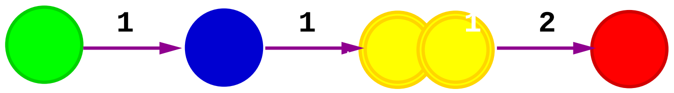
No matter which window of 5 states we take there will be a state repeating!





We can short-circuit the steps from the yellow state to itself, and the result will still be a legit trace, but for 112.



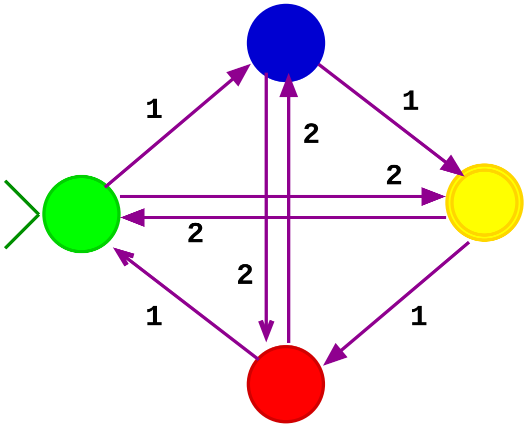


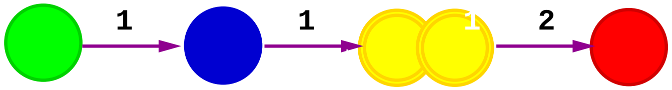
We can short-circuit the steps from the yellow state to itself, and the result will still be a legit trace, but for 112.

Shortcuts in traces

- We observed:

  Let M *be a* k *-state DFA.*
  If $q \xrightarrow{u} p$ *and* $|u| \geqslant k$ *then*
  $q \xrightarrow{u'} p$ *where* u' *is* u *with some*
  substring $y \neq \varepsilon$ *clipped off, i.e. removed.*



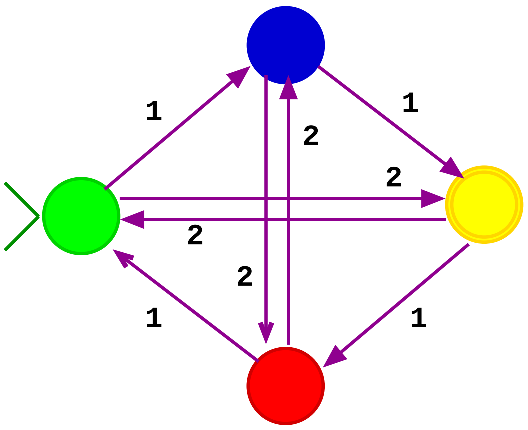


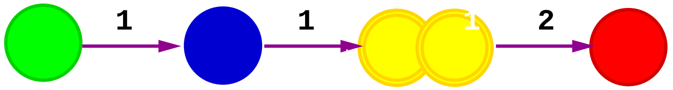
with $|u| \geqslant k$.

Shortcuts in traces

- We observed:

  Let M *be a* k *-state DFA.*
  If $q \xrightarrow{u} p$ *and* $|u| \geqslant k$ *then*
  $q \xrightarrow{u'} p$ *where* u' *is* u *with some*
  substring $y \neq \varepsilon$ *clipped off, i.e. removed.*





- Suppose we have

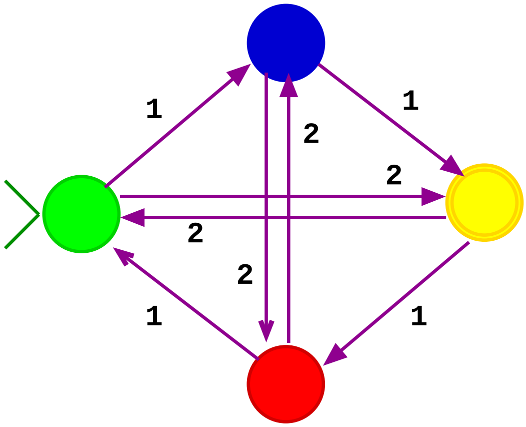
$$s \xrightarrow{w_0} p \xrightarrow{u} q \xrightarrow{w_1} A$$

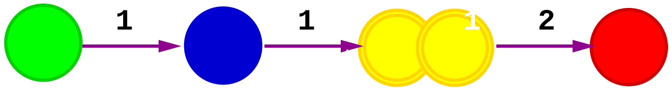
with $|u| \geqslant k$.

Shortcuts in traces

- We observed:

  Let M *be a* k *-state DFA.*
  If $q \xrightarrow{u} p$ *and* $|u| \geqslant k$ *then*
  $q \xrightarrow{u'} p$ *where* u' *is* u *with some*
  substring $y \neq \varepsilon$ *clipped off, i.e. removed.*





- Suppose we have

  $$s \xrightarrow{w_0} p \xrightarrow{u} q \xrightarrow{w_1} A$$

  with $|u| \geqslant k$.

  Then

  $$s \xrightarrow{w_0} p \xrightarrow{u'} q \xrightarrow{w_1} A$$

The Clipping Theorem

- **Theorem.** *If a k -state DFA accepts a string w , and u is a substring of w of length $\geqslant k$, then u has a substring $y \neq \varepsilon$ such that w with y removed is also accepted.*

The Clipping Theorem

- **Theorem.** *If a* k *-state DFA accepts a string* w *,*
  and u *is a substring of* w *of length* $\geqslant k$*,*
  then u *has a substring* $y \neq \varepsilon$ *such that*
  w *with* y *removed is also accepted.*

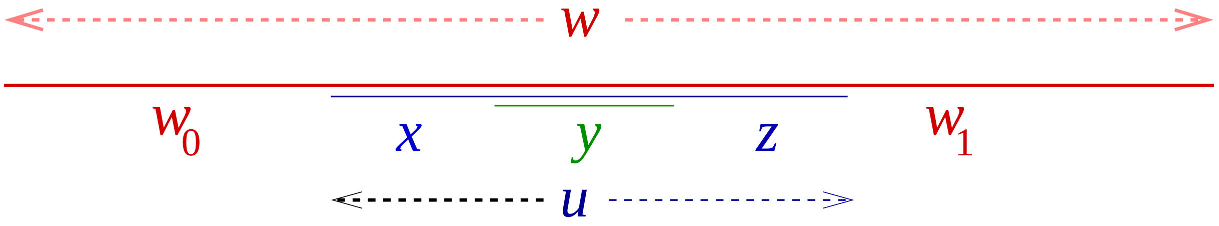
- That is, if M accepts $w_0 \cdot u \cdot w_1$, where $|u| \geqslant k$,
  then there is a split $u = x \cdot y \cdot z$, with $y \neq \varepsilon$,
  such that $w' = w_0 \cdot x \cdot z \cdot w_1$ is also accepted.

The Clipping Theorem

- **Theorem.** *If a k -state DFA accepts a string w , and u is a substring of w of length $\geqslant k$, then u has a substring $y \neq \varepsilon$ such that w with y removed is also accepted.*
- That is, if M accepts $w_0 \cdot u \cdot w_1$, where $|u| \geqslant k$, then there is a split $u = x \cdot y \cdot z$, with $y \neq \varepsilon$, such that $w' = w_0 \cdot x \cdot z \cdot w_1$ is also accepted.
- We call u the ***critical*** substring, the particular occurrence of substring y the ***clipped*** substring, and w' the ***reduced*** string.

The Clipping Theorem

- **Theorem.** *If a k -state DFA accepts a string w , and u is a substring of w of length $\geqslant k$, then u has a substring $y \neq \varepsilon$ such that w with y removed is also accepted.*
- That is, if M accepts $w_0 \cdot u \cdot w_1$, where $|u| \geqslant k$, then there is a split $u = x \cdot y \cdot z$, with $y \neq \varepsilon$, such that $w' = w_0 \cdot x \cdot z \cdot w_1$ is also accepted.
- We call u the ***critical*** substring, the particular occurrence of substring y the ***clipped*** substring, and w' the ***reduced*** string.



An application: the shortest string accepted

- If M is a 10 state automaton that accepts some string. What is the length ℓ of the ***shortest*** string accepted?

  1. $\ell \in [30..100]$
  2. $\ell \in [10..25]$
  3. $\ell \in [0..9]$
  4. Can’t tell, could be anything.

An application: the shortest string accepted

- If M is a 10 state automaton that accepts some string. What is the length ℓ of the ***shortest*** string accepted?
- **Theorem.** *If a k-state automaton M accepts some string, then it accepts a string of length $< k$.*

An application: the shortest string accepted

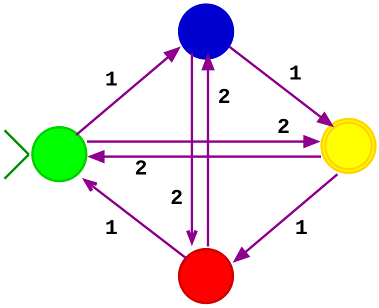
- If M is a 10 state automaton that accepts some string. What is the length ℓ of the ***shortest*** string accepted?
- **Theorem.** *If a k-state automaton M accepts some string, then it accepts a string of length $< k$.*
- **Proof:** Let w be a shortest string accepted by M.
  If $|w| \geqslant k$ then we invoke the Clipping Theorem,
  with w itself for u,
  and obtain a $w' \in L$ shorter than w.
  This contradicts the assumed minimality of $|w|$.

An application: the shortest string accepted

- If M is a 10 state automaton that accepts some string. What is the length ℓ of the ***shortest*** string accepted?
- **Theorem.** *If a k-state automaton M accepts some string, then it accepts a string of length $< k$.*
- **Proof:** Let w be a shortest string accepted by M.

  If $|w| \geqslant k$ then we invoke the Clipping Theorem,
  with w itself for u,
  and obtain a $w' \in L$ shorter than w.

  This contradicts the assumed minimality of $|w|$.
- Example: What is the shortest string accepted by



The dual question

- I want a DFA that accepts exactly the strings of length $\geqslant 100$.
- What's the smallest number ℓ of states I need?

1. $\ell \in [1..9]$
2. $\ell \in [10..99]$
3. $\ell \in [100..999]$
4. Can't tell, could be anything.

The dual question

- I want a DFA that accepts exactly the strings of length $\geqslant 100$.
- What's the smallest number ℓ of states I need?
  1. $\ell \in [1..9]$
  2. $\ell \in [10..99]$
  3. $\ell \in [100..999]$
  4. Can't tell, could be anything.
- Answer: 101:

  A DFA with 100 states will accept some string of length < 100.

On not being an insect

- How do you tell that the critter on your desk is not an insect?

On not being an insect

- How do you tell that the critter on your desk is not an insect?
- Check that it violates some property of insects, e.g. it has eight rather than six legs.
- How do you tell that a given language L is not recognized by any automaton?
- Refer to a property that all recognized languages have, but L does not.

On not being an insect

- How do you tell that the critter on your desk is not an insect?
- Check that it violates some property of insects, e.g. it has eight rather than six legs.
- How do you tell that a given language L is not recognized by any automaton?
- Refer to a property that all recognized languages have, but L does not.

The Clipping Property

- The Clipping Theorem says that

  Every language L recognized by a DFA has the following **Clipping Property:**

  ⋆ There is a k (the number of states in an acceptor for L),

  ⋆ so that for every $w \in L$

  ⋆ if u is a substring of w of length $\geqslant k$,

  ⋆ then it has a “clippable” substring $y \neq \varepsilon$:
  removing y from w yields a string in L .

The Clipping Property

- The Clipping Theorem says that

  Every language L recognized by a DFA has the following **_Clipping Property:_**

  ⋆ There is a k (the number of states in an acceptor for L),

  ⋆ so that for every $w \in L$

  ⋆ if u is a substring of w of length $\geqslant k$,

  ⋆ then it has a “clippable” substring $y \neq \varepsilon$:
  removing y from w yields a string in L .

- A language ***fails Clipping*** when

  ⋆ for any $k > 0$

  ⋆ we can choose some $w \in L$
  and a substring u of w of length $\geqslant k$,

  ⋆ so that ***any*** clipping within u yields a $w' \notin L$.

The Clipping Property

- The Clipping Theorem says that

  Every language L recognized by a DFA has the following **Clipping Property:**

  - ⋆ There is a k (the number of states in an acceptor for L),
  - ⋆ so that for every $w \in L$
  - ⋆ if u is a substring of w of length $\geqslant k$,
  - ⋆ then it has a “clippable” substring $y \neq \varepsilon$:
    removing y from w yields a string in L .

- A language ***fails Clipping*** when

  - ⋆ for any $k > 0$
  - ⋆ we can choose some $w \in L$
    and a substring u of w of length $\geqslant k$,
  - ⋆ so that ***any*** clipping within u yields a $w' \notin L$.

- If L fails Clipping then it is not recognized.

Example: an-bn

- Let $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \geqslant 0\}$
- L fails clipping:

  1. Let $k > 0$
  2. Choose $w = \mathsf{a}^k\mathsf{b}^k$ and $u = \mathsf{a}^k$.
     We have $w \in L$ and $|u| \geqslant k$.
  3. Any clipping in u yields from w
     a w' of the form $\mathsf{a}^p\mathsf{b}^k$ with $p < k$.
     So $w' \notin L$.

- Consequence: L fails the Clipping Property and cannot be recognized.

Example: Unary addition

- Consider the strings representing addition in unary:
  $$A = \{\mathbf{1}^p + \mathbf{1}^q = \mathbf{1}^{p+q} \mid p, q > 0\}.$$
- A fails the Clipping Property:

  1. Let $k > 0$.
  2. Choose $w = \mathbf{1}^k + \mathbf{1} = \mathbf{1}^{k+1}$
     and u the substring $\mathbf{1}^{k+1}$.
     $w \in A$ and $|u| \geqslant k$.
  3. Any clipping in u yields from w a string
     $w' = \mathbf{1}^\ell + \mathbf{1} = \mathbf{1}^{k+1}$ with $\ell < k$.
     $w' \notin A$.

- A fails Clipping, and so cannot be recognized.

Example: Perfect squares in unary

- Consider $L = \{\mathtt{1}^{n^2} \mid n \geqslant 0\}$.
- L fails the Clipping Property:

1. Let $k > 0$.
2. Choose $w = \mathtt{1}^{k^2}$ and $u = \mathtt{1}^{k}$.
   $w \in L$ and $|u| \geqslant k$.
3. For any clipped y we have $1 \leqslant |y| \leqslant |u| = k$,
   so for the reduced string $w' = \mathtt{1}^{\ell}$ where $k^2 - k \leqslant \ell < k^2$.
   $w' \notin L$ because ℓ cannot be a square: the largest square preceding k^2 is $(k-1)^2 = k^2 - 2k + 1$ which is $< k^2 - k \leqslant \ell$.

- So L fails Clipping, and cannot be recognized.

Example: The mahimahi language

- Consider $L = \{x \cdot x \mid x \in \{\mathtt{0}, \mathtt{1}\}^*\}$
- Idea: Take $w = x \cdot x$ with x that starts with a marker.

Example: The mahimahi language

- Consider $L = \{x \cdot x \mid x \in \{\mathtt{0}, \mathtt{1}\}^*\}$
- Idea: Take $w = x \cdot x$ with x that starts with a marker.

1. Let $k > 0$.
2. Choose $w = \mathtt{0}\mathtt{1}^k\mathtt{0}\mathtt{1}^k$ and $u =$ left substring $\mathtt{1}^k$ in w.
   $w \in L$ and $|u| \geqslant k$.

Example: The mahimahi language

- Consider $L = \{x \cdot x \mid x \in \{\mathtt{0},\mathtt{1}\}^*\}$
- Idea: Take $w = x \cdot x$ with x that starts with a marker.

1. Let $k > 0$.
2. Choose $w = \mathtt{01}^k\mathtt{01}^k$ and $u =$ left substring $\mathtt{1}^k$ in w.
   $w \in L$ and $|u| \geqslant k$.
3. Any clipped y in u yields from w
   a reduced string $w' = \mathtt{01}^\ell\mathtt{01}^k$
   where $\ell < k$.
   Such w' cannot be of the form xx,
   because its first half starts with $\mathtt{0}$
   while its second half starts with $\mathtt{1}$.

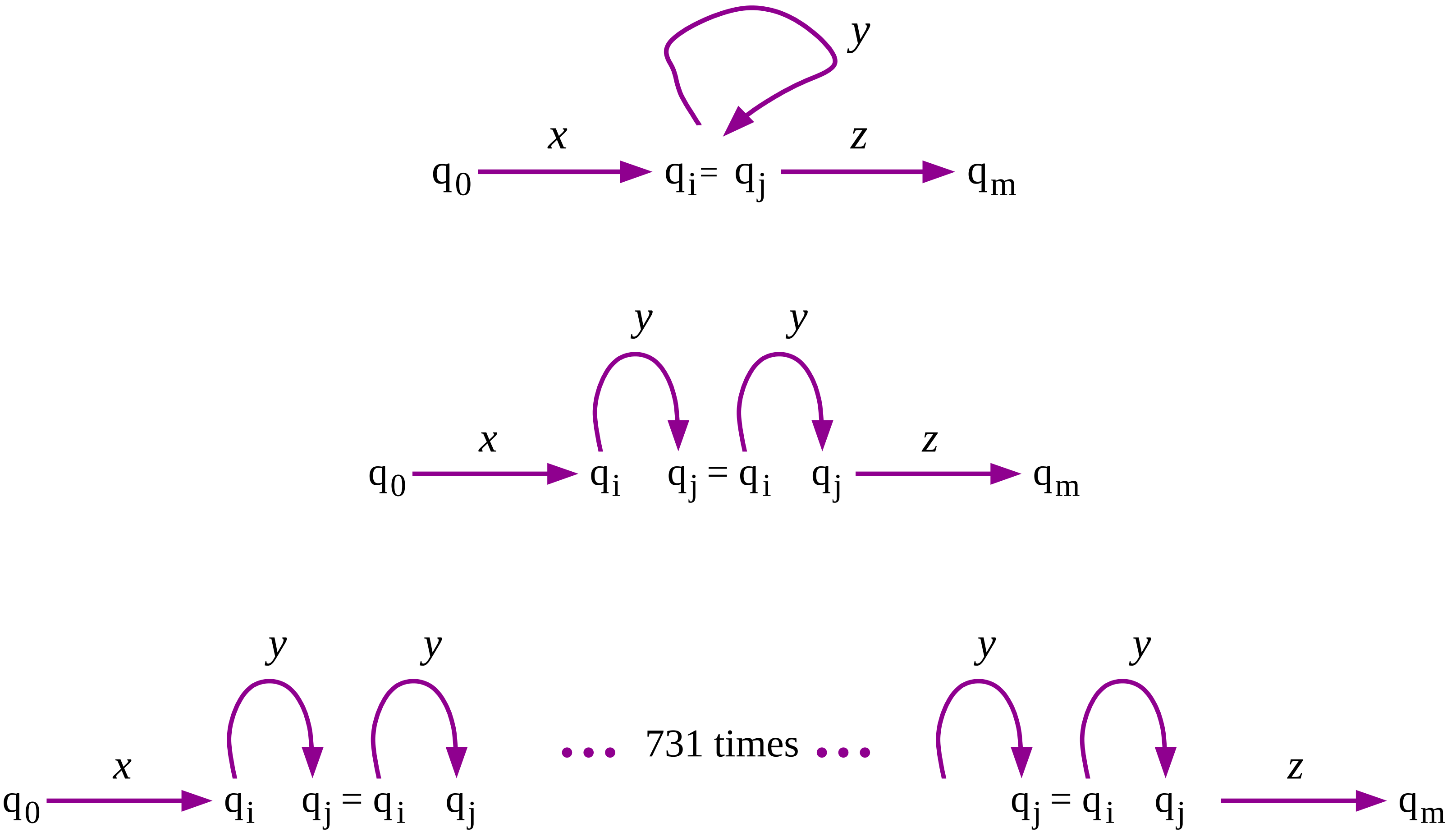
Example: The mahimahi language

- Consider $L = \{x \cdot x \mid x \in \{\mathtt{0}, \mathtt{1}\}^*\}$
- Idea: Take $w = x \cdot x$ with x that starts with a marker.

1. Let $k > 0$.
2. Choose $w = \mathtt{0}\mathtt{1}^k\mathtt{0}\mathtt{1}^k$ and $u =$ left substring $\mathtt{1}^k$ in w.
   $w \in L$ and $|u| \geqslant k$.
3. Any clipped y in u yields from w
   a reduced string $w' = \mathtt{0}\mathtt{1}^\ell\mathtt{0}\mathtt{1}^k$
   where $\ell < k$.
   Such w' cannot be of the form xx,
   because its first half starts with $\mathtt{0}$
   while its second half starts with $\mathtt{1}$.

- L fails the Clipping Property, and cannot be recognized.

Pumping up rather than clipping



Pumping instances

- Let $w \in \Sigma^*$ and

  y a particular substring of w: $w = x \cdot y \cdot z$.

- The $\boxed{n\text{-th pumping instance}}$ of $w = x \cdot y \cdot z$

  over (the exhibited occurrence of) y

  is defined to be $x \cdot y^n \cdot z$.

The Pumping Theorem

- Let M be a k-state DFA over Σ, $L = \mathcal{L}(M)$.
- As for Clipping, choose $w \in L$ and a substring u of w of length $\geqslant k$.
- CONCLUDE: u has a non-empty substring y
  such that all pumping instances of w over y are in L.
- Recall: The n*-th pumping instance* of w over
  (a particular occurrence of) y
  is the result of replacing y by y^n.

Failing Pumping

A language ***fails Pumping*** when:

1. For any $k > 0$
2. there are $w \in L$
   and substring u of w of length $\geqslant k$
3. so that for ***every*** y within u
   there is a pumping instance w over y which is not in L.

Example: The Primes

- $L = \{1^p \mid p$ is prime $\}$
- Suppose L is recognized by a k-state DFA M.

Example: The Primes

- $L = \{1^p \mid p \text{ is prime }\}$
- Suppose L is recognized by a k-state DFA M.
- Take a prime $p > k$ and $w = 1^p \in L$.
- There is a pumping segment y in w of length $\ell \neq 0$.

Example: The Primes

- $L = \{1^p \mid p \text{ is prime }\}$
- Suppose L is recognized by a k-state DFA M.
- Take a prime $p > k$ and $w = 1^p \in L$.
- There is a pumping segment y in w of length $\ell \neq 0$.
- The $(p{+}1)$-st pumping instance of w over y has length $|w| - \ell + (p+1)\ell = p + p\ell = p(\ell + 1)$, which is not prime.

Example: The Primes

- $L = \{1^p \mid p \text{ is prime }\}$
- Suppose L is recognized by a k-state DFA M.
- Take a prime $p > k$ and $w = 1^p \in L$.
- There is a pumping segment y in w of length $\ell \neq 0$.
- The $(p{+}1)$-st pumping instance of w over y has length $|w| - \ell + (p+1)\ell = p + p\ell = p(\ell+1)$, which is not prime.
- Contradiction. M cannot exist.

Example: Necessary use of Pumping

- Show that the language

$$L = \{ w \cdot \mathtt{a}^n \mid w \in \{\mathtt{a}, \mathtt{b}\}^*,\ \#_a(w) = n \}$$

is not recognized.

Example: Necessary use of Pumping

- Show that the language

$$L = \{w \cdot \mathtt{a}^n \mid w \in \{\mathtt{a}, \mathtt{b}\}^*,\ \#_a(w) = n\ \}$$

  is not recognized.

- Suppose L were recognized by a k-state DFA.
  Let $w = \mathtt{b}^k\mathtt{a}^k$, which is in L,
  and take $u = \mathtt{b}^k$, the prefix of w.

Example: Necessary use of Pumping

- Show that the language

$$L = \{ w \cdot \mathtt{a}^n \mid w \in \{\mathtt{a}, \mathtt{b}\}^*,\ \#_a(w) = n \}$$

  is not recognized.

- Suppose L were recognized by a k-state DFA.
  Let $w = \mathtt{b}^k \mathtt{a}^k$, which is in L,
  and take $u = \mathtt{b}^k$, the prefix of w.

- By the Pumping Theorem u has a substring $y = \mathtt{b}^\ell$ where $\ell > 0$ such that $\mathtt{b}^{k+n\ell}\,\mathtt{a}^k \in L$ for all $n \geqslant 0$. In particular, for $n = 1$ we have $w' = \mathtt{b}^{k+\ell}\,\mathtt{a}^k \in L$.

Example: Necessary use of Pumping

- Show that the language

  $$L = \{w \cdot \mathtt{a}^n \mid w \in \{\mathtt{a},\mathtt{b}\}^*,\ \#_a(w) = n\,\}$$

  is not recognized.

- Suppose L were recognized by a k-state DFA.
  Let $w = \mathtt{b}^k\mathtt{a}^k$, which is in L,
  and take $u = \mathtt{b}^k$, the prefix of w.

- By the Pumping Theorem u has a substring $y = \mathtt{b}^\ell$ where $\ell > 0$ such that $\mathtt{b}^{k+n\ell}\,\mathtt{a}^k \in L$ for all $n \geqslant 0$. In particular, for $n = 1$ we have $w' = \mathtt{b}^{k+\ell}\,\mathtt{a}^k \in L$.

  But this is impossible, because the second half of w' must have $\mathtt{b}$ ’s.

- Thus no DFA recognizing L exists.

Minimum states for finite language recognition

- Any ***finite*** language L is recognized by an automaton!
- But how many states are needed?

Minimum states for finite language recognition

- Any ***finite*** language L is recognized by an automaton!
- But how many states are needed?
- At least as many as the longest string-length in L.

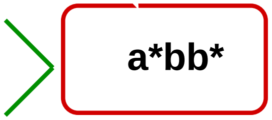
Minimum states for finite language recognition

- Any ***finite*** language L is recognized by an automaton!
- But how many states are needed?
- At least as many as the longest string-length in L.
- Proof: If M with k states recognizes a string longer than k, then Pumping applies, and L is infinite!

CONSTRUCTING AUTOMATA

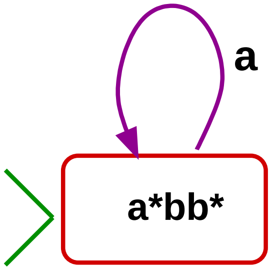
- We give a method that, given a language L, attempts to construct a DFA M recognizing L.
- If and when the process teminates, we obtain such an M.
- We start with a couple of non-trivial examples, before articulating the method and giving more examples.

Example: a ’s precede b ’s



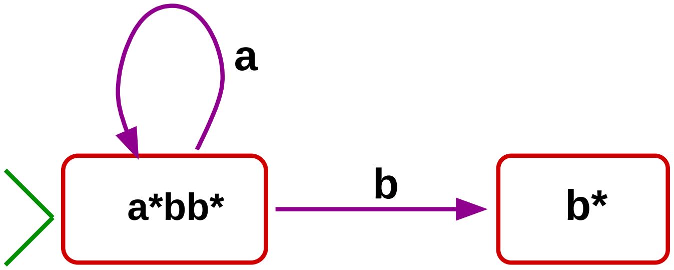
- Construct an automaton recognizing $\mathcal{L}(a^*bb^*)$. That is, accepting strings of a ’s followed by one or more b ’s, and ***only*** those.
- The initial state is the declaration of this goal.
- What will be an updated goal after reading an a ?

Reading an `a`



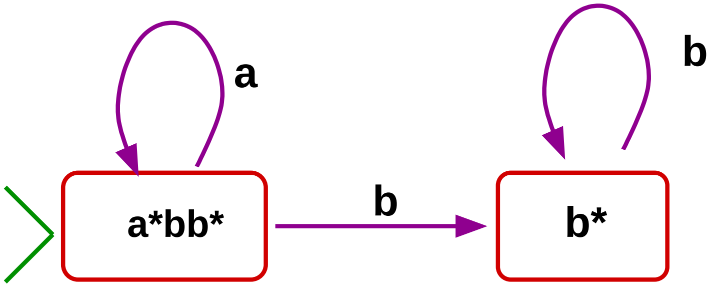
- The goal is unchanged!.
- But what happens if we read a `b` ?

Reading a b



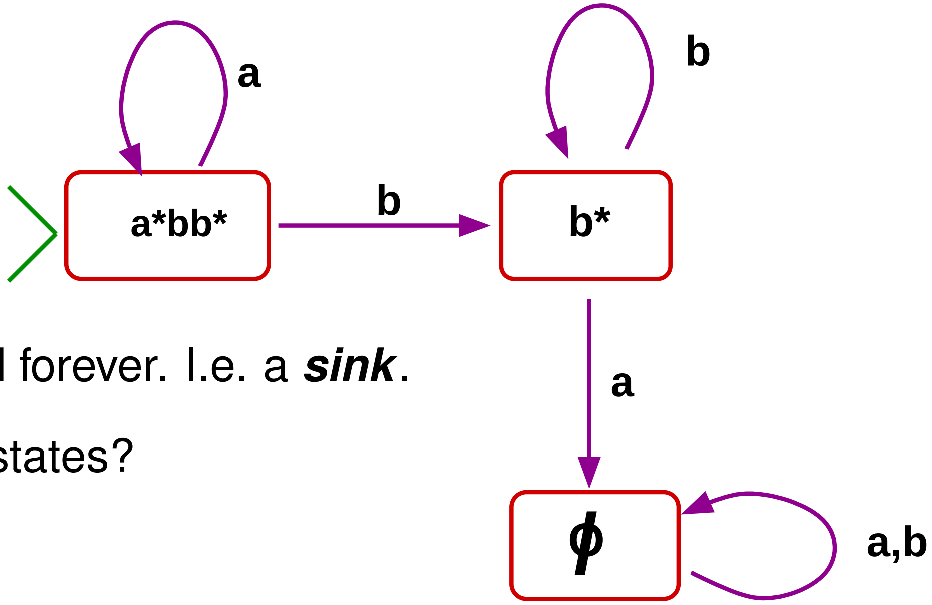
- A new goal: from now on only b ’s, any number.
- What if we read a b ***now***?

Reading another `b`



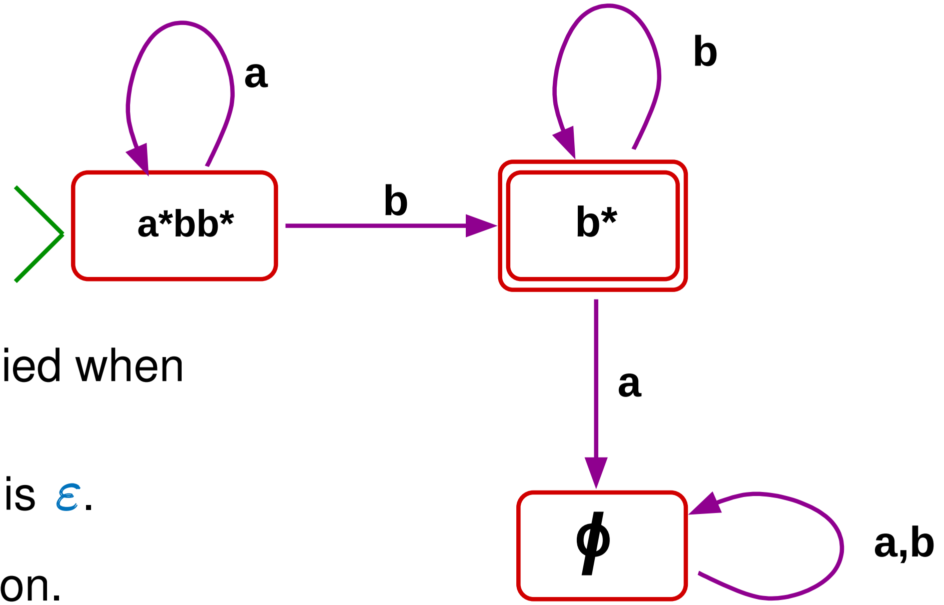
- No change.
- And what if, instead, we read an `a` ?

Reading an `a` *instead*



- This is a non-accept, now and forever. I.e. a ***sink***.
- And which are the accepting states?

What are the accepting states

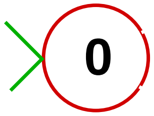


- Accept if current goal is satisfied when nothing left to read,
  i.e. when the current string is ε.
- This completes the construction.

Example: Ending as it starts

0 $\sigma w \sigma$

ε



*****

- Construct an automaton accepting strings $\sigma w \sigma$,

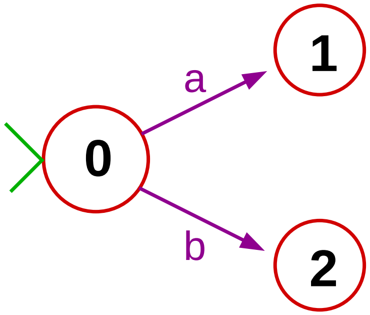
  i.e. with last letter identical to the first, and ***no others***.

- The initial state is the declaration of this goal.

- What will be the updated goals after reading the first letter?

Example: Ending as it starts

Reading the first letter:



0 $\sigma w \sigma$

1 $\varepsilon \mid w\mathrm{a}$

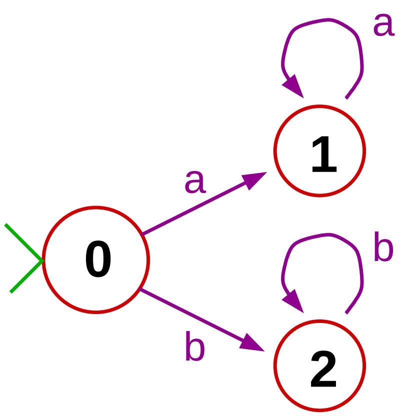
2 $\mid w\mathrm{b}$

*

- Either this is the last letter, or else it repeats at the end.
- What if we now read this letter again?

Example: Ending as it starts

Sought letter repeated:



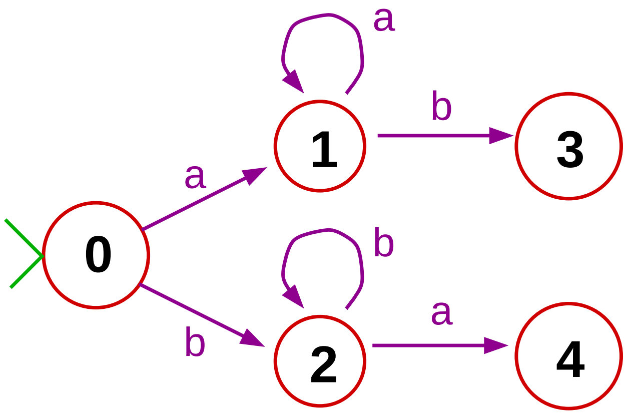
0	$\sigma w \sigma$
1	$\varepsilon \mid w\mathrm{a}$
2	$\varepsilon \mid w\mathrm{b}$

*

- The goal does not change.
- And what about the opposite letter ***now***?

Example: Ending as it starts

Reading opposite letter:



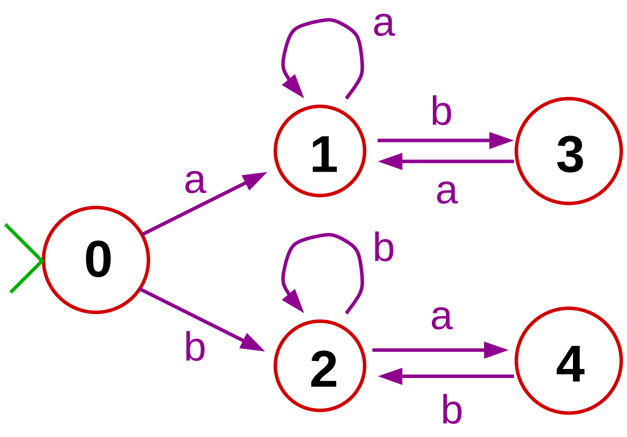
0	$\sigma w \sigma$
1	$\varepsilon \mid w$a
2	$\varepsilon \mid w$b
3	wa
4	wb

*

- The option of not reading further has been blocked.

Example: Ending as it starts

Opposite letter repeating:



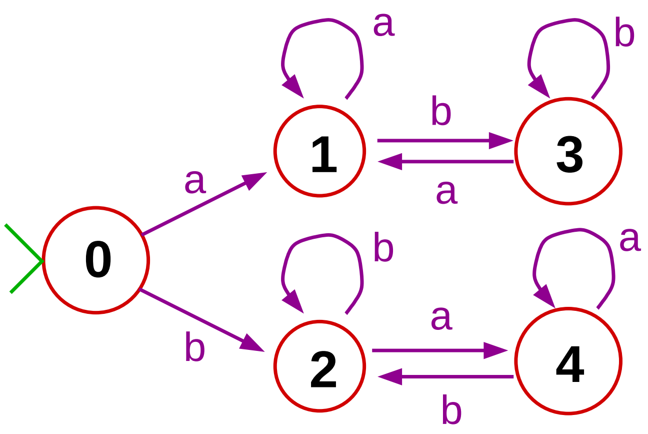
0	$\sigma w \sigma$
1	ε **|** wa
2	ε **|** wb
3	wa
4	wb

*

- But if the sought letter is read now,
  the previous goal is restored.
- And if we keep reading the wrong letter?

Example: Ending as it starts

Return to sought letter:



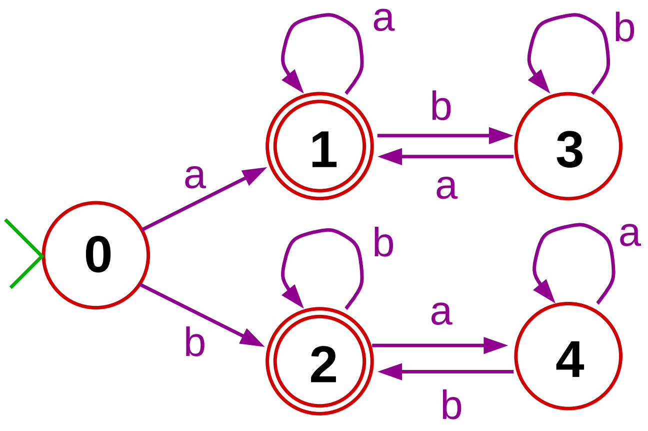
0	$\sigma w \sigma$
1	$\varepsilon \mid w\mathrm{a}$
2	$\varepsilon \mid w\mathrm{b}$
3	$w\mathrm{a}$
4	$w\mathrm{b}$

*

- No change of goal.
- What are the accepting states?

Example: Ending as it starts

The accepting states:



0	$\sigma w \sigma$
1	$\varepsilon \mid w\mathrm{a}$
2	$\varepsilon \mid w\mathrm{b}$
3	$w\mathrm{a}$
4	$w\mathrm{b}$

*

- Accept if current goal is satisfied when nothing left to read.
- This completes the construction.

Goal oriented automaton construction

- *When you head to an unfamiliar destination,
would you prefer the GPS map to display the road already covered,
or rather the road ahead?*

Goal oriented automaton construction

- ***When you head to an unfamiliar destination,***
  would you prefer the GPS map to display the road already covered,
  or rather the road ahead?

- Programming is a ***goal oriented*** process.
  The relevant mission is to achieve a goal.
  The initial task of an acceptor for L is
  "accept the strings in L and no others"!

Goal oriented automaton construction

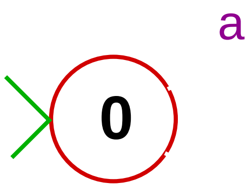
- ***When you head to an unfamiliar destination,***
  would you prefer the GPS map to display the road already covered,
  or rather the road ahead?
- Programming is a ***goal oriented*** process.
  The relevant mission is to achieve a goal.
  The initial task of an acceptor for L is
  "accept the strings in L ***and no others"!***
- The tasks are adjusted as the input string is read.
  Each task is of the form

the string ahead leads into a string in L

Identifying accepting tasks

- The development above updates states (conditions) as required when symbols σ are read.
- A string $x = \sigma u$ satisfying the current condition (=state) leads to A iff u started at the next condition leads to A.
- So the accepting conditions are the ones that are satisfied when reading ends, i.e. when the string-ahead is ε.

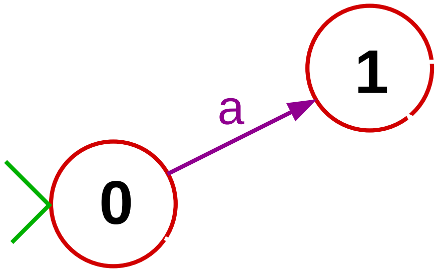
Example: Repeated last symbol



state dictionary

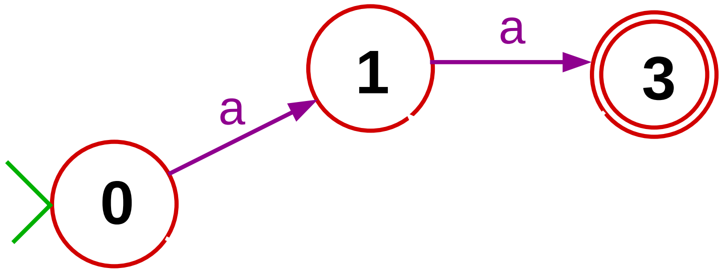
0	$w\sigma\sigma$

Example: Repeated last symbol



0	$w\sigma\sigma$
1	a **|** $w\sigma\sigma$

Example: Repeated last symbol

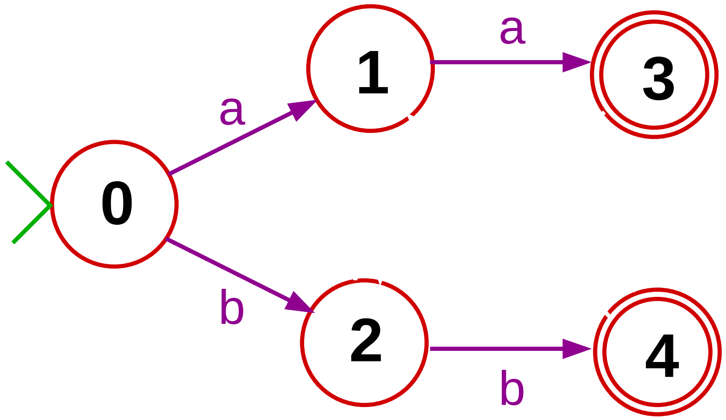


0 $w\sigma\sigma$

1 a **|** $w\sigma\sigma$

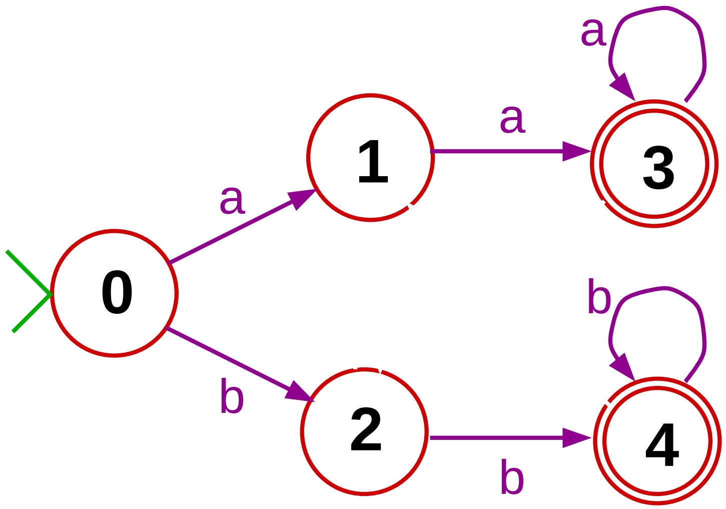
3 ε **|** a **|** $w\sigma\sigma$

Example: Repeated last symbol



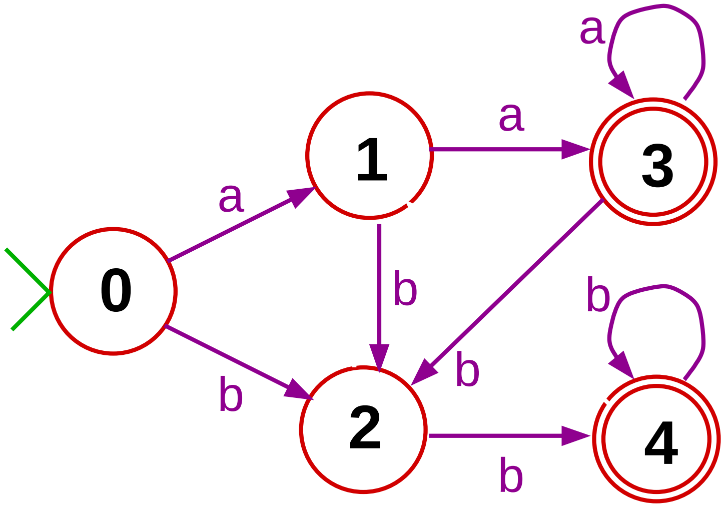
0	$w\sigma\sigma$
1	$\mathrm{a} \mid w\sigma\sigma$
2	$\mathrm{b} \mid w\sigma\sigma$
3	$\varepsilon \mid \mathrm{a} \mid w\sigma\sigma$
4	$\varepsilon \mid \mathrm{b} \mid w\sigma\sigma$

Example: Repeated last symbol



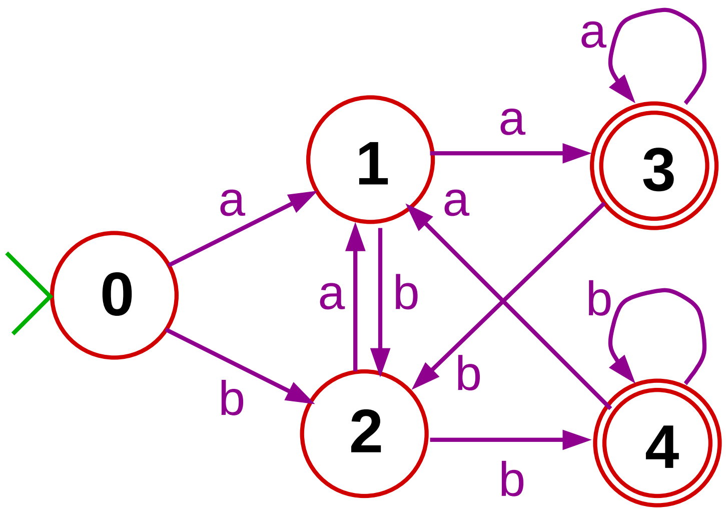
0	$w\sigma\sigma$
1	a **|** $w\sigma\sigma$
2	b **|** $w\sigma\sigma$
3	ε **|** a **|** $w\sigma\sigma$
4	ε **|** b **|** $w\sigma\sigma$

Example: Repeated last symbol



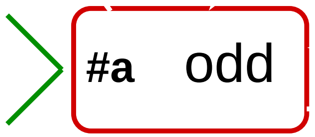
0	$w\sigma\sigma$
1	$\mathrm{a} \mid w\sigma\sigma$
2	$\mathrm{b} \mid w\sigma\sigma$
3	$\varepsilon \mid \mathrm{a} \mid w\sigma\sigma$
4	$\varepsilon \mid \mathrm{b} \mid w\sigma\sigma$

Example: Repeated last symbol



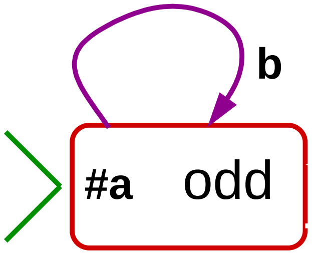
0	$w\sigma\sigma$
1	a $\mid$ $w\sigma\sigma$
2	b $\mid$ $w\sigma\sigma$
3	ε $\mid$ a $\mid$ $w\sigma\sigma$
4	ε $\mid$ b $\mid$ $w\sigma\sigma$

Example: Recognizing odd length



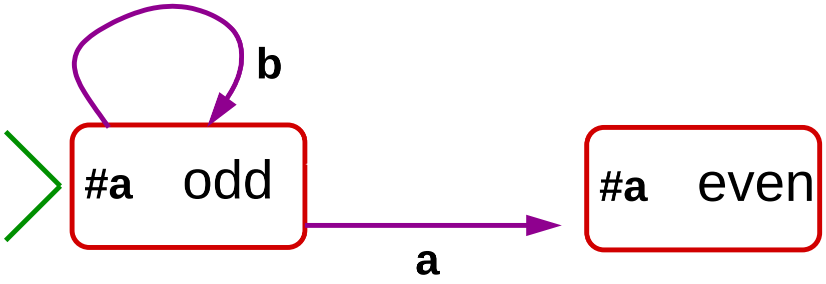
- Initial task: accept strings with an odd number of a’s

Example: Recognizing odd length



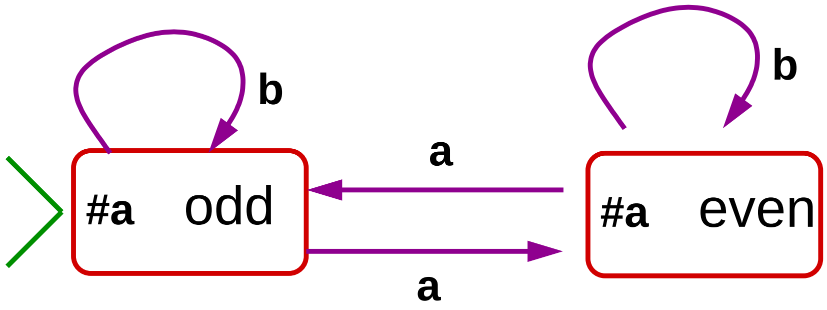
- Reading a b does not change the task

Example: Recognizing odd length



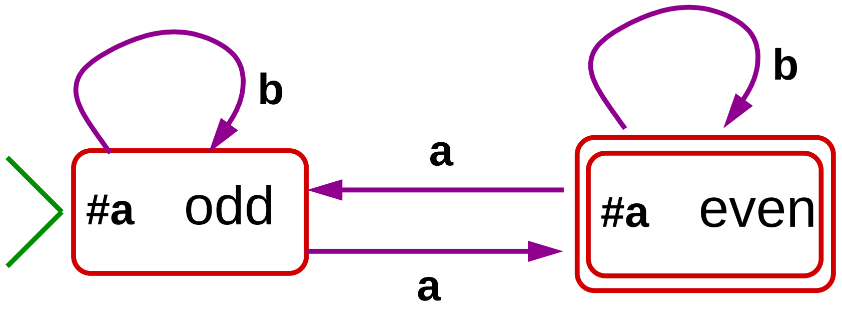
- Reading an a revises the task to:
  accept strings with an even number of a's

Example: Recognizing odd length



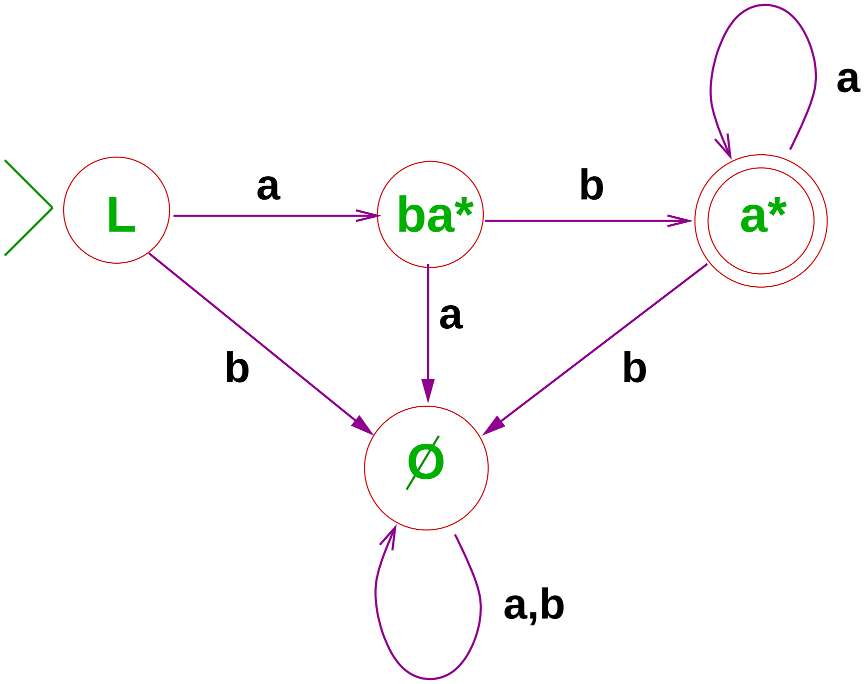
- Same reasoning for the “even” task

Example: Recognizing odd length



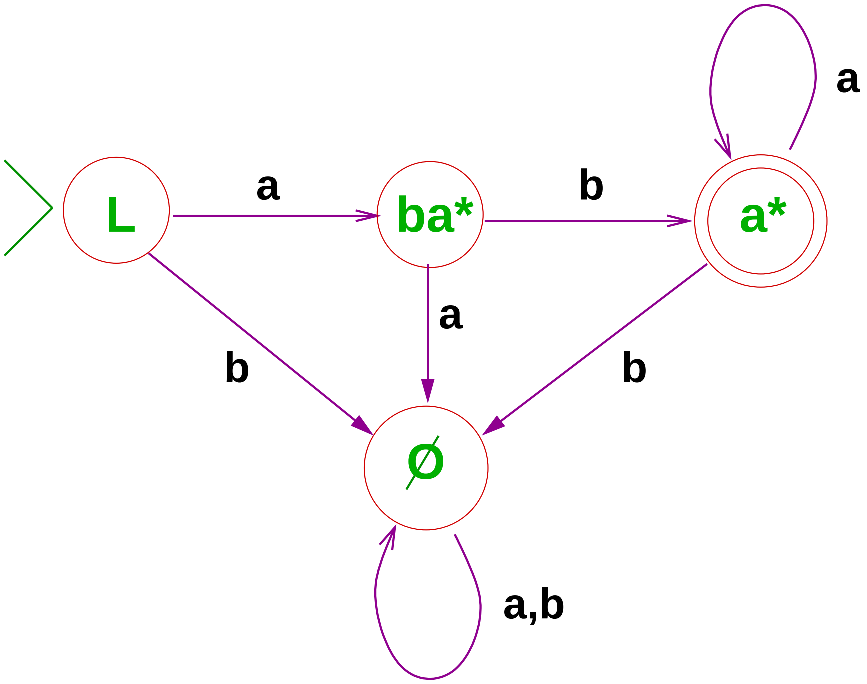
- Accept description fulfilled by ε.

Example: aba*



Accepts the strings of the form $\mathtt{aba}^n$ with $n \geqslant 0$, and no others.

Example: aba*

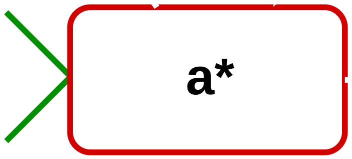


Accepts the strings of the form $\mathtt{aba}^n$ with $n \geqslant 0$, and no others.

- Note the sink at the bottom of the diagram.

A trivial example: Just $\mathtt{a}$ ’s

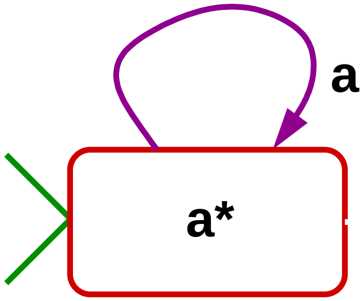
Construct an automaton recognizing $\mathcal{L}(\mathtt{a}^{\star})$
as a sub-language of $\{\mathtt{a}, \mathtt{b}\}^{*}$



- Initial task: accept strings of $\mathtt{a}$’s

A trivial example: Just a *’s*

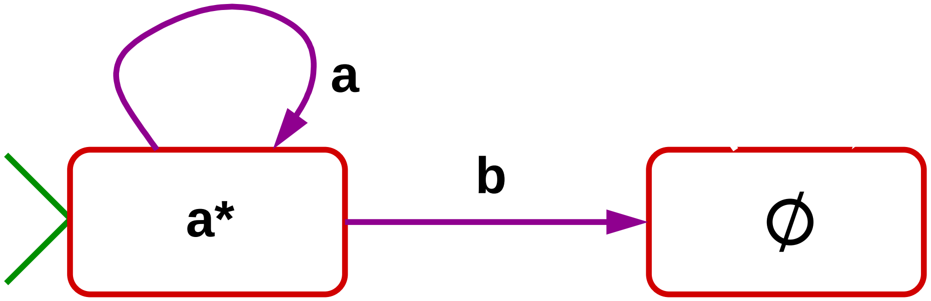
Construct an automaton recognizing $\mathcal{L}(\mathtt{a}^{\star})$ as a sub-language of $\{\mathtt{a}, \mathtt{b}\}^{*}$



- Reading an a does not change the task

A trivial example: Just $\mathtt{a}$ *'s*

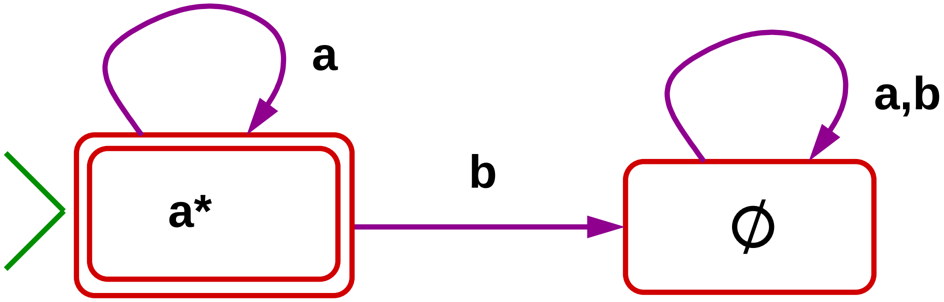
Construct an automaton recognizing $\mathcal{L}(\mathtt{a}^{\star})$ as a sub-language of $\{\mathtt{a}, \mathtt{b}\}^{*}$



- Reading a $\mathtt{b}$ revises the task to not accepting anything. A ***sink.***

A trivial example: Just $\mathtt{a}$ *’s*

Construct an automaton recognizing $\mathcal{L}(\mathtt{a}^*)$ as a sub-language of $\{\mathtt{a},\mathtt{b}\}^*$

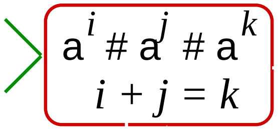


- No escape from the sink

Example: Addition mod 2

Automaton over $\{\mathsf{a}, \#\}$ recognizing

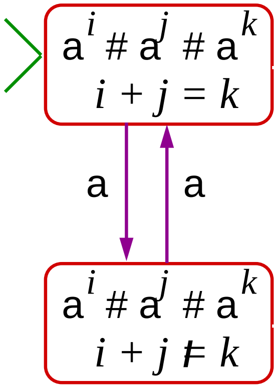
$$\{\mathsf{a}^i \# \mathsf{a}^j \# \mathsf{a}^k \;\mid\; i + j = k \text{ (mod 2) } \}$$



Example: Addition mod 2

Automaton over $\{\mathsf{a}, \#\}$ recognizing

$$\{\mathsf{a}^i \# \mathsf{a}^j \# \mathsf{a}^k \;\mid\; i + j = k \text{ (mod 2) } \}$$

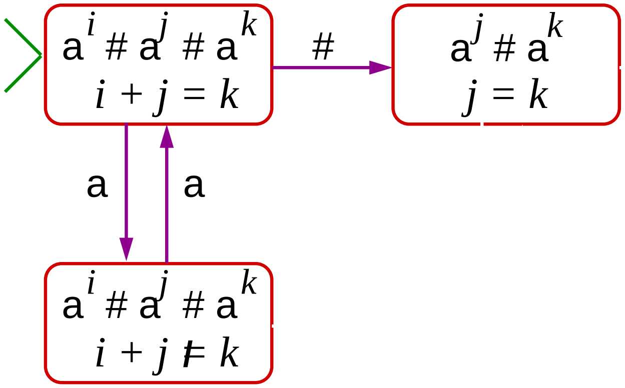

$\mathsf{a}^i \# \mathsf{a}^j \# \mathsf{a}^k$
$i + j = k$
a
a
$\mathsf{a}^i \# \mathsf{a}^j \# \mathsf{a}^k$
$i + j \neq k$

Reading a's toggles between equlity and inequality of parities.

Example: Addition mod 2

Automaton over $\{a, \#\}$ recognizing

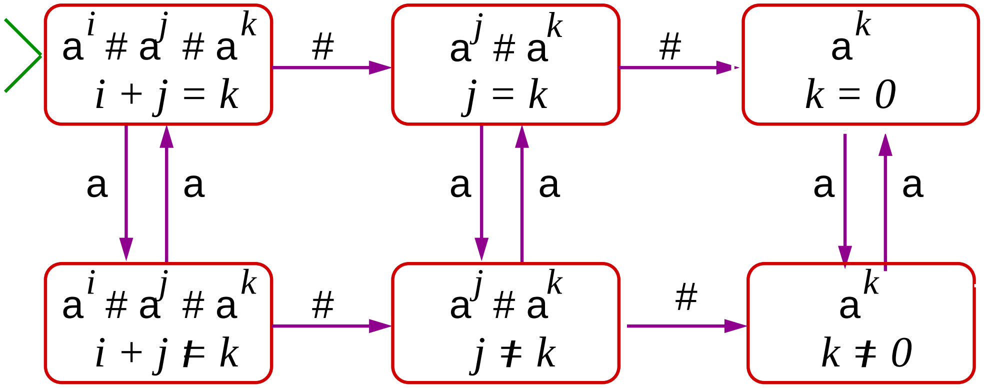
$$\{a^i \# a^j \# a^k \mid i + j = k \text{ (mod 2) } \}$$



Reading the separator $\#$ means $i = 0$.

Example: Addition mod 2

Automaton over $\{a, \#\}$ recognizing
$$\{a^i \# a^j \# a^k \mid i + j = k \text{ (mod 2)} \}$$

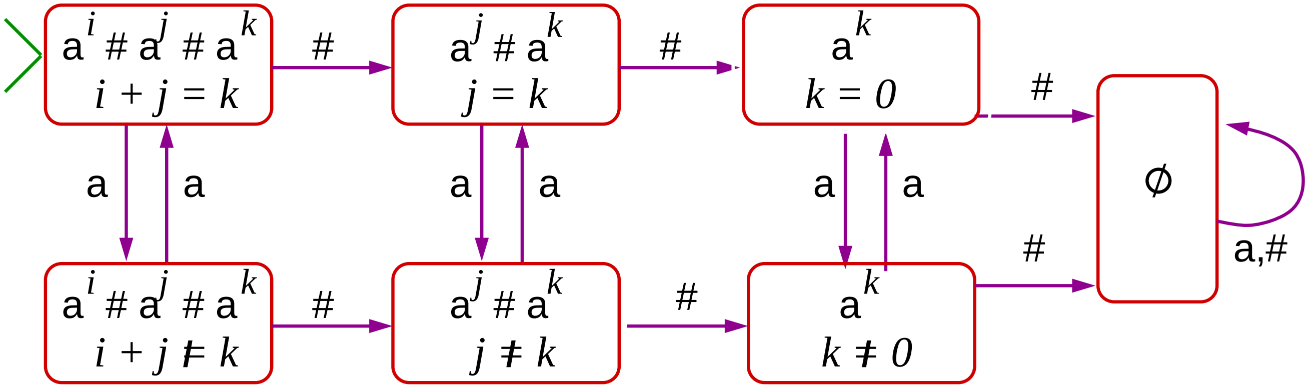


The same arguments are repeated

Example: Addition mod 2

Automaton over $\{a, \#\}$ recognizing

$$\{a^i \# a^j \# a^k \mid i + j = k \text{ (mod 2)} \}$$

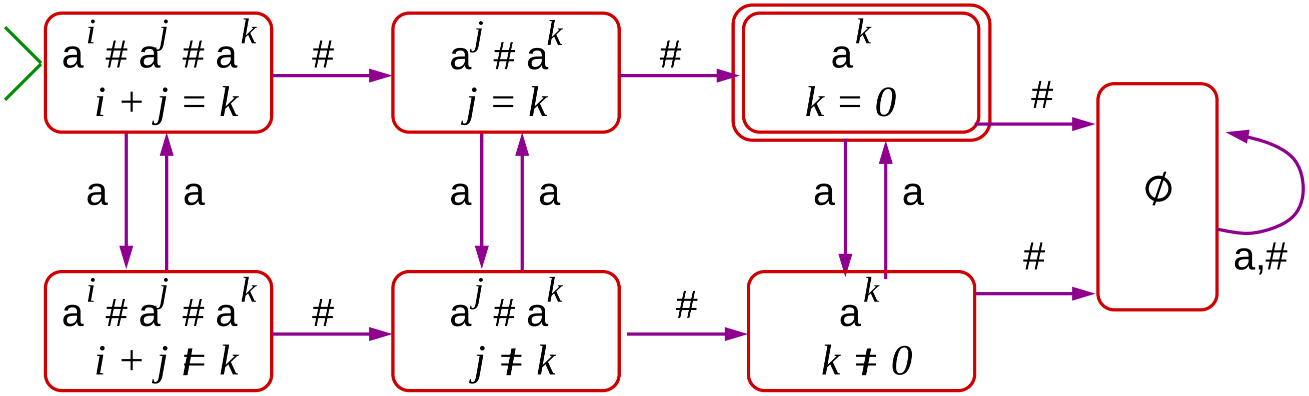


Encountering an extra separator leads to a sink

Example: Addition mod 2

Automaton over $\{a, \#\}$ recognizing

$$\{a^i \# a^j \# a^k \mid i + j = k \text{ (mod 2) } \}$$



The single one accepting state is the one satisfied by ε.

Summary of the method, again

- The initial acceptance-condition is the language to be recognized.
- Given a new acceptance-condition we calculate for each $\sigma \in \Sigma$ how reading σ leads to a new acceptance-condition.

  That is, a string $w = \sigma u$ satisfies the current acceptance condition iff u satisfies the acceptance-condition after σ is read.
- An acceptance-condition is an accepting state iff it is satisfied by ε.

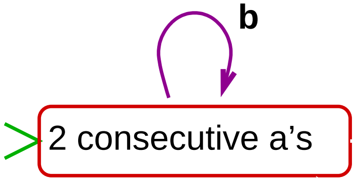
Example: Two consecutive $\mathtt{a}$*'s*

Construct an automaton recognizing $\mathcal{L}(\Sigma^* \cdot \mathtt{aa} \cdot \Sigma^*)$

2 consecutive a's

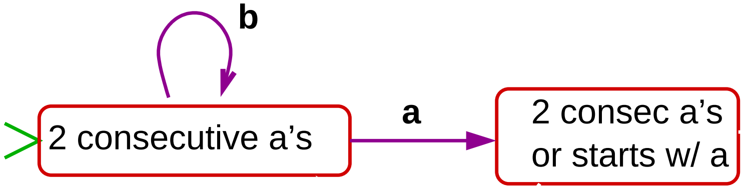
Example: Two consecutive `a`'s

Reading `b` leaves the task unchanged:



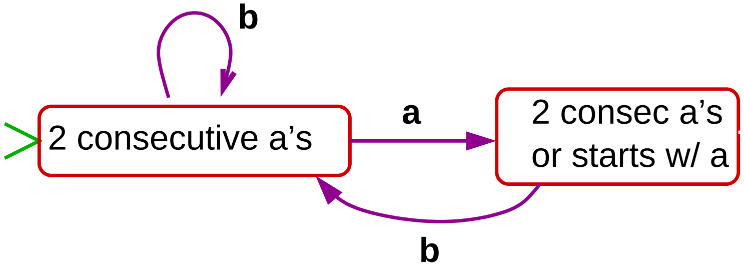
Example: Two consecutive a *’s*

But reading a opens two future options:



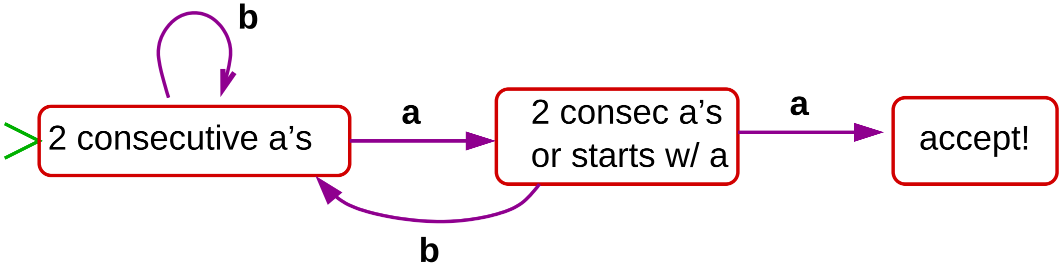
Example: Two consecutive a's

From these two options reading b kills the first:



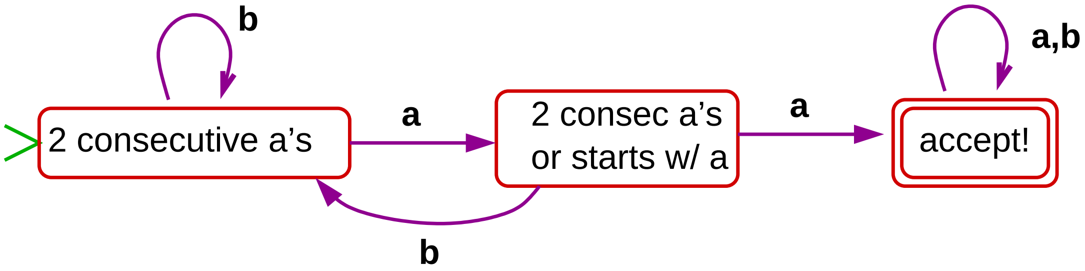
Example: Two consecutive a *'s*

But reading an a settles acceptance:

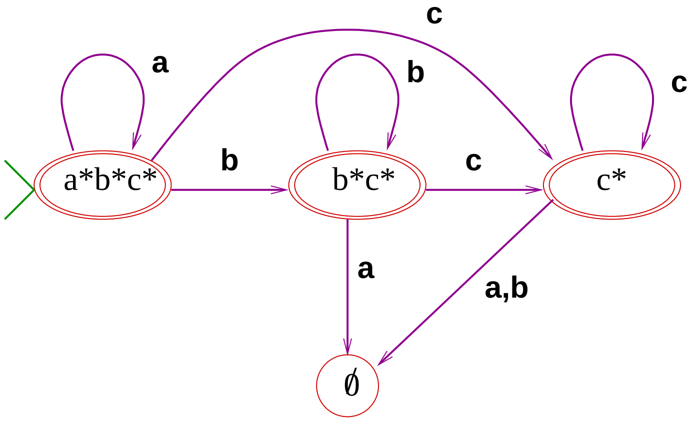


Example: Two consecutive `a` *’s*

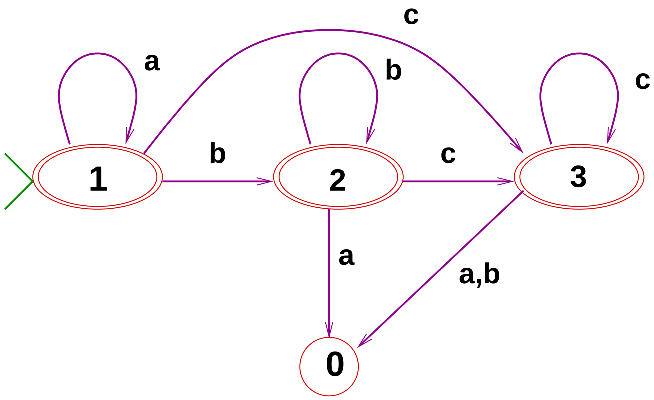
No further reading alterns that conclusion:



Example 7: `a*b*c*`

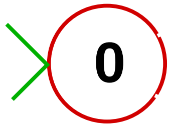


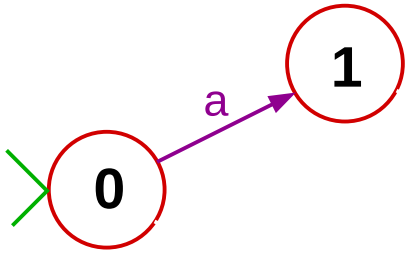
- Label states as we wish, with optional “dictionary.”



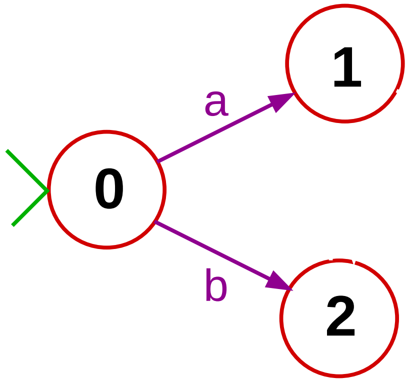
Example 8: Ends with two identical

0 $*\sigma\sigma$

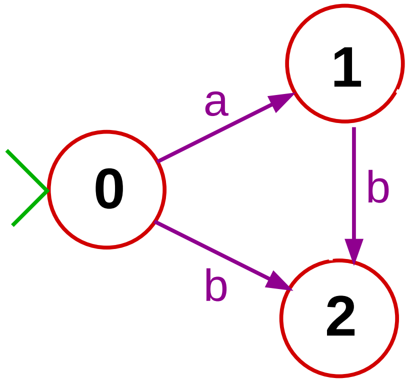




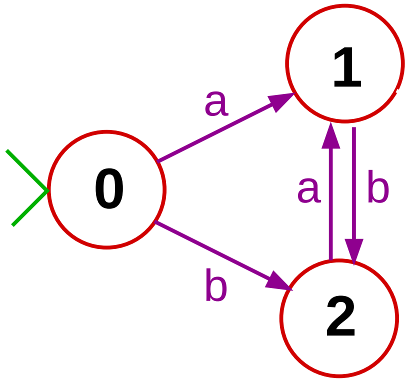
0	*σσ
1	a **|** *σσ



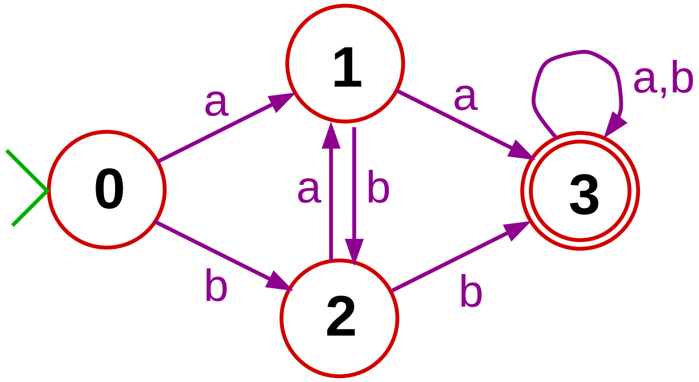
0	*σσ
1	a | *σσ
2	b | *σσ



0	*σσ
1	a **|** *σσ
2	b **|** *σσ

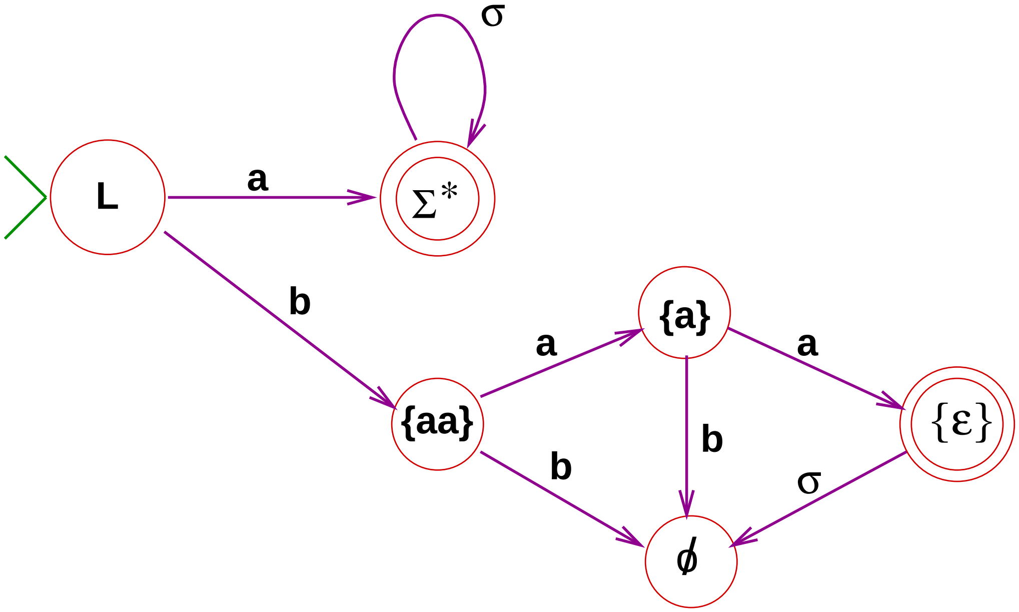


0 *σσ
1 a | *σσ
2 b | *σσ



0	*σσ
1	a | *σσ
2	b | *σσ
3	*

Example: Initial `a` *or the string* `baa`



Example: Symbolic binary addition

- The following example illustrates the use of compound data as “symbols” of an alphabet.

- Consider a long addition in binary, such as $$\begin{array}{cccccc} & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 \end{array}$$

Example: Symbolic binary addition

- The following example illustrates the use of compound data as “symbols” of an alphabet.

- Consider a long addition in binary, such as $\begin{array}{cccccc} & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 \end{array}$

- This table does not look like a string.
  But all such tables have height 3 we can consider each column as a “symbol” in the alphabet $\Sigma = \{0,1\}^3$, that is

$$\Sigma^3 = \{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \}$$

Example: Symbolic binary addition

- The following example illustrates the use of compound data as "symbols" of an alphabet.
- Consider a long addition in binary, such as $\begin{array}{cccccc} & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 \end{array}$
- This table does not look like a string.
  But all such tables have height 3 we can consider each column as a "symbol" in the alphabet $\Sigma = \{0,1\}^3$, that is

$$\Sigma^3 = \{ \begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\1\\1\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\end{bmatrix}, \begin{bmatrix}1\\1\\0\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix} \}$$

- The long addition above can be consrued as the string $\begin{bmatrix}0\\0\\1\end{bmatrix} \begin{bmatrix}0\\1\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\end{bmatrix} \begin{bmatrix}1\\0\\1\end{bmatrix} \begin{bmatrix}0\\1\\1\end{bmatrix}$

An automaton recognizing symbolic binary addition

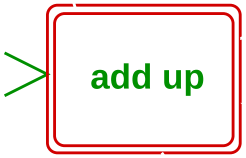
- Is there an automaton over Σ^3 that recognizes the correct symbolic binary additions?
- That is, can we construct an automaton M that accepts strings like

$$\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}$$

but not strings like

$$\begin{bmatrix}0\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}$$

An automaton recognizing symbolic binary addition



Start state is the goal that the table ***adds-up***:

remaining columns add up

An automaton recognizing symbolic binary addition

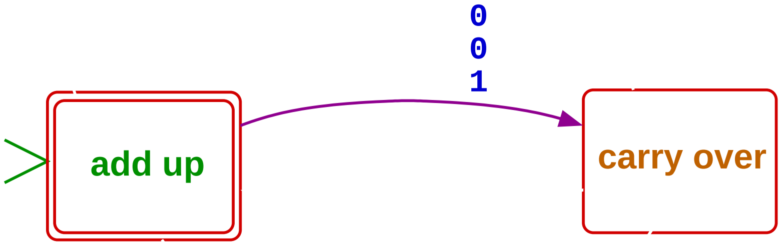
add up
carry over

Start state is the goal that the table ***adds-up***:

remaining columns add up

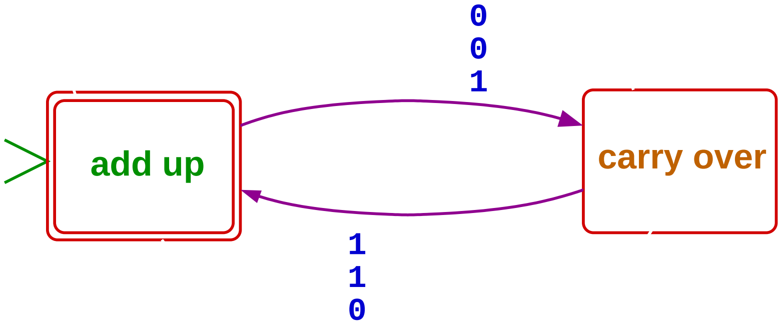
The main other state is *remaining columns yield* ***carry-over***

An automaton recognizing symbolic binary addition



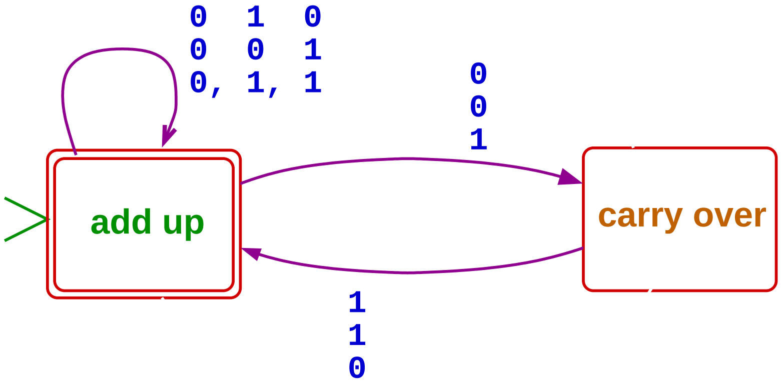
There is one column switching from ***add-up*** to ***carry-over***

An automaton recognizing symbolic binary addition



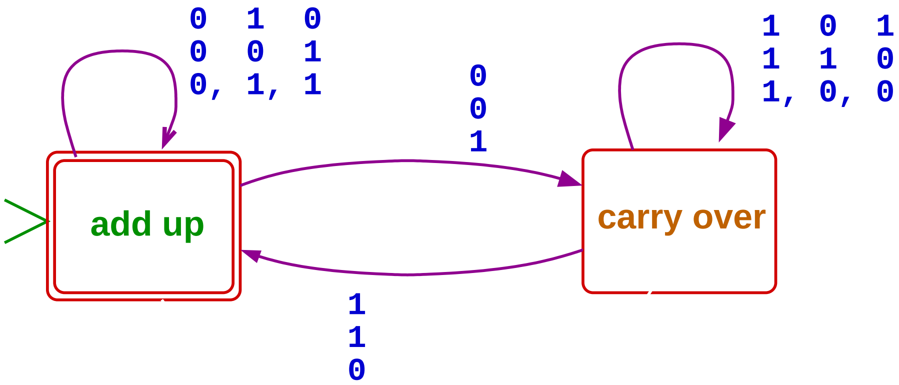
There is one column switching from ***add-up*** to ***carry-over***
and one column switching back from ***carry-over*** to ***add-up***

An automaton recognizing symbolic binary addition



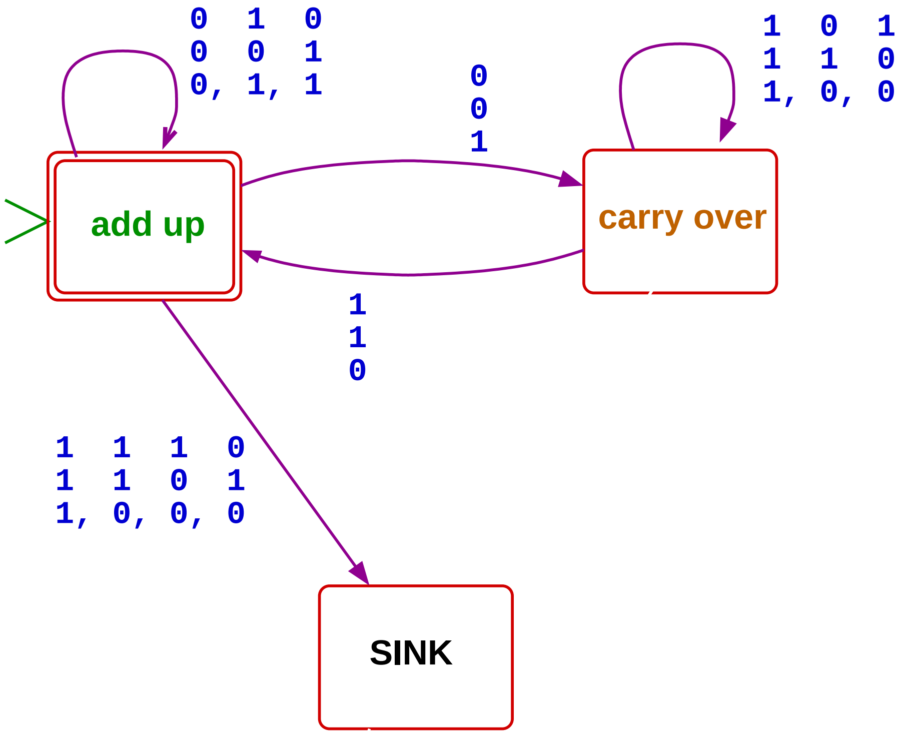
Three columns leave the ***add-up*** goal unchanged

An automaton recognizing symbolic binary addition



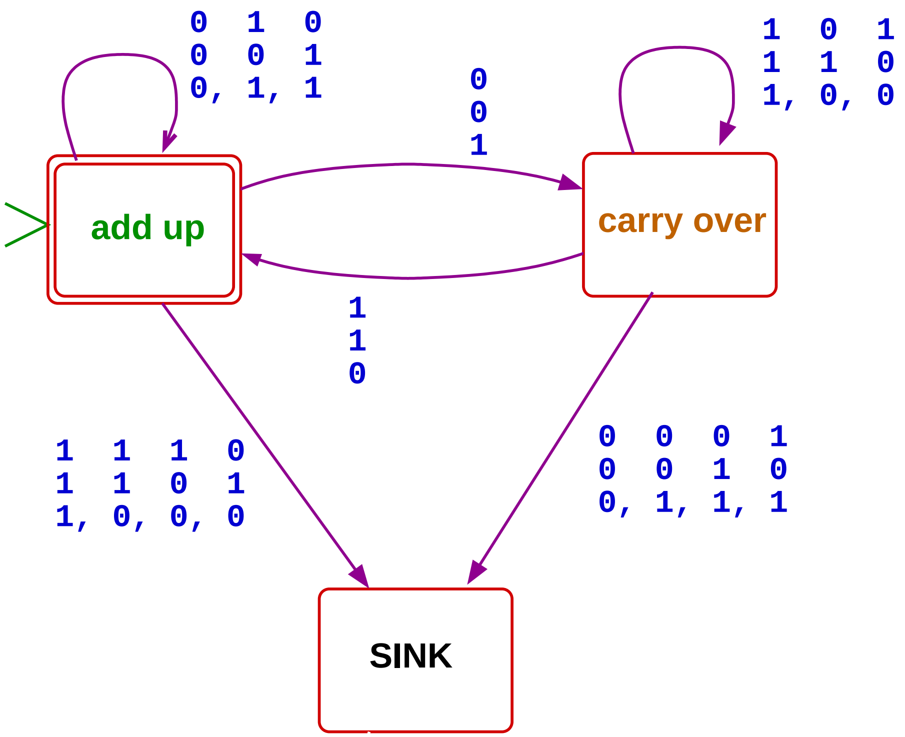
Three columns leave the ***add-up*** goal unchanged
and three leaave ***carry-over*** unchaged

An automaton recognizing symbolic binary addition



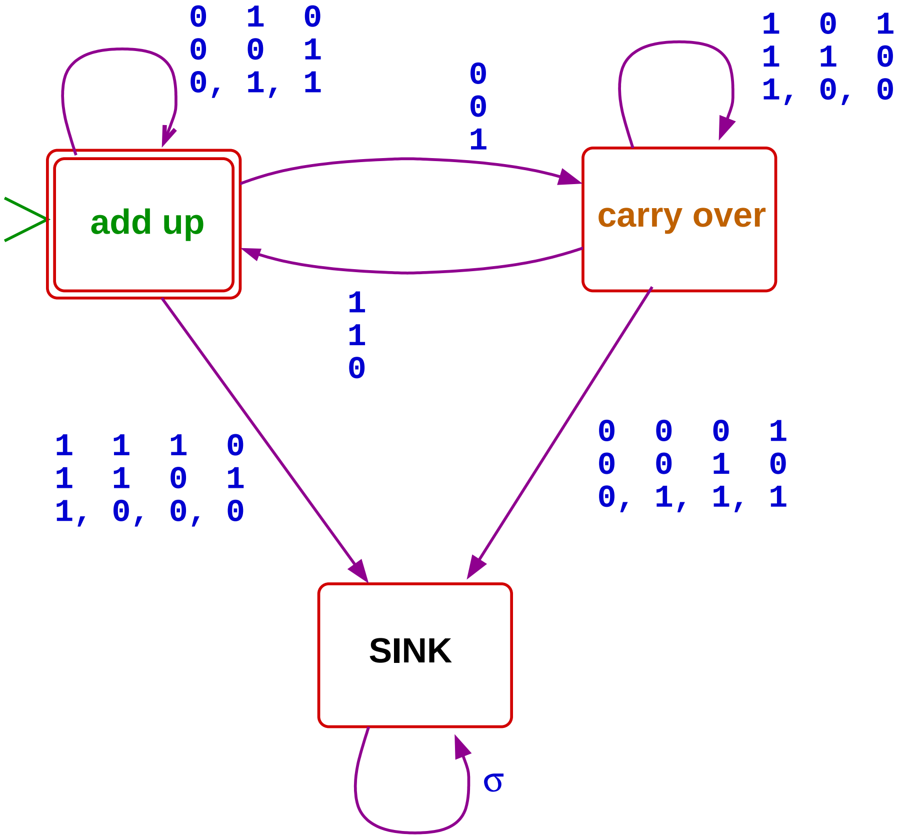
Four columns lead from ***add-up*** to a ***sink***

An automaton recognizing symbolic binary addition



Four columns lead from ***add-up*** to a ***sink***
and four from ***carry-over*** to that ***sink***

An automaton recognizing symbolic binary addition



Finally, ***sink*** is a sink.

Example: Binary numerals divisible by 3

- Consider every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$ to be a binary numerals.
- The ***numeric value*** $[w]_2$ of a string $w = d_k d_{k-1} \cdots d_0$ is $\Sigma_i\ 2^i$.
- The numerals divisible by 2 are those that end with $\mathtt{0}$.

Example: Binary numerals divisible by 3

- Consider every string $w \in \{0, 1\}^*$ to be a binary numerals.
- The ***numeric value*** $[w]_2$ of a string $w = d_k d_{k-1} \cdots d_0$ is $\Sigma_i \ 2^i$.
- Problem: Construct a DFA over $\{0, 1\}^*$ that accepts the numerals divisble by 3.

Example: Binary numerals divisible by 3

- Consider every string $w \in \{0,1\}^*$ to be a binary numerals.
- The ***numeric value*** $[w]_2$ of a string $w = d_k d_{k-1} \cdots d_0$ is $\Sigma_i\ 2^i$.
- Problem: Construct a DFA over $\{0,1\}^*$ that accepts the numerals divisible by 3.
- Preliminary: What is the value mod(3) of the digits, i.e. what is 2^k mod(3).

Example: Binary numerals divisible by 3

- Consider every string $w \in \{0,1\}^*$ to be a binary numerals.
- The ***numeric value*** $[w]_2$ of a string $w = d_k d_{k-1} \cdots d_0$ is $\Sigma_i \; 2^i$.
- Problem: Construct a DFA over $\{0,1\}^*$ that accepts the numerals divisble by 3.
- Preliminary: What is the value mod(3) of the digits, i.e. what is 2^k mod(3).

We have that $4^k =_3 1$, by induction on k.

  - ▸ $4^0 = 1$
  - ▸ If $4^k = 3x + 1$ then $4^{k+1} = 4(3x+1) = 13x + 1.$

Example: Binary numerals divisible by 3

- Consider every string $w \in \{\mathtt{0},\mathtt{1}\}^*$ to be a binary numerals.
- The ***numeric value*** $[w]_2$ of a string $w = d_k d_{k-1} \cdots d_0$ is $\Sigma_i\ 2^i$.
- Problem: Construct a DFA over $\{\mathtt{0},\mathtt{1}\}^*$ that accepts the numerals divisble by 3.
- Preliminary: What is the value mod(3) of the digits, i.e. what is 2^k mod(3).

We have that $4^k =_3 1$, by induction on k.

So $2^{2k} = 3x + 1$ for some x , and $2^{2k+1} = 2(3x+1) = 6x+2$.

$\therefore$ $2^n =_3 1$ for even n , and $=_3 2$ for odd n .

Example: Binary numerals divisible by 3

- For any input w the expectation depends on the parity of $|w|$, the goals are therefore of the form

  Either $|w|$ is even and $[w] =_3 x$ or $|w|$ is odd and $[w] =_3 y$

  Let's abbreviate this as (x, y).

Example: Binary numerals divisible by 3

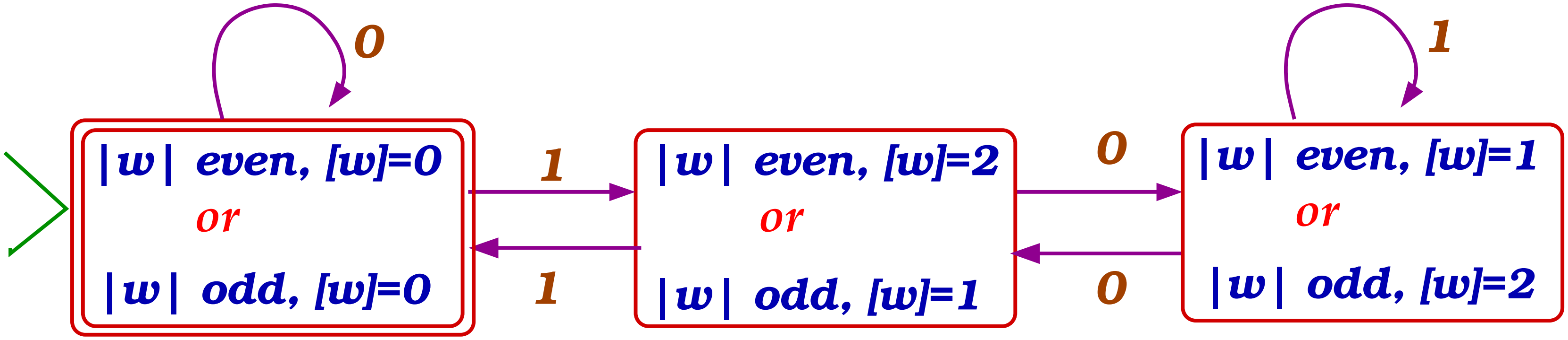
- For any input w the expectation depends on the parity of $|w|$, the goals are therefore of the form

  Either $|w|$ is even and $[w] =_3 x$ or $|w|$ is odd and $[w] =_3 y$

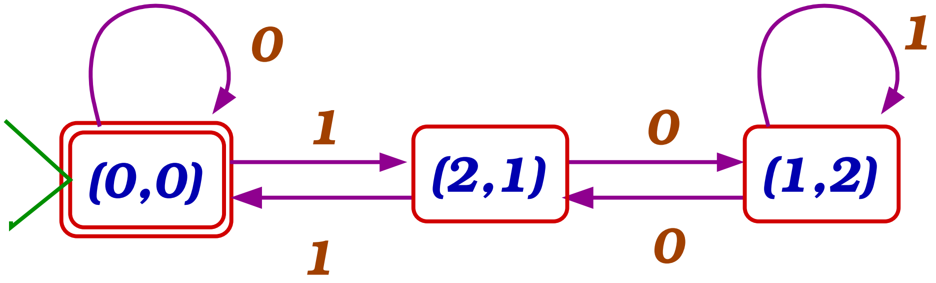
  Let's abbreviate this as (x, y).

- From the observation above it follows that $(x,\ y) \xrightarrow{1} (y{+}2,\ x{+}1)$, and $(x,\ y) \xrightarrow{0} (y,\ x)$.

- This yields the following DFA:



Condensed:



RESIDUES AND THEIR APPLICATIONS

More examples of residues

- Take $L =$ English words.

  $L/\mathtt{invent}$ contains the strings $\mathtt{or}, \mathtt{ion}, \mathtt{ive}, \mathtt{ed}$ and $\mathtt{ing}$
  since $\mathtt{inventor}, \mathtt{invention}, \mathtt{inventive}$ and $\mathtt{invented}$ are words.

- ϵ is also in $L/\mathtt{invent}$ since invent is a word.

- The residue $L/\mathtt{ad}$ contains the strings $\mathtt{vance}, \mathtt{apt}, \mathtt{opt}, \mathtt{d},$ and ϵ.

- Take $L = \{\mathsf{ab}\}$, a singleton language.
  We have $L/\varepsilon = \{\mathsf{ab}\}, L/\mathsf{a} = \{\mathsf{b}\}$, and $L/\mathsf{ab} = \varepsilon$.

  For any other string w, $L/w = \emptyset$.

- For any language L we have $L/\varepsilon = L$:

  $w \in L$ iff $\varepsilon \in L/w$.

More examples yet

- $L = \{0,\ 00,\ 010\}$

$$\begin{aligned} L/\varepsilon &= L \\ L/0 &= \{\varepsilon,\ 0, 10\} \\ L/00 &= \{\varepsilon\} \\ L/01 &= \{0\} \\ L/010 &= \{\varepsilon\} \\ L/w &= \emptyset \text{for any other} w \end{aligned}$$

$L/00 = L/010$, so there are five (different) residues.

An example with language union

- $L = \{\mathtt{a}w \mid w \in \Sigma^*\} \cup \{\mathtt{baa}\}$.

$$\begin{array}{rcll} L/\varepsilon & = & L & \\ L/w & = & \Sigma^* & \text{if } w \text{ starts with } \mathtt{a} \\ L/\mathtt{b} & = & \{\mathtt{aa}\} & \\ L/\mathtt{ba} & = & \{\mathtt{a}\} & \\ L/\mathtt{baa} & = & \{\varepsilon\} & \\ L/w & = & \emptyset & \text{for any other } w \end{array}$$

There are 6 residues.

L and Σ^* are infinite languages, the others are finite.

A single-letter language

- $\Sigma = \{0, 1\}$, $L = \{0\}^*$.

- If $w \in \Sigma^*$ contains 1 then $L/w = \emptyset$.
  Otherwise $L/w = L$.
  There are two residues.

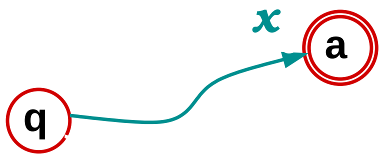
A language based on occurrence count

- $L = \{w \in \{0, 1\} \mid \#_0(w)$ is even $\}$.

  If $\#_0(w)$ is even then L/w is L,
  otherwise $L/w = \{w \mid \#_0(w)$ is odd $\}$

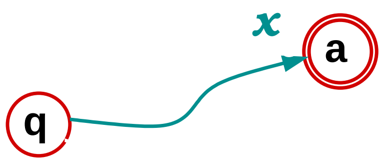
Each state determines a language

- Consider a DFA M recognizing L and a state q in it.
  Some string x may lead from q to acceptance.

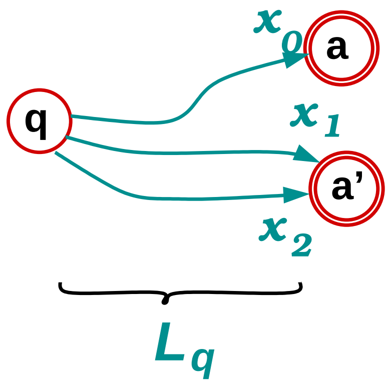


Each state determines a language

- Consider a DFA M recognizing L and a state q in it.
  Some string x may lead from q to acceptance.

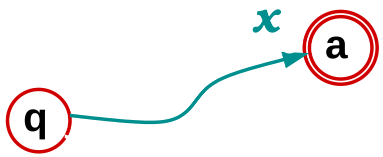


- Denote the set of all such x's by L_q.
  In particular, $L_s = L$.

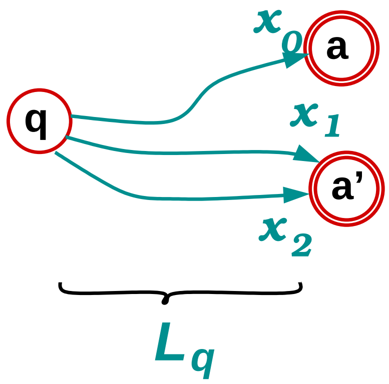


Each state determines a language

- Consider a DFA M recognizing L and a state q in it.
  Some string x may lead from q to acceptance.



- Denote the set of all such x's by L_q.
  In particular, $L_s = L$.



- Note: We focus on the future of q , not its past!
  (The past would be the set of strings leading to q)

States and residues

- Now suppose that $s \xrightarrow{w} q$.

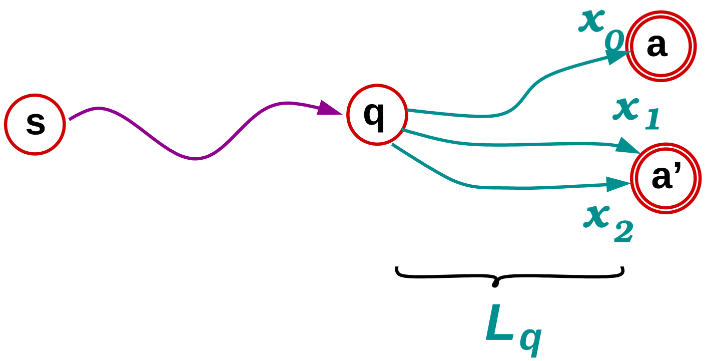
  A string $w \cdot x$ is accepted by M iff $x \in L_q$.

States and residues

- Now suppose that $s \xrightarrow{w} q$.

  A string $w \cdot x$ is accepted by M iff $x \in L_q$.

- x completes w to a string in L:

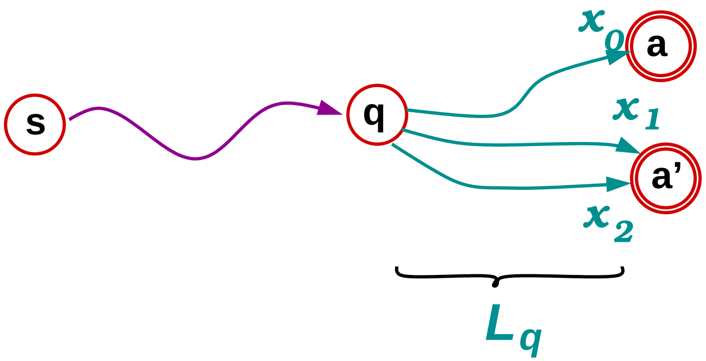


States and residues

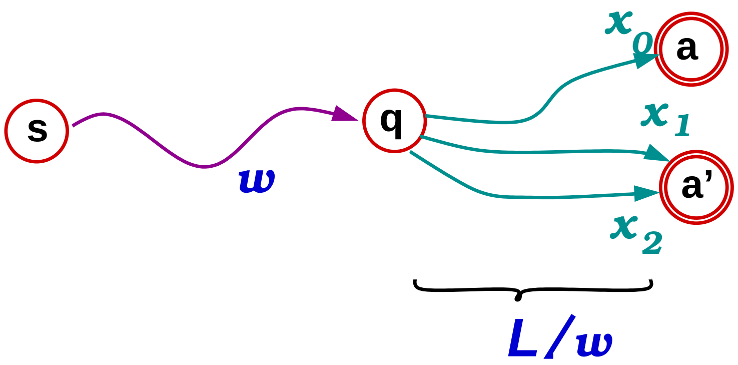
- Now suppose that $s \xrightarrow{w} q$.

  A string $w \cdot x$ is accepted by M iff $x \in L_q$.

- x completes w to a string in L:



- L_q is $L/w =$ the residue of L over w:



A property of recognized languages

- **Theorem.** (Myhill-Nerode) ***A language recognized by a k-state DFA has $\leqslant k$ residues.***

A property of recognized languages

- **Theorem.** (Myhill-Nerode) *A language recognized by a k-state DFA has $\leqslant k$ residues.*
- Proof. If $s \xrightarrow{u} q$ and $s \xrightarrow{v} q$ then $L/u = L/v$.

A property of recognized languages

- **Theorem.** (Myhill-Nerode) ***A language recognized by a k-state DFA has $\leqslant k$ residues.***
- Proof. If $s \xrightarrow{u} q$ and $s \xrightarrow{v} q$ then $L/u = L/v$.
- Consequently:
  Theorem.
  A language with infinitely many residues is not recognized.

Languages with infinitely many residues

- Let $L = \{w \in \{0, 1\}^* \mid \#_0(w) = \#_1(w)\}$.

Languages with infinitely many residues

- Let $L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#_0(w) = \#_1(w)\}$.
- Consider the residues of L the form $L/\mathtt{1}^n \quad (n \geqslant 0)$.

Languages with infinitely many residues

- Let $L = \{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$.
- Consider the residues of L the form $L/1^n \quad (n \geqslant 0)$.
- For each n we have
$$L/1^n = \{x \mid \#_0(x) = \#_1(x) + n\} \ ,$$
since to compensate for an initial substring of n 1's the rest of the string should have n extra 0's.

Languages with infinitely many residues

- Let $L = \{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$.
- Consider the residues of L the form $L/1^n \quad (n \geqslant 0)$.
- For each n we have
  $$L/1^n = \{x \mid \#_0(x) = \#_1(x) + n\} \text{ ,}$$
  since to compensate for an initial substring of n 1's the rest of the string should have n extra 0's.
- If $i \neq j$ then $0^i \in L/1^i$ but $\notin L/1^j$ so the two residues are ***different***.

Languages with infinitely many residues

- Let $L = \{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$.
- Consider the residues of L the form $L/1^n \quad (n \geqslant 0)$.
- For each n we have

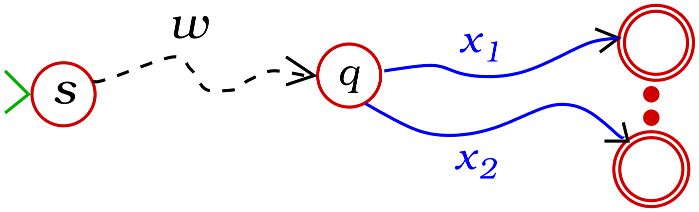
  $$L/1^n = \{x \mid \#_0(x) = \#_1(x) + n\}\ ,$$

  since to compensate for an initial substring of n 1's the rest of the string should have n extra 0's.
- If $i \neq j$ then $0^i \in L/1^i$ but $\notin L/1^j$ so the two residues are ***different***.

$\therefore$ L is not recognized, since it has infinitely many residues.

States and residues

- We developed automata by thinking of residues as states.
- Let M be an automaton over Σ.
  For a state q of M define
  $$L_q =_{\mathrm{df}} \{x \in \Sigma^* \mid q \xrightarrow{x} A\}$$
- In particular, for the start state $L_s = L$.
- If $s \xrightarrow{w} q$ then $L_q = L/w$.

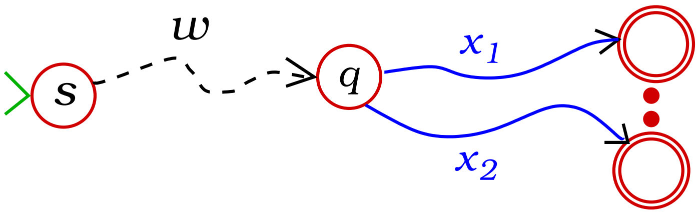


$L_q = L / w$

⋆ Each string leads from s to some state.

⋆ All strings leading from s to a state q have the same residue.

The Myhill-Nerode Theorem



$L_q = L / w$

- Every residue L/w is L_q for q as above.
- And two different residues $L/w \neq L/x$ must correspond to two different states.
- So we have an injection that maps residues to states,
  I.e. the number of residues is bounded by the number of states.
- **Theorem.** (John Myhill and Anil Nerode (1958)) (simplified and rephrased):
  $\mathcal{L}(M)$ cannot have more residues than M has states.
- Consequence: *A language with infinitely many residues cannot be recognized by any automaton!*

Showing that a language fails recognition

- We saw that $L = \{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$ has infinitely many residues.
- Consequence: It cannot be recognized by any automaton!!!
- General method: show that L is not recognized by showing that there are infinitely many residues.
- We do not need to consider all residues, ***only some infinite selection, defined by a template***
- We ***do not need to calculate*** the residues we choose, ***only show that each two of them are different***.
- We show them different by exhibiting a string which is in one but not in the other.

Example: Unary addition

- Representing unary addition, using unary numerals and the symbols for addition and equality:
- $L = \{\mathtt{1}^k + \mathtt{1}^m = \mathtt{1}^{k+m} \mid k, m \geqslant 1\}$
- What residues would you select?

- $L/\,\mathtt{1}^n + \mathtt{1} =$ for each $n \geqslant 1$.
- Suppose $i \neq j$.
  What string is in $L/\,\mathtt{1}^i + \mathtt{1} =$ but not in $L/\,\mathtt{1}^j + \mathtt{1} =$?

Example: Residues for Mahimahi

- Consider $L = \{u \cdot u \mid u \in \{\mathtt{0}, \mathtt{1}\}^* \}$.
  What residues L/w to take?

Example: Residues for Mahimahi

- Consider $L = \{u \cdot u \mid u \in \{\mathtt{0},\mathtt{1}\}^* \}$.
  What residues L/w to take?
- w with an end-mark would help with differentiating residues.
  Say $\mathtt{0}^n\mathtt{1}$?

Example: Residues for Mahimahi

- Consider $L = \{u \cdot u \mid u \in \{0,1\}^* \}$.
  What residues L/w to take?
- w with an end-mark would help with differentiating residues.
  Say 0^n1?
- Then $0^i1 \in L/0^i1$,
  but for $j > i$ we have $0^i1 \notin L/0^j1$,
  because it has two 1's in its first half and none in the second.

Example: Residues for Mahimahi

- Consider $L = \{u \cdot u \mid u \in \{0,1\}^* \}$.
  What residues L/w to take?

- w with an end-mark would help with differentiating residues.
  Say 0^n1?

- Then $0^i1 \in L/0^i1$,
  but for $j > i$ we have $0^i1 \notin L/0^j1$,
  because it has two 1's in its first half and none in the second.

- Since each two of these residues are different,
  L has infinitely many residues,
  and cannot be recognized by a DFA.

Example: Residues for perfect squares

- $L = \{\mathtt{1}^{n^2} \mid n \geqslant 0\}$.
- Consider the residues $L/\mathtt{1}^{n^2}$ for each $n > 0$.
- The first perfect square following n^2 is $(n{+}1)^2 = n^2 + 2n + 1$.
- So the shortest non-null string of $L/\mathtt{1}^{i^2}$ is $\mathtt{1}^{2i+1}$.
- It follows that $\mathtt{1}^{2i+1} \in L/\mathtt{1}^{i^2}$
  but $\mathtt{1}^{2i+1} \notin L/\mathtt{1}^{j^2}$ for any $j > i$.
- Since every two of these residues are different,
  L has infinitely many residues,
  and cannot be recognized by any automaton.

Building automata directly from residues

- We showed that every recognized language has finitely many residues.
- The converse is also true:
- If $L \subseteq \Sigma^*$ has finitely many residues, then $L = \mathcal{L}(M)$ where:
  - ⋆ The states of M are the residues.
  - ⋆ The initial state is $L/\varepsilon = L$.
  - ⋆ A state L/w is accepting iff it contains ε.
  - ⋆ The transitions are given by $L/w \quad \xrightarrow{\sigma} \quad L/w\sigma$.
- We used the same idea to construct automata, except that here we assume that the residues are given to us.
- We write *Res*(L) for the automaton constructed from residues.

Recognized = finitely many residues

- A language L is recognized iff it has finitely many residues.
- The DFA constructed from L's residues
  has the fewer states
- Given a DFA M recognizing L, and a state q,

Minimizing an automaton: Rationale

- Suppose M is a k-state DFA over Σ, recognizing L.

  For each accessible state q the language L_q is a residue of L. If M is the smallest automaton recognizing L

  then these residues are all different.

Minimizing an automaton: Rationale

- Suppose M is a k-state DFA over Σ, recognizing L.
  For each accessible state q the language L_q is a residue of L. If M is the smallest automaton recognizing L
  then these residues are all different.
- M might be constructed using residues as states
  and yet not be minimal, because the same residue might have been introduced twice for different property descriptions.

Minimizing an automaton: Rationale

- Suppose M is a k-state DFA over Σ, recognizing L.
  For each accessible state q the language L_q is a residue of L. If M is the smallest automaton recognizing L
  then these residues are all different.

- M might be constructed using residues as states
  and yet not be minimal, because the same residue might have been introduced twice for different property descriptions.

  But when M is not minimal we can still obtain
  a minimal automaton by identifying duplicates and unifying them.

Minimizing an automaton: Separating residues

- Say that a string x ***separates*** q from q'
  if x is in one of L_q and $L_{q'}$ but not in the other.
  That is, x is a witness for $L_q \neq L_{q'}$.
- Write $q \mathrm{D} q'$ if there is such an x ,
  i.e. L_q and $L_{q'}$ are different.
- Write $q \mathrm{D}_n q'$ if q is separated from q'
  by some string of length $\leqslant n$.

Minimizing an automaton: Separating residues

- Say that a string x ***separates*** q from q' if x is in one of L_q and $L_{q'}$ but not in the other. That is, x is a witness for $L_q \neq L_{q'}$.
- Write $q \mathrm{D} q'$ if there is such an x , i.e. L_q and $L_{q'}$ are different.
- Write $q \mathrm{D}_n q'$ if q is separated from q' by some string of length $\leqslant n$.
  - Note: $\mathrm{D}_{n+1} \supseteq \mathrm{D}_n$.
  - Let's show that if $\mathrm{D}_{n+1} = \mathrm{D}_n$ then $\mathrm{D}_{n+2} = \mathrm{D}_{n+1}$

Minimizing an automaton: Bounding the separator

- Suppose $q \ \mathrm{D}_{n+2} q'$, i.e. some σx of length $n{+}2$ separates between q and q' .
  Let $q \xrightarrow{\sigma} p$ and $q' \xrightarrow{\sigma} p'$.
  Then x separates between p and p' , so $pdm_{n+1}p'$.
- But we assume $\mathrm{D}_{n+1} = \mathrm{D}_n$, so $p \ \mathrm{D}_n p'$,
  and therefore $q \ \mathrm{D}n+1q'$.

Minimizing an automaton: Bounding the separator

- Suppose $q\ \mathrm{D}_{n+2} q'$, i.e. some σx of length $n{+}2$ separates between q and q' .
  Let $q \xrightarrow{\sigma} p$ and $q' \xrightarrow{\sigma} p'$.
  Then x separates between p and p' , so $pdm_{n+1}p'$.
- But we assume $\mathrm{D}_{n+1} = \mathrm{D}_n$, so $p\ \mathrm{D}_n p'$,
  and therefore $q\ \mathrm{D}n + 1q'$.
- By induction, if $\mathrm{D}_{n+1} = \mathrm{D}_n$ then $\mathrm{D}_i = \mathrm{D}_n$ for all $i \geqslant n$, and so $\mathrm{D}_n = \mathrm{D}$.

Minimizing an automaton: Bounding the separator

- Suppose $q\ \mathrm{D}_{n+2}q'$, i.e. some σx of length $n{+}2$ separates between q and q' .
  Let $q \xrightarrow{\sigma} p$ and $q' \xrightarrow{\sigma} p'$.
  Then x separates between p and p' , so $pdm_{n+1}p'$.
- But we assume $\mathrm{D}_{n+1} = \mathrm{D}_n$, so $p\ \mathrm{D}_n p'$,
  and therefore $q\ \mathrm{D}n + 1q'$.
- By induction, if $\mathrm{D}_{n+1} = \mathrm{D}_n$ then $\mathrm{D}_i = \mathrm{D}_n$ for all $i \geqslant n$, and so $\mathrm{D}_n = \mathrm{D}$.
- Conclusion: For some n $\mathrm{D}_0 \subset \mathrm{D}_1 \subset \mathrm{D}_2 \subset \cdots \subset \mathrm{D}_n = \mathrm{D}_{n+1} = \mathrm{D}_n$
  where $n \leqslant$ the number of pairs of distinct states.
  i.e. $\ell = k(k-1)/2$.

Minimizing an automaton: Bounding the separator

- Suppose $q\ \mathrm{D}_{n+2} q'$, i.e. some σx of length $n+2$ separates between q and q' .
  Let $q \xrightarrow{\sigma} p$ and $q' \xrightarrow{\sigma} p'$.
  Then x separates between p and p' , so $pdm_{n+1}p'$.
- But we assume $\mathrm{D}_{n+1} = \mathrm{D}_n$, so $p\ \mathrm{D}_n p'$,
  and therefore $q\ \mathrm{D}n+1q'$.
- By induction, if $\mathrm{D}_{n+1} = \mathrm{D}_n$ then $\mathrm{D}_i = \mathrm{D}_n$ for all $i \geqslant n$, and so $\mathrm{D}_n = \mathrm{D}$.
- Conclusion: For some n $\mathrm{D}_0 \subset \mathrm{D}_1 \subset \mathrm{D}_2 \subset \cdots \subset \mathrm{D}_n = \mathrm{D}_{n+1} = \mathrm{D}_n$
  where $n \leqslant$ the number of pairs of distinct states.
  i.e. $\ell = k(k-1)/2$.
- The stable D_n is the relation $L_q \neq L'_q$ between states.

Minimizing an automaton: Bounding the separator

- Suppose $q\ \mathrm{D}_{n+2}q'$, i.e. some σx of length $n+2$ separates between q and q' .
  Let $q \xrightarrow{\sigma} p$ and $q' \xrightarrow{\sigma} p'$.
  Then x separates between p and p' , so $pdm_{n+1}p'$.
- But we assume $\mathrm{D}_{n+1} = \mathrm{D}_n$, so $p\ \mathrm{D}_n p'$,
  and therefore $q\ \mathrm{D}n+1q'$.
- By induction, if $\mathrm{D}_{n+1} = \mathrm{D}_n$ then $\mathrm{D}_i = \mathrm{D}_n$ for all $i \geqslant n$, and so $\mathrm{D}_n = \mathrm{D}$.
- Conclusion: For some n $\mathrm{D}_0 \subset \mathrm{D}_1 \subset \mathrm{D}_2 \subset \cdots \subset \mathrm{D}_n = \mathrm{D}_{n+1} = \mathrm{D}_n$
  where $n \leqslant$ the number of pairs of distinct states.
  i.e. $\ell = k(k-1)/2$.
- The stable D_n is the relation $L_q \neq L'_q$ between states.
- Conclusion: If $q\ \mathrm{D}q'$ then q, q' are separated
  by a string of length $\leqslant k(k-1)/2$.

Minimization algorithm for DFAs

Outline of a ***minimization algorithm***:
Given a k -state DFA M recognizing L :

1. For each pair q, q' determine if $L_q = L'_q$ by checking all strings of length $k(k-1)/2$.

Minimization algorithm for DFAs

Outline of a ***minimization algorithm***:
Given a k -state DFA M recognizing L :

1. For each pair q, q' determine if $L_q = L'_q$ by checking all strings of length $k(k-1)/2$.
2. Obtain the minimal DFA recognizing L by unifying equivalent states.

MODIFYING & COMBINING AUTOMATA

Partial-automata

- A ***partial-automaton*** is an automaton whose transition mapping is a ***partial*** function (recall that a total-function is also a partial-function).

Partial-automata

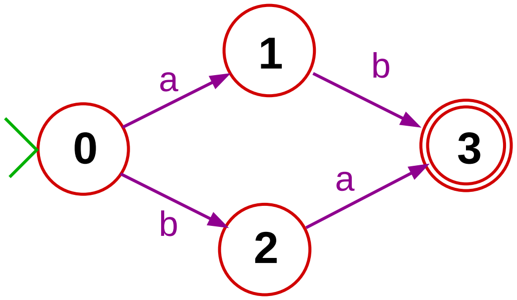
- A ***partial-automaton*** is an automaton whose transition mapping is a ***partial*** function (recall that a total-function is also a partial-function).
- A partial-automaton M terminates execution when it cannot proceed: no applicable transition (due to partiality) or no next-letter to move to.

  It ***accepts*** w if its state-trace for w ends with an accepting state.

Partial-automata

- A ***partial-automaton*** is an automaton whose transition mapping is a ***partial*** function (recall that a total-function is also a partial-function).
- A partial-automaton M terminates execution when it cannot proceed: no applicable transition (due to partiality) or no next-letter to move to.

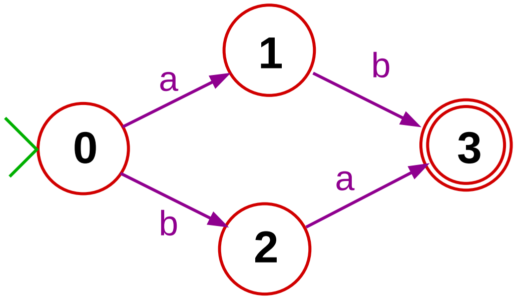
  It ***accepts*** w if its state-trace for w ends with an accepting state.
- Example: A partial automaton recognizing $\{\mathtt{ab}, \mathtt{ba}\}$:



Partial-automata

- A ***partial-automaton*** is an automaton whose transition mapping is a ***partial*** function (recall that a total-function is also a partial-function).
- A partial-automaton M terminates execution when it cannot proceed: no applicable transition (due to partiality) or no next-letter to move to.

  It ***accepts*** w if its state-trace for w ends with an accepting state.
- Example: A partial automaton recognizing $\{\mathtt{ab}, \mathtt{ba}\}$:



- Some people use “automaton” for our “partial-automaton” and “total-automaton” for our “automaton.”

From partial- to total-automaton

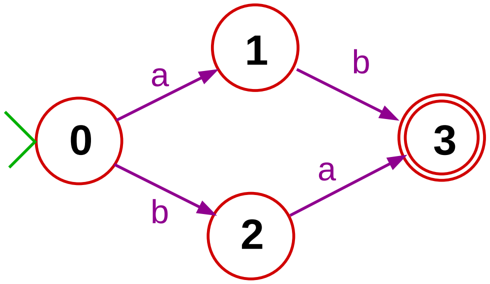
- **Theorem.** Every partial-automaton M can be converted into a total-automaton $\bar{M}$ equivalent to M, i.e. recognizing the same language.

  Do you seee how?

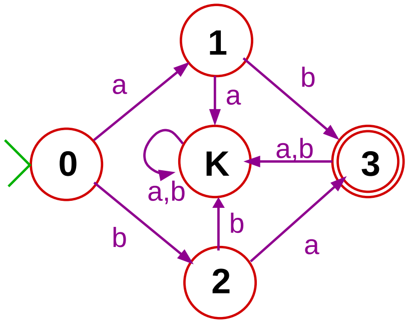
From partial- to total-automaton

- **Theorem.** Every partial-automaton M can be converted into a total-automaton $\bar{M}$ equivalent to M, i.e. recognizing the same language.
  Do you seee how?
- Just add a sink to M :

convert



to

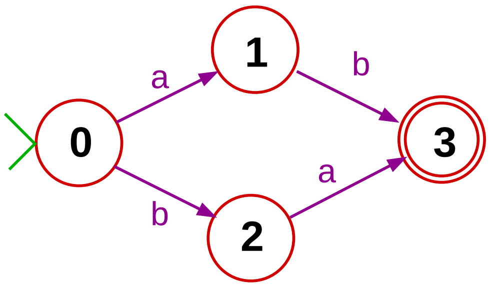


From partial- to total-automaton

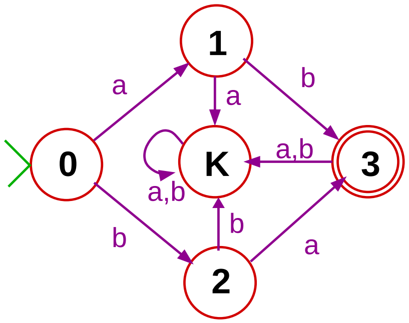
- **Theorem.** Every partial-automaton M can be converted into a total-automaton $\bar{M}$ equivalent to M, i.e. recognizing the same language.
  Do you seee how?

- Just add a sink to M :

convert



to



- That is, $\bar{M}$ is obtained by adding to M
  a sink state K , with all missing transitions of M
  as well as outgoing transition from K , pointing to K .

Example: Equiping strings with start signal

- $M = (\Sigma, Q, s, A, \delta)$ is a partial-automaton recognizing L .
  Convert M to M' recognizing $\mathtt{a} \cdot L$.
  ($\mathtt{a}$ can be construed as a start-signal.

Example: Equiping strings with start signal

- $M = (\Sigma, Q, s, A, \delta)$ is a partial-automaton recognizing L .
  Convert M to M' recognizing $\mathtt{a} \cdot L$.
  ($\mathtt{a}$ can be construed as a start-signal.

  Fix some $t \notin Q$ and let M' be
  M augmented with t as the new start state,
  and the transition $q \xrightarrow{a} s)$

Example: Equiping strings with end signal

- Let $\Box \notin \Sigma$.
  Convert M to M'' recognizing $L \cdot \Box$.

Example: Equiping strings with end signal

- Let $\square \notin \Sigma$.

  Convert M to M'' recognizing $L \cdot \square$.

  Let M'' be M with z the accepting state,
  augmented with the transitions $a \xrightarrow{\square} z$ for each $a \in A$.

Example: Equiping strings with end signal

- Let $\square \notin \Sigma$.

  Convert M to M'' recognizing $L \cdot \square$.

  Let M'' be M with z the accepting state,
  augmented with the transitions $a \xrightarrow{\square} z$ for each $a \in A$.

  This construction won’t work if $\square \in \Sigma$, why?

The complement of a recognized language

- **Theorem.** *If* $L \subseteq \Sigma^*$ *is recognized then so is* $\bar{L} = \Sigma^* - L$.

The complement of a recognized language

- **Theorem.** *If $L \subseteq \Sigma^*$ is recognized then so is $\bar{L} = \Sigma^* - L$.*

  The proof is another example of manipulating automata:

  An automaton recognizing L is converted into one for $\bar{L}$.

The complement of a recognized language

- **Theorem.** *If* $L \subseteq \Sigma^*$ *is recognized then so is* $\bar{L} = \Sigma^* - L$*.*

  The proof is another example of manipulating automata:
  An automaton recognizing L is converted into one for $\bar{L}$.

- Given DFA M , how do you get a DFA $\bar{M}$
  that accepts when M rejects, and rejects when M accepts?

The complement of a recognized language

- **Theorem.** *If $L \subseteq \Sigma^*$ is recognized then so is $\bar{L} = \Sigma^* - L$.*

  The proof is another example of manipulating automata:

  An automaton recognizing L is converted into one for $\bar{L}$.

- We simply intechange accepting and non-accepting states.

The complement of a recognized language

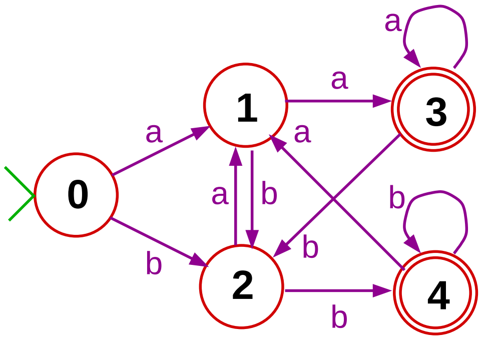
- **Theorem.** *If $L \subseteq \Sigma^*$ is recognized then so is $\bar{L} = \Sigma^* - L$.*

  The proof is another example of manipulating automata:

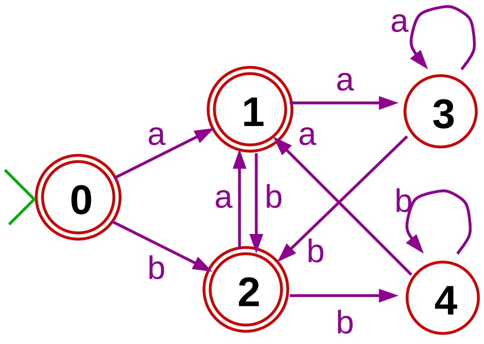
  An automaton recognizing L is converted into one for $\bar{L}$.

- We simply intechange accepting and non-accepting states.

  For example, the automaton recognizing $\{w\sigma\sigma \mid w \in \Sigma^*, \sigma \in \Sigma\}$



is converted to



which accepts the strings of length < 2 and the strings ending with two different letters.

Application: Additional languages recognized

- Suppose M recognizes $\{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \#_a(w) = \#_b(w) \mod 2\}$.
- Then swapping states in M yields an automaton recognizing

$$\{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \#_a(w) \neq \#_b(w) \mod 2\}$$

Application: Showing a language not-recognized

- Show that $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \#_a(w) \neq \#_b(w)\}$ is not recognized.

Application: Showing a language not-recognized

- Show that $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \#_a(w) \neq \#_b(w)\}$ is not recognized.
- Clipping doesn't work!

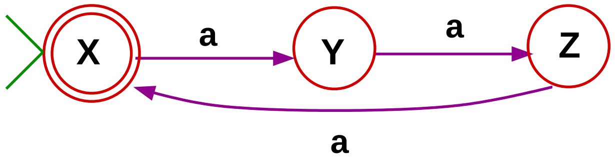
Application: Showing a language not-recognized

- Show that $L = \{w \in \{\mathtt{a},\mathtt{b}\}^* \mid \#_a(w) \neq \#_b(w)\}$ is not recognized.
- Clipping doesn't work!
- Use Clipping to show that the complement
  $\bar{L} = \{w \in \{\mathtt{a},\mathtt{b}\}^* \mid \#_a(w) = \#_b(w)\}$ is not recognized.
- Conclude: L is not recognized, or else $\bar{L}$ would be.

Combining two automata

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

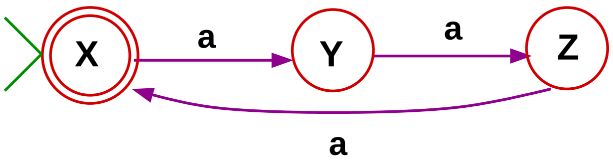
- Suppose M_3 recognizes $L_3 = \{w \in \Sigma^* \mid \#_a(w) = 0 \mod (3)\ \}$



Combining two automata

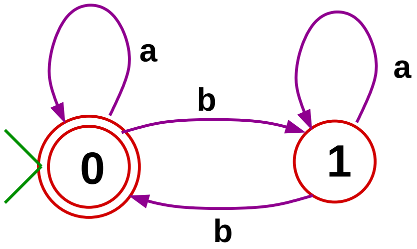
Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

- Suppose M_3 recognizes $L_3 = \{w \in \Sigma^* \mid \#_a(w) = 0 \mod (3)\}$



and

- M_2 recognizes $L_2 = \{w \in \Sigma^* \mid \#_b(w) = 0 \mod (2)\}$.

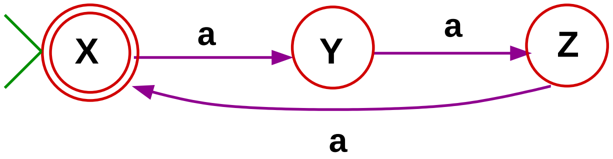


$\#_b w = 0 \mod 2$

Combining two automata

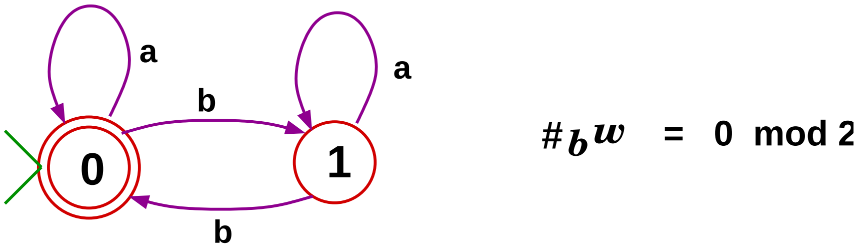
Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

- Suppose M_3 recognizes $L_3 = \{w \in \Sigma^* \mid \#_a(w) = 0 \mod (3)\ \}$



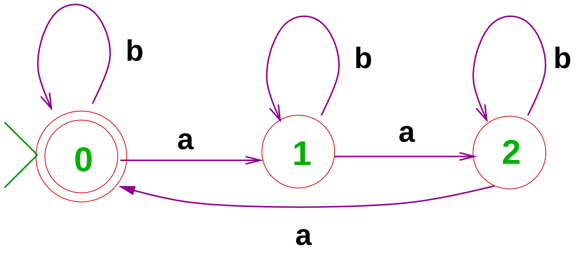
and

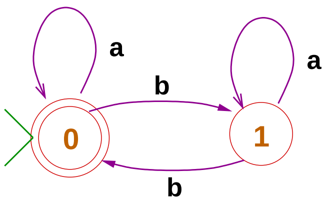
- M_2 recognizes $L_2 = \{w \in \Sigma^* \mid \#_b(w) = 0 \mod (2)\ \}$.

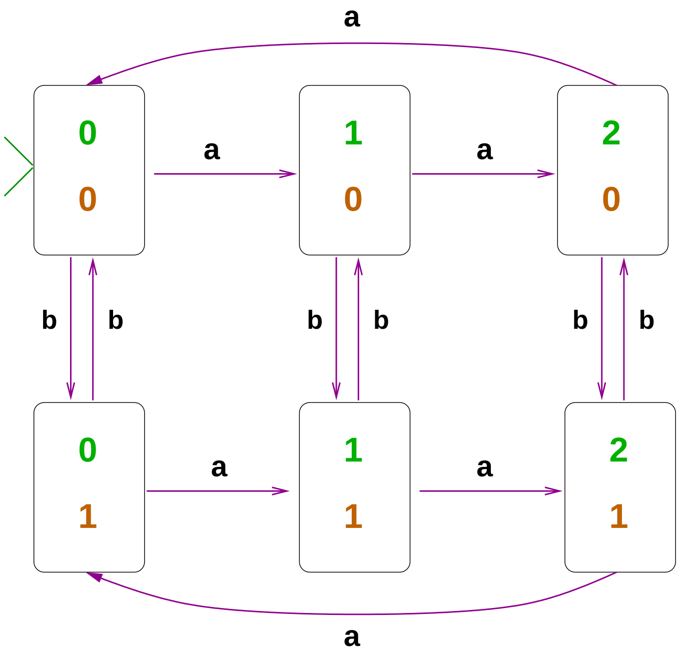


Parallel programming is tricky, but here we have a special form of parallelism: the two processors may work in tandem, because they read the same input one symbol at a time.

Two automata collaborating

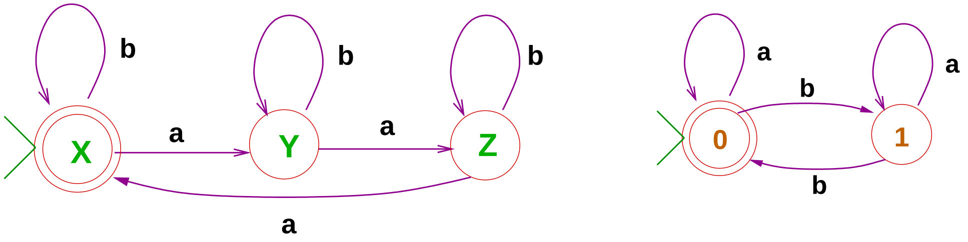


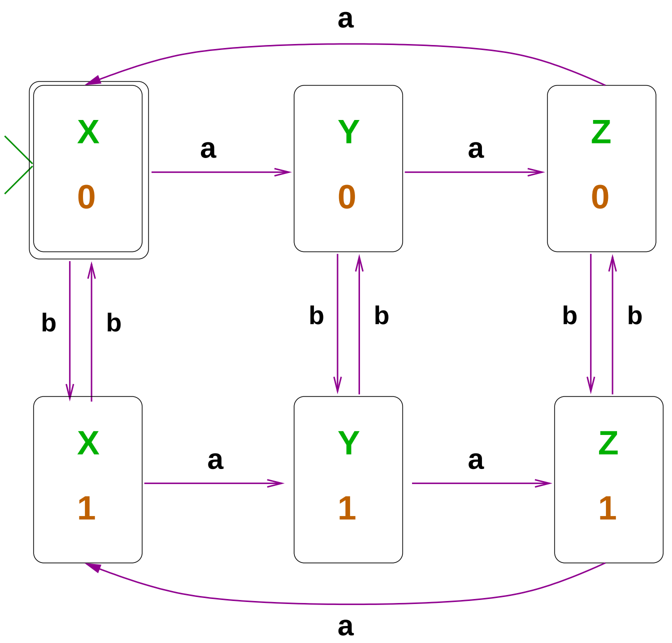




Conjuctive pairing

- Accepting when both accept:

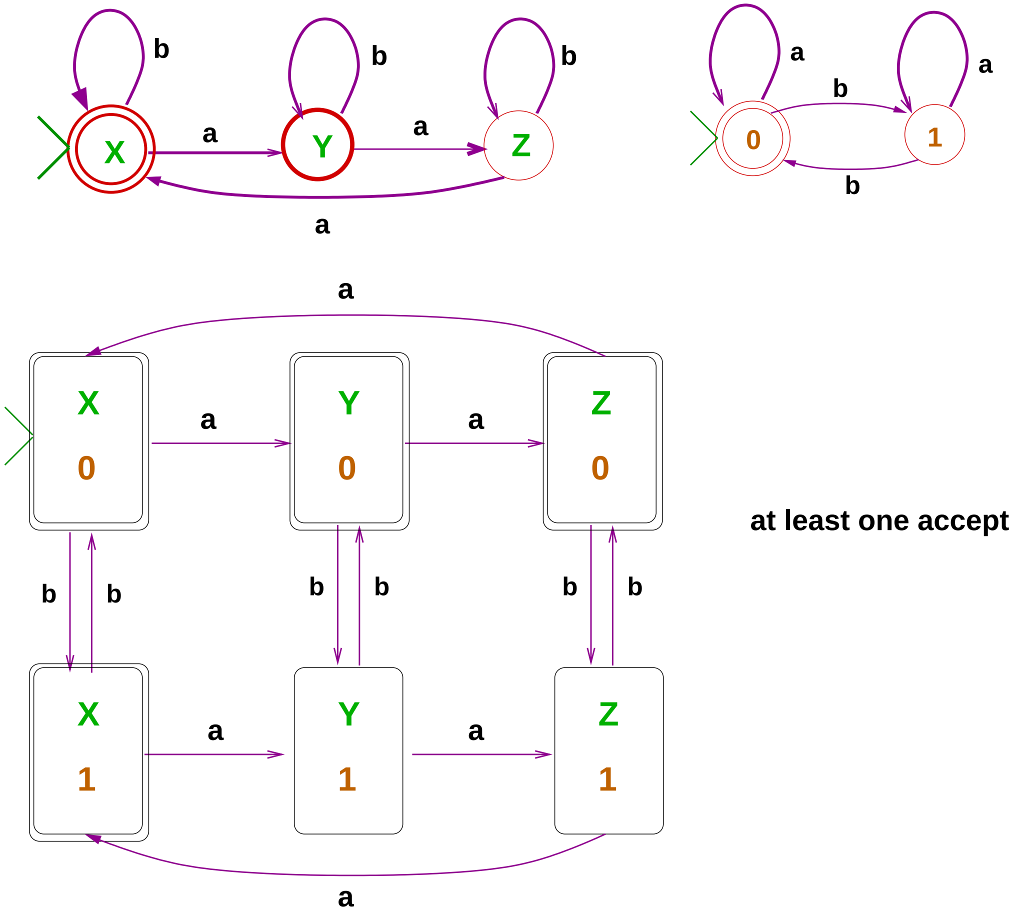




both accept

Disjunctive pairing

- Accepting when at least one automaton accepts:



at least one accepts

Formal definition of automata product

- Given automata $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q', s', A', \delta')$ consider a ***coupling***:

Formal definition of automata product

- Given automata $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q', s', A', \delta')$ consider a ***coupling***:
  - States are pairs $\langle q, q' \rangle$ where $q \in Q$ and $q' \in Q'$.
    I.e. the set of states is $Q \times Q'$.
  - The initial state is $\langle s, s' \rangle$.

Formal definition of automata product

- Given automata $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q', s', A', \delta')$ consider a ***coupling***:
  - States are pairs $\langle q, q' \rangle$ where $q \in Q$ and $q' \in Q'$.
    I.e. the set of states is $Q \times Q'$.
  - The initial state is $\langle s, s' \rangle$.
  - The transitions are $\langle q, q' \rangle \xrightarrow{\sigma} \langle p, p' \rangle$ where $q \xrightarrow{\sigma} p$ in M and $q' \xrightarrow{\sigma} p'$ in M'.

Formal definition of automata product

- Given automata $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q', s', A', \delta')$ consider a ***coupling***:

  - ► States are pairs $\langle q, q' \rangle$ where $q \in Q$ and $q' \in Q'$. I.e. the set of states is $Q \times Q'$.
  - ► The initial state is $\langle s, s' \rangle$.
  - ► The transitions are $\langle q, q' \rangle \xrightarrow{\sigma} \langle p, p' \rangle$ where $q \xrightarrow{\sigma} p$ in M and $q' \xrightarrow{\sigma} p'$ in M'.

  – In a ***conjunctive product*** the set of accepting states is $A \times A'$ (both automata accept).

Formal definition of automata product

- Given automata $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q', s', A', \delta')$ consider a ***coupling***:
  - States are pairs $\langle q, q' \rangle$ where $q \in Q$ and $q' \in Q'$. I.e. the set of states is $Q \times Q'$.
  - The initial state is $\langle s, s' \rangle$.
  - The transitions are $\langle q, q' \rangle \xrightarrow{\sigma} \langle p, p' \rangle$ where $q \xrightarrow{\sigma} p$ in M and $q' \xrightarrow{\sigma} p'$ in M'.

– In a ***conjunctive product*** the set of accepting states is $A \times A'$ (both automata accept).

– In a ***disjunctive product*** the set of accepting states is $(A \times Q') \cup (Q \times A')$ (at least one automaton accepts).

Some applications

- $L = \{\ \mathtt{a}\, w\, \mathtt{z}\ \mid\ w \in \Sigma^*\ \}$

Some applications

- $L = \{\ \mathtt{a}\, w \mathtt{z}\ \mid\ w \in \Sigma^*\ \}$
- $\{\mathtt{a}^p\mathtt{b}^q\ \mid\ p$ is odd $\}$.

Some applications

- $L = \{ \mathtt{a}\, w \mathtt{z} \mid w \in \Sigma^* \}$
- $\{\mathtt{a}^p\mathtt{b}^q \mid p$ is odd $\}$.
- An automaton over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ recognizingthe string that miss at least one letter.

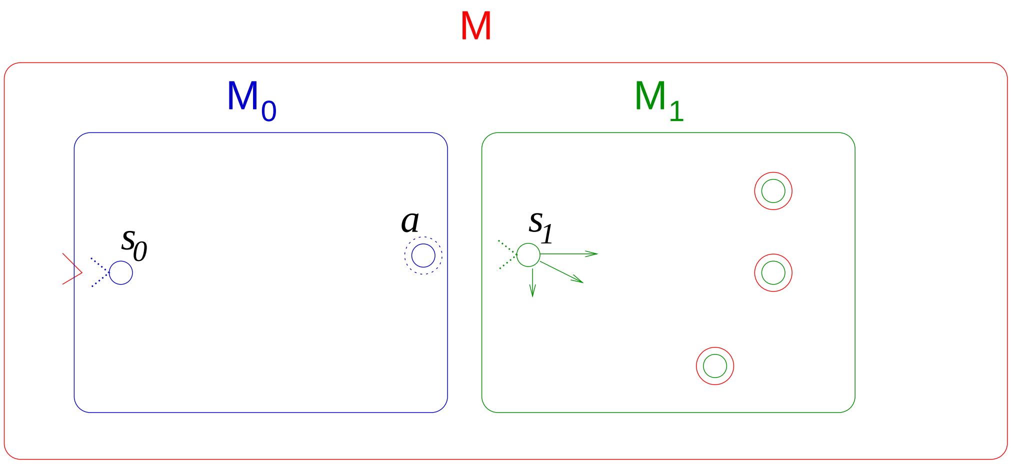
Nondeterministic Automata

Capturing operationally language concatenation

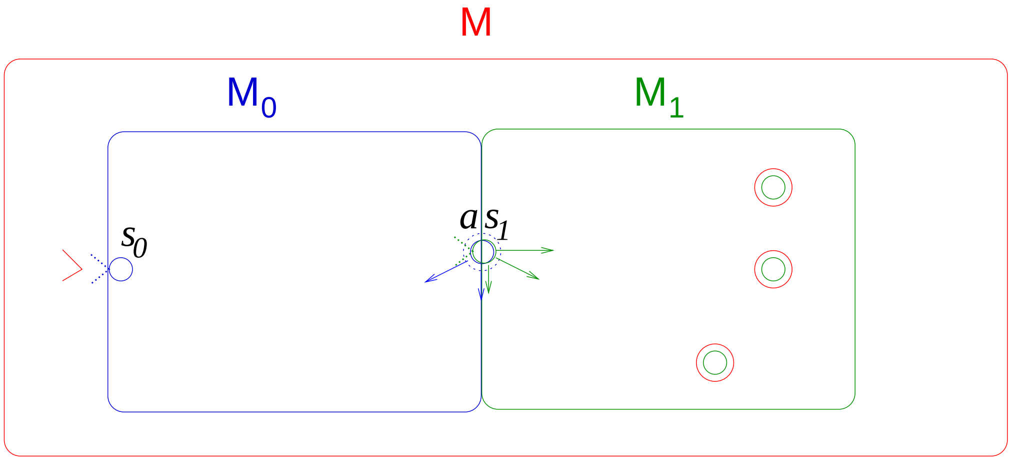
- We verified that combining recognized languages
  by union, intersection, and difference,
  yields recognized languages.

- What about concatenation?
  li Suppose we have two automata M_0 and M_1.
  Construct automaton M such that

$$\mathcal{L}(M) = \mathcal{L}(M_0) \cdot \mathcal{L}(M_1)$$

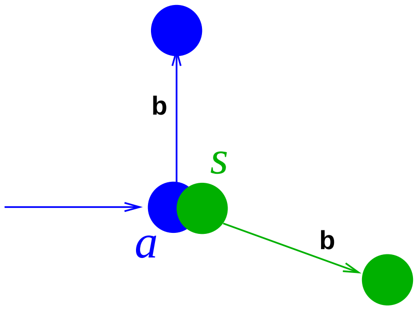


Trying to make this work



- Problem: Can't coalesce a and σ_1 :

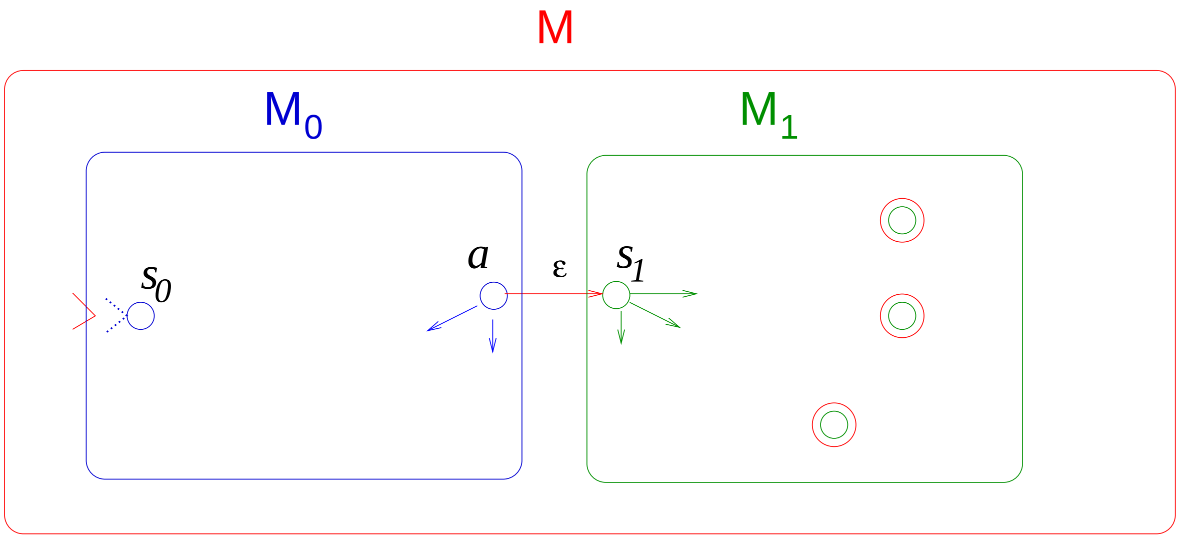
  They might have conflicting transitions rules:



And computation might proceed back and forth between M_0 and M_1 .

Spontaneous transitions

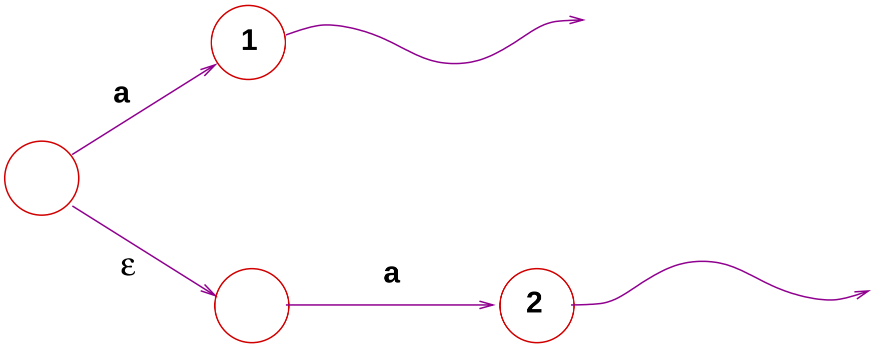
- How about allowing spontaneous transitions between states, $q \xrightarrow{p}$ without any symbol read.
- To streamline notation we can think of such transitions triggered by ε: $q \xrightarrow{\epsilon} p$.



- We call these $\boxed{\textit{epsilon-transitions}}$, in analogy to our previous notation: $q \xrightarrow{w} p$ for a combined transition from state q to p obtained by reading the string w.

Nondeterminism

- ε-transitions yield “ambiguous” computation:
  multiple transitions for a state+symbol may be created:



Admitting non-determinism

- We consider relaxing the requirements that each transition rule is a function (univalent and total) and triggered by reading a letter.
- This relaxation does not correspond to normal hardware behavior, but:

Admitting non-determinism

- We consider relaxing the requirements that each transition rule is a function (univalent and total) and triggered by reading a letter.
- This relaxation does not correspond to normal hardware behavior, but:

  1. The notion is important in other computation models;

Admitting non-determinism

- We consider relaxing the requirements that each transition rule is a function (univalent and total) and triggered by reading a letter.
- This relaxation does not correspond to normal hardware behavior, but:

1. The notion is important in other computation models;
2. It can be simulated by ε-transitions, which do model natural phenomena; and

Admitting non-determinism

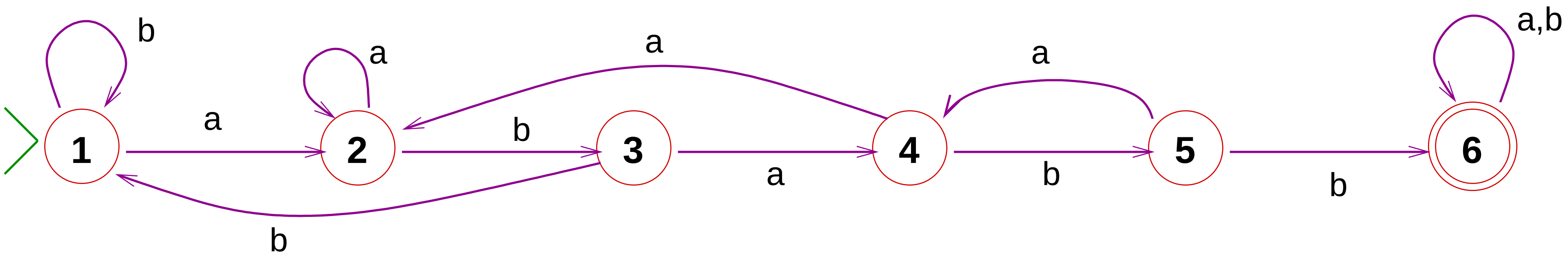
- We consider relaxing the requirements that each transition rule is a function (univalent and total) and triggered by reading a letter.
- This relaxation does not correspond to normal hardware behavior, but:

1. The notion is important in other computation models;
2. It can be simulated by ε-transitions, which do model natural phenomena; and
3. It is algorithmically natural, as we shall now see.

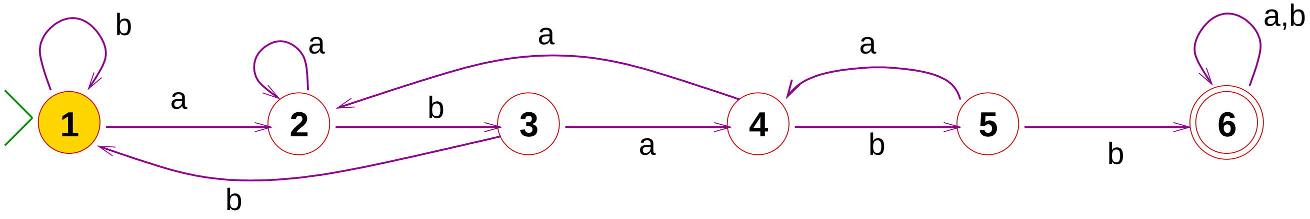
AUTOMATA AS ON-LINE ALGORITHMS

Automata as on-line algorithms

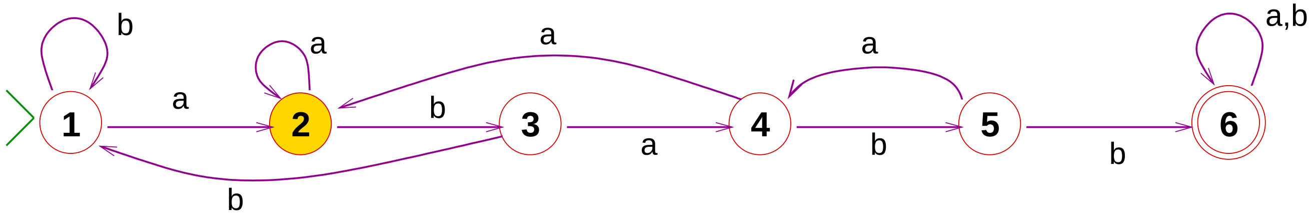
- Automata can be viewed as efficient real time algorithms, which move pointers (or “tokens”) around.
- An automaton to recognize the presence of `ababb`:



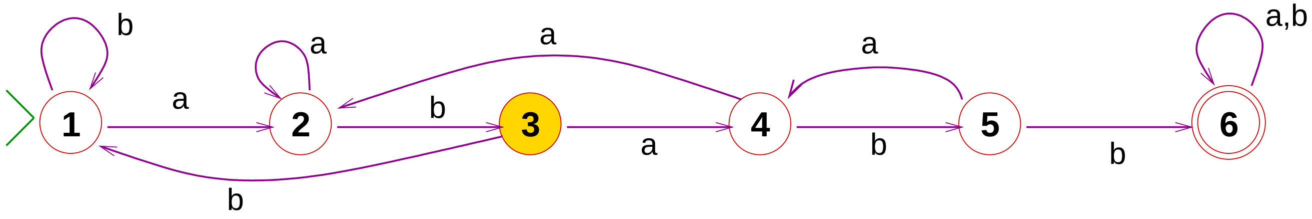
- It is visualized by moving a token for the state position.



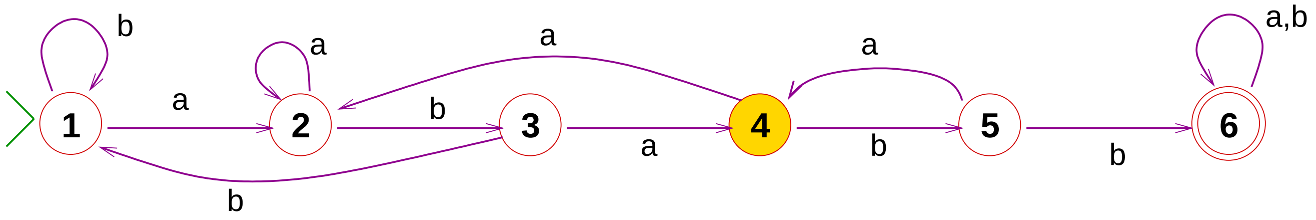
a b a b a b b a



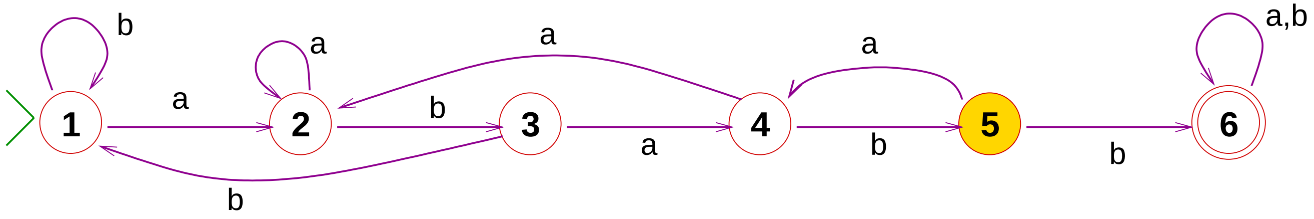
a b a b a b b a



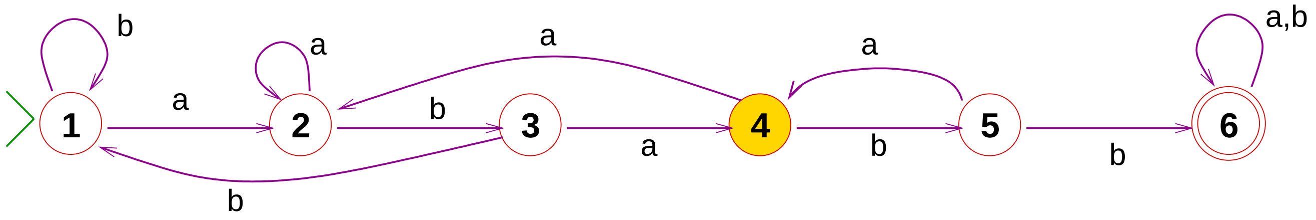
a b $\underline{\textbf{a}}$ b a b b a



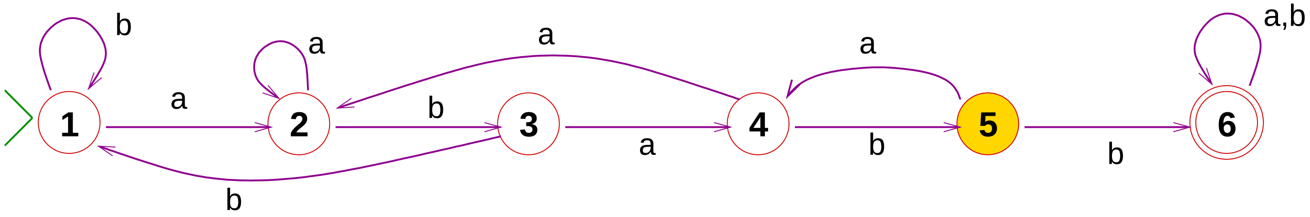
a b a b a b b a



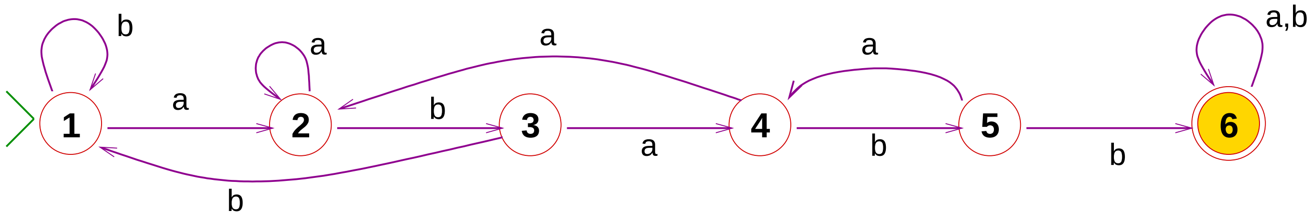
a b a b $\underline{\textbf{a}}$ **b b a**



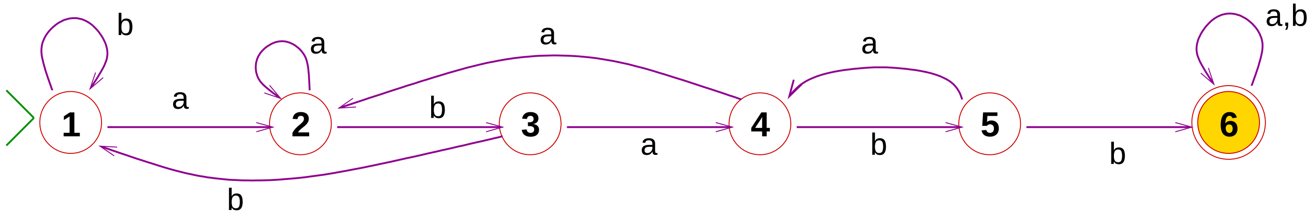
a b a b a $\underline{\mathbf{b}}$ **b a**



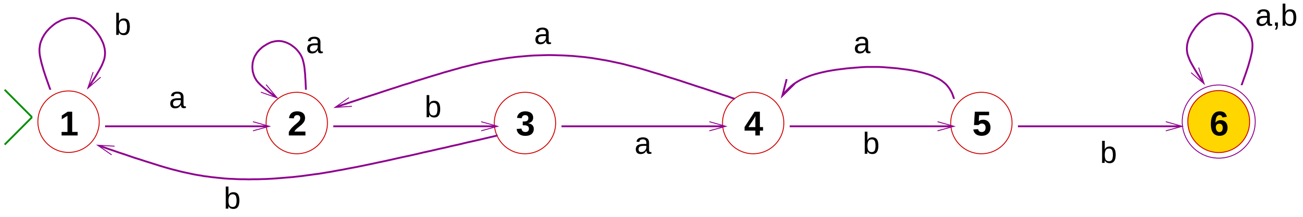
a b a b a b $\underline{\text{b}}$ a



a b a b a b $\underline{\textbf{b}}$ **a**

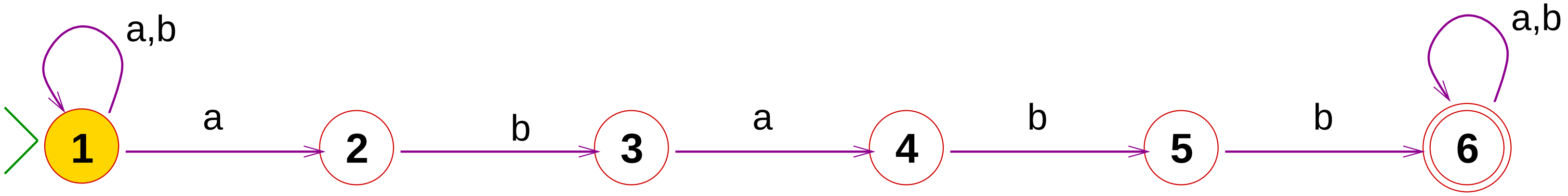


a b a b a b b a



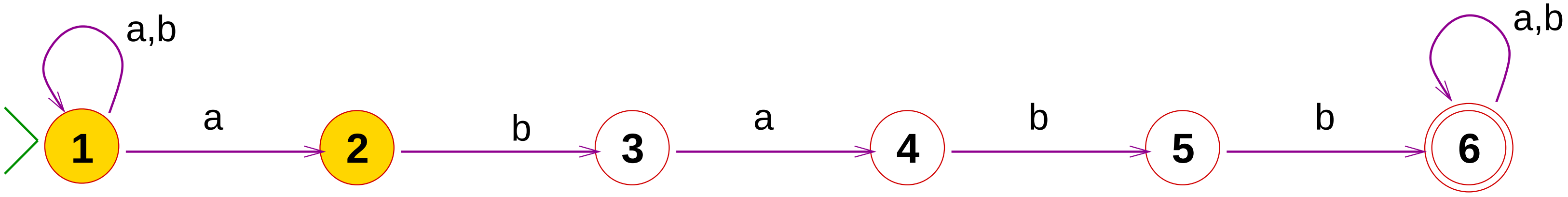
a b a b a b b a _

An alternative, with token rules relaxed.



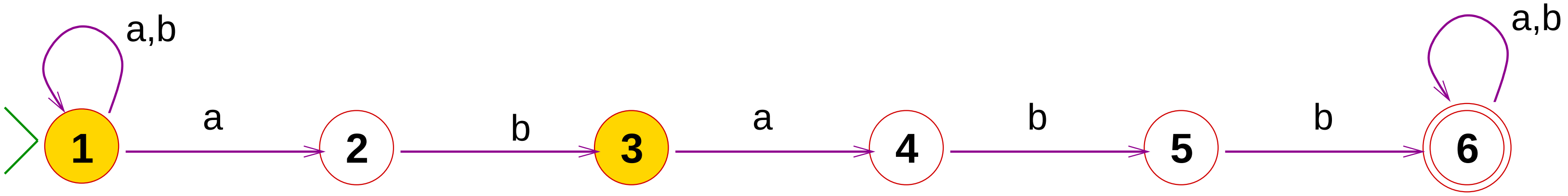
<u>a</u> b a b a b b a

An alternative, with token rules relaxed.



a b a b a b b a

An alternative, with token rules relaxed.



a b <u>a</u> b a b b a

- Next states marked are 1,2 and 4. Etc.

Non-deterministic automata

A non-deterministic automaton over Σ:

- Finite (non-empty) set Q of states
- Start state s and accepting states $A \subseteq Q$
- Transition mapping: $\delta : (Q \times \Sigma_\epsilon) \Rightarrow Q$
- Here $\Sigma_\epsilon = \Sigma \cup \{\varepsilon\}$
- Still using the notation $q \xrightarrow{\sigma} p$ for $\langle q, \sigma, p \rangle \in \delta$
- But $q \xrightarrow{\epsilon} p$ is also an option.

Computation state-traces

- If $w = \sigma_1 \cdot \sigma_2 \cdot \cdots \cdot \sigma_n$ where $\sigma_i \in \Sigma_\varepsilon$,

  and $q \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} r_2 \quad \cdots \quad r_{n-1} \xrightarrow{\sigma_n} p$

  then $q \overset{w}{\Longrightarrow} p$.

Computation state-traces

- If $w = \sigma_1 \cdot \sigma_2 \cdots \cdot \sigma_n$ where $\sigma_i \in \Sigma_\varepsilon$,
  and $q \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} r_2 \quad \cdots \quad r_{n-1} \xrightarrow{\sigma_n} p$
  then $q \overset{w}{\Longrightarrow} p$.
- The sequence of states

$$q \;\; r_1 \;\; r_2 \quad \cdots \quad r_{n-1} \;\; p$$

as above is a $\boxed{\textit{state-trace}}$ of the NFA for input w.

Generative definition of $q \overset{w}{\Longrightarrow} p$

- **Base.** $q \xrightarrow{\epsilon} q$ for all $q \in Q$.
- **Step.** If $q \xrightarrow{\sigma} p$ by the NFA's transition,
  and $p \overset{w}{\Longrightarrow} r$ has been generated already (where $\sigma \in \Sigma_\epsilon$) then $q \overset{\sigma\cdot w}{\Longrightarrow} r$.

Acceptance by an NFA

- M ***accepts*** a string $w \in \Sigma^*$ if $s \overset{w}{\Longrightarrow} A$.

Acceptance by an NFA

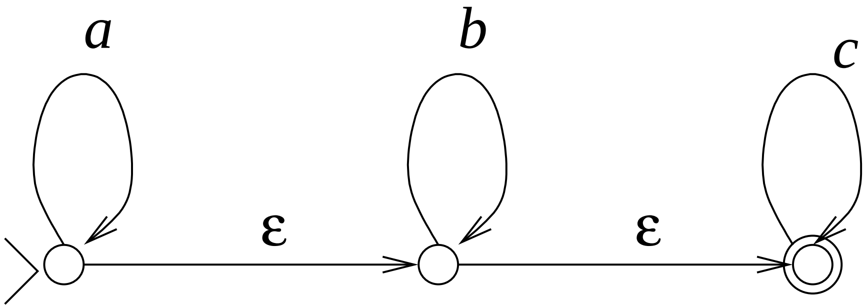
- M $\boxed{\textit{accepts}}$ a string $w \in \Sigma^*$ if $s \overset{w}{\Longrightarrow} A$.
- This dfn is like for DFAs, but now

1. A string w is accepted if there is ***some*** state-trace for $s \overset{w}{\Longrightarrow} A$.
2. A “lucky trace” may include ε-transitions.

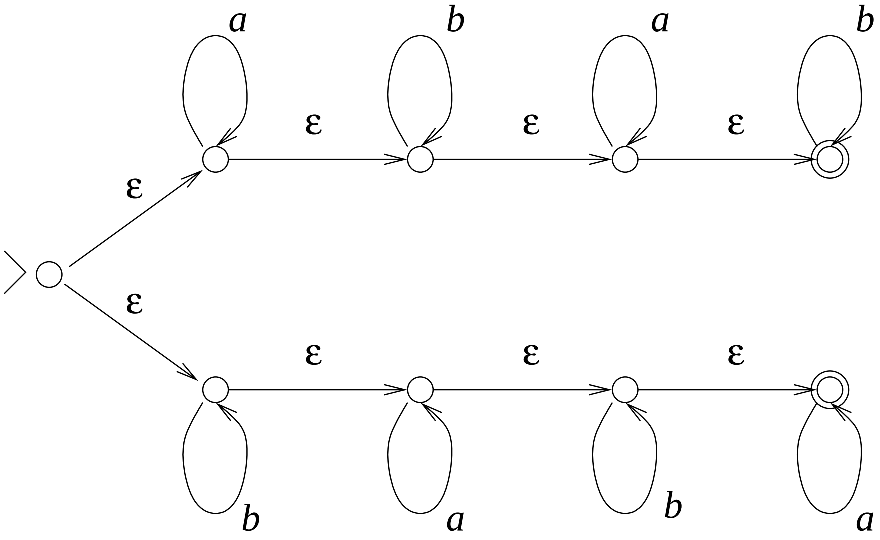
Acceptance by an NFA

- M $\boxed{\textit{accepts}}$ a string $w \in \Sigma^*$ if $s \overset{w}{\Longrightarrow} A$.
- This dfn is like for DFAs, but now
  1. A string w is accepted if there is ***some*** state-trace for $s \overset{w}{\Longrightarrow} A$.
  2. A “lucky trace” may include ε-transitions.
- The $\boxed{\textit{language recognized}}$ by M
  is the set of accepted strings.

Example: $\mathcal{L}(\mathtt{a^*b^*c^*})$

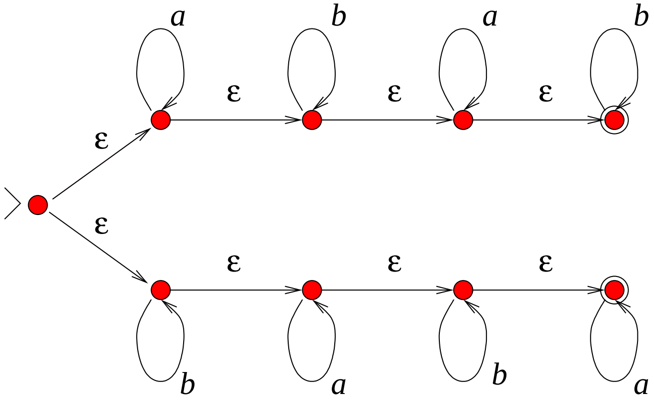


Recognizing $\mathcal{L}(a^*b^*a^*b^* \cup b^*a^*b^*a^*)$



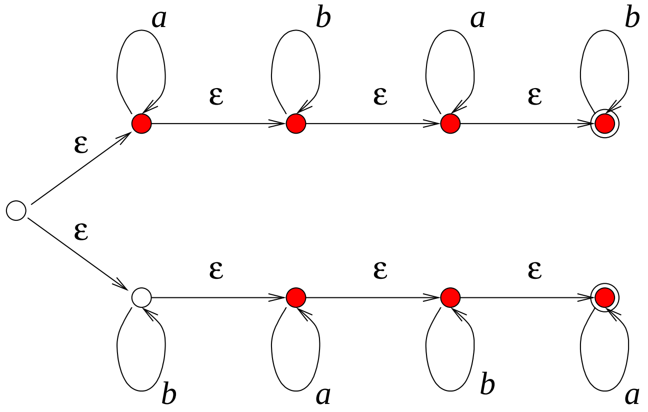
$a^*b^*a^*b^* \cup b^*a^*b^*a^*$

Recognizing $\mathcal{L}(a^*b^*a^*b^* \cup b^*a^*b^*a^*)$



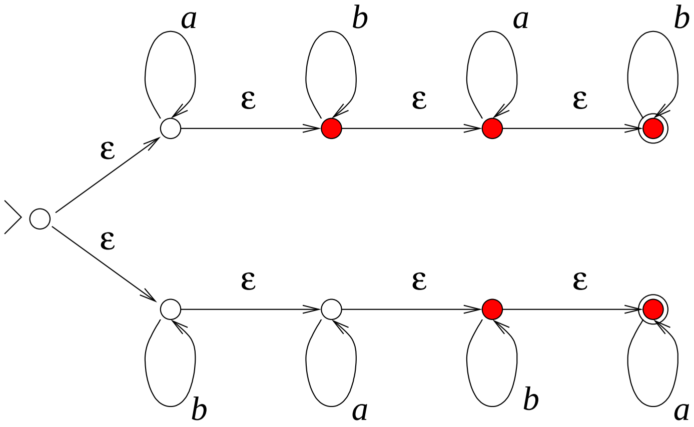
>abb

Recognizing $\mathcal{L}(\mathtt{a^*b^*a^*b^*} \cup \mathtt{b^*a^*b^*a^*})$



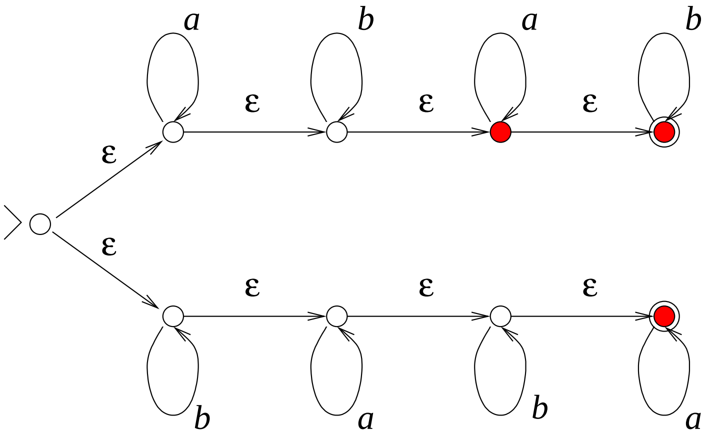
a>bb

Recognizing $\mathcal{L}(\mathtt{a^*b^*a^*b^*} \cup \mathtt{b^*a^*b^*a^*})$



ab>b

Recognizing $\mathcal{L}(\mathtt{a^*b^*a^*b^*} \cup \mathtt{b^*a^*b^*a^*})$



abb>

So the number of states is *reduced* with each step.

DFA-RECOGNIZED = NFA-RECOGNIZED

DFA-RECOGNIZED = NFA-RECOGNIZED

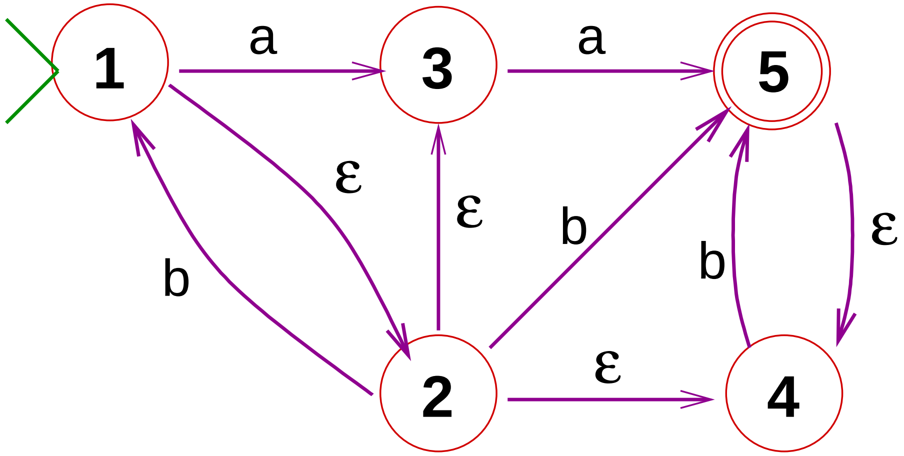
- DFA-RECOGNZD $\Longrightarrow$ NFA-RECOGNZD:

  TRIVIAL: Every DFA is an NFA

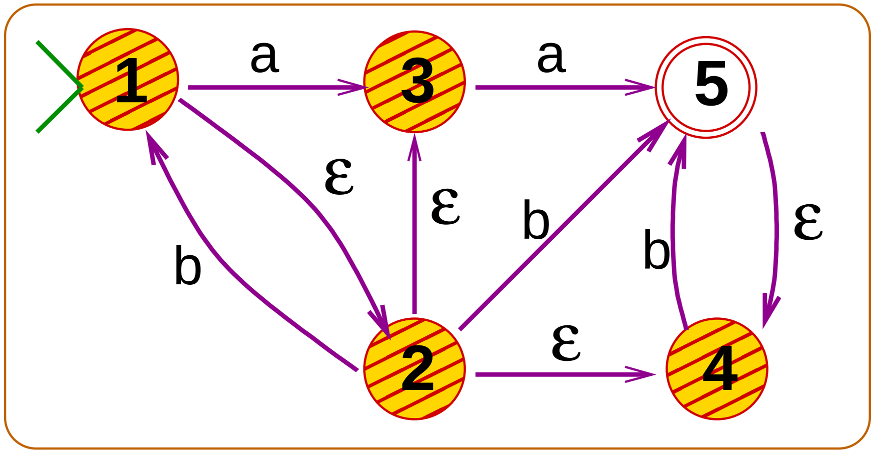
- NFA-RECOGNZD $\Longrightarrow$ DFA-RECOGNZD...

Converting NFAs to equivalent DFAs

- Given an NFA N :

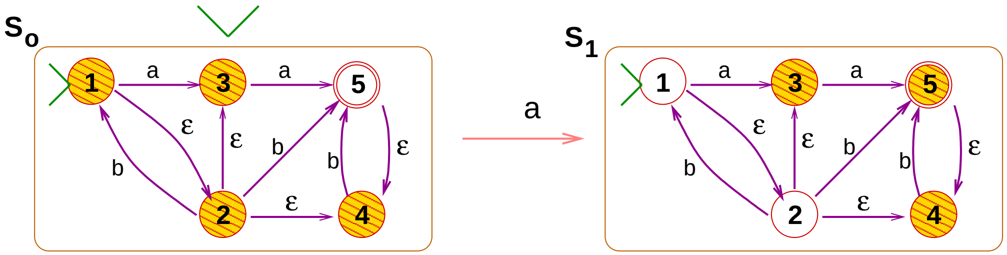


- Mark as “on” the states reachable before reading any input:

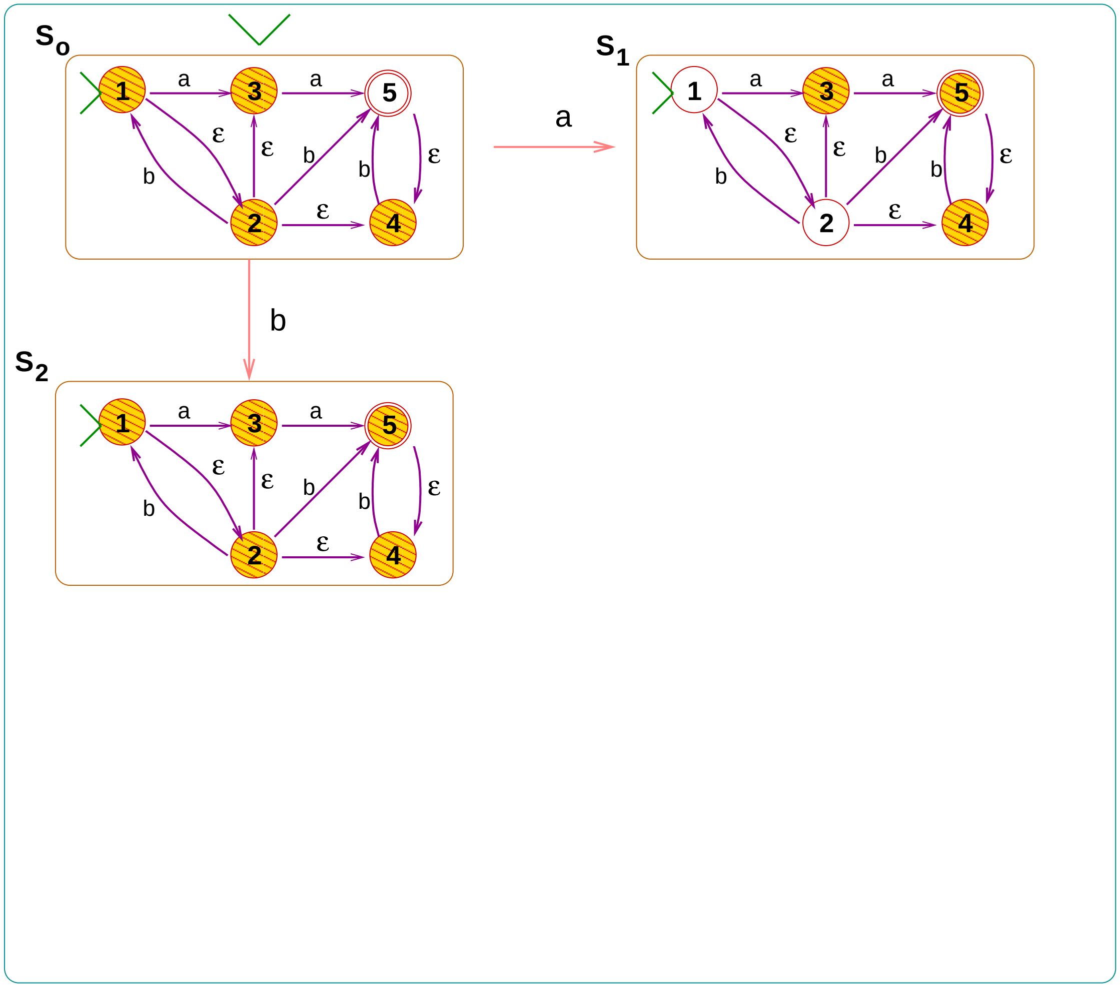


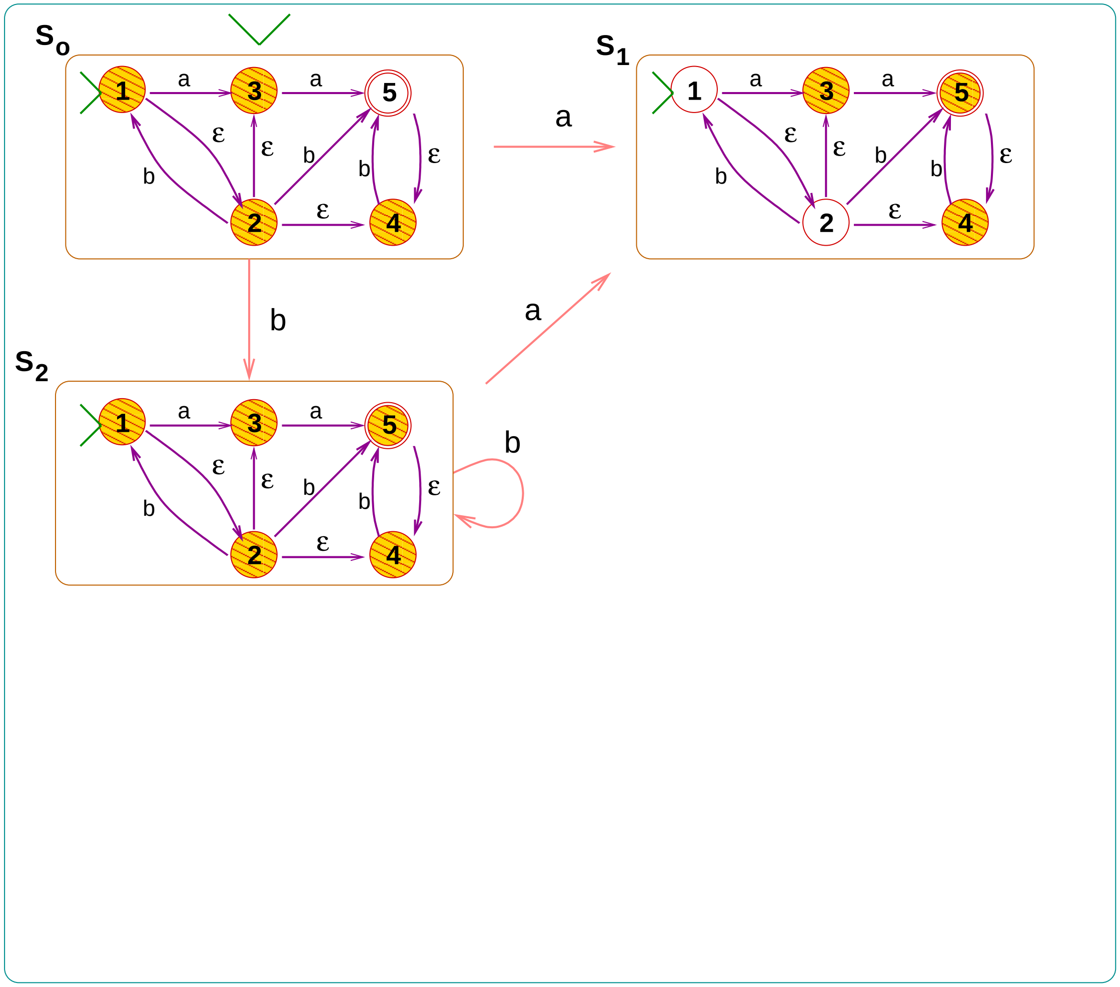
- This setup is the “start state” of our deterministic automaton.

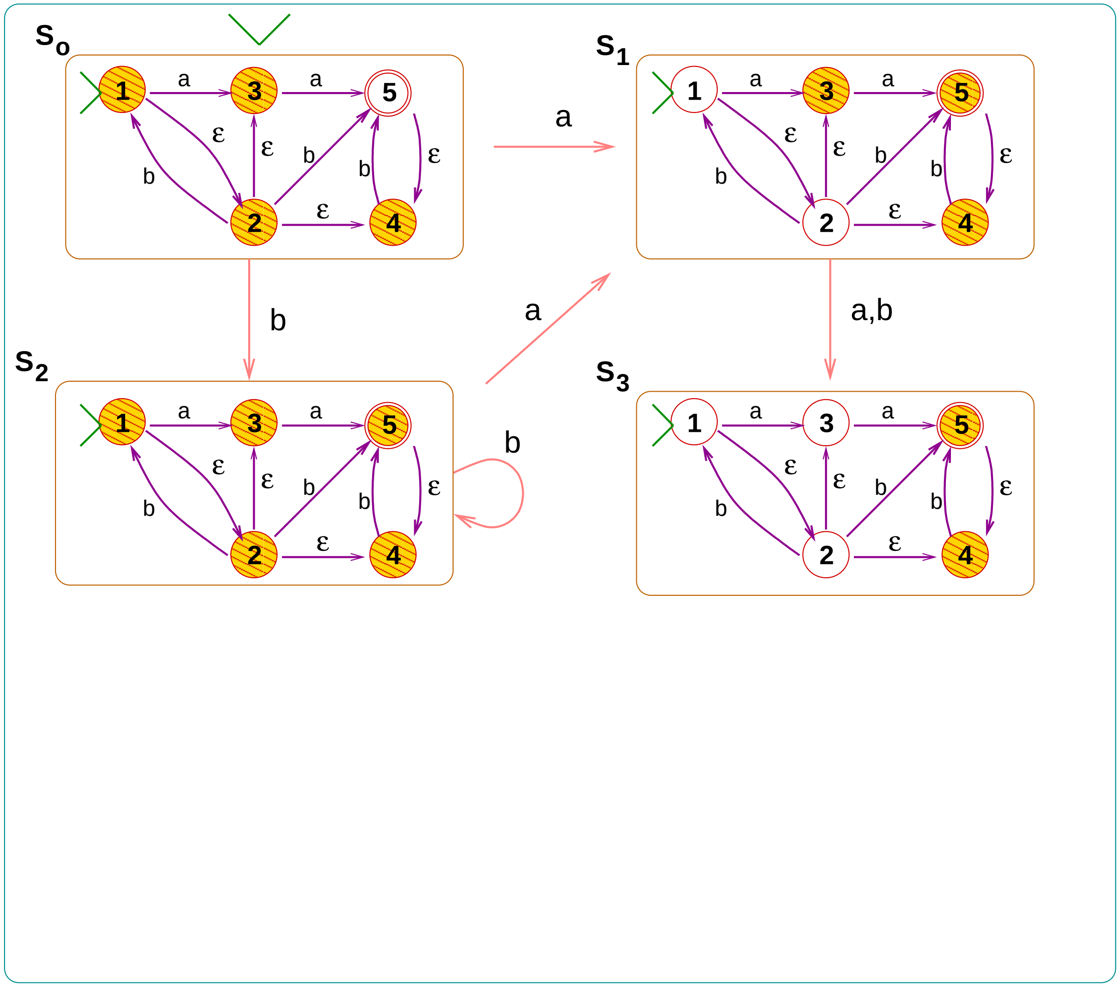
- On rreading a the NFA can be in one of possible states:

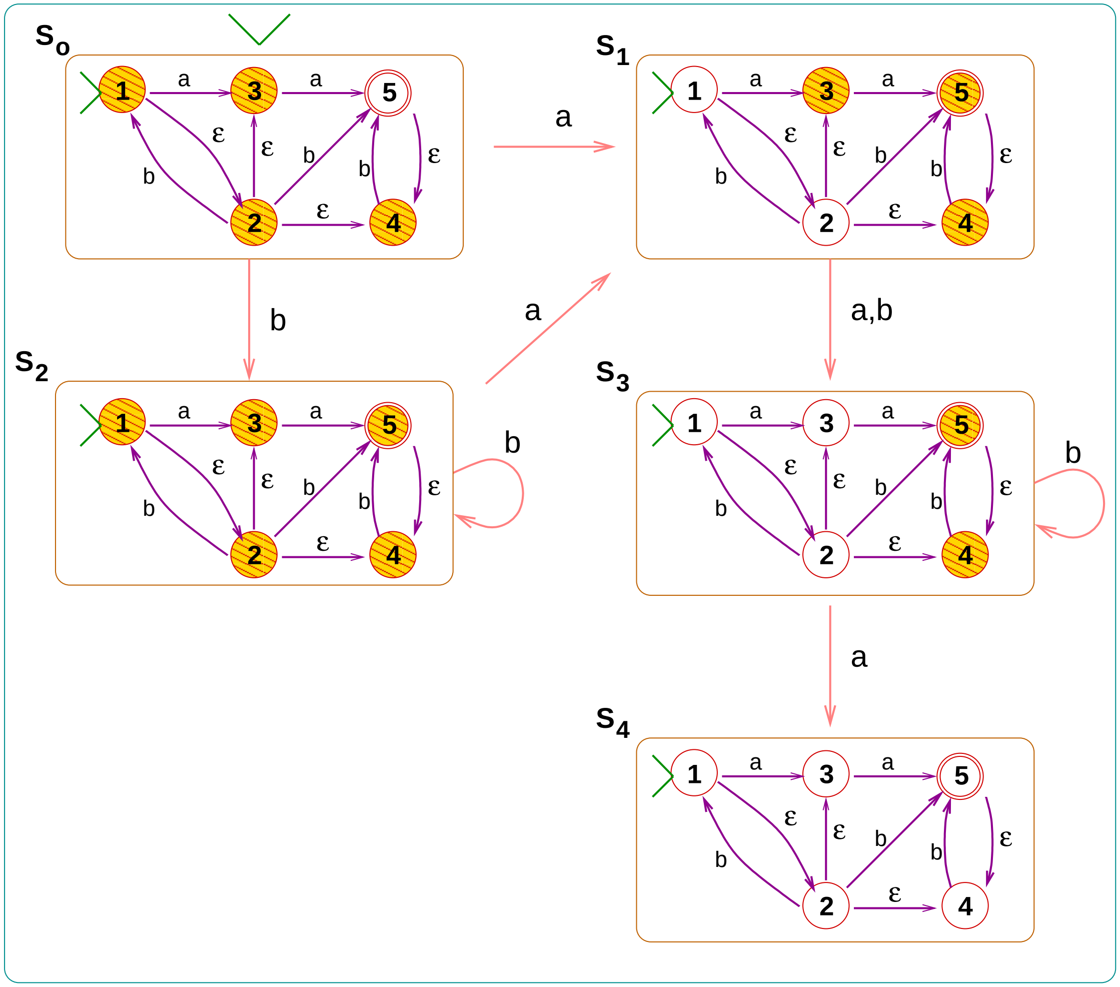


- Proceed to explore the set of reachable states of N :



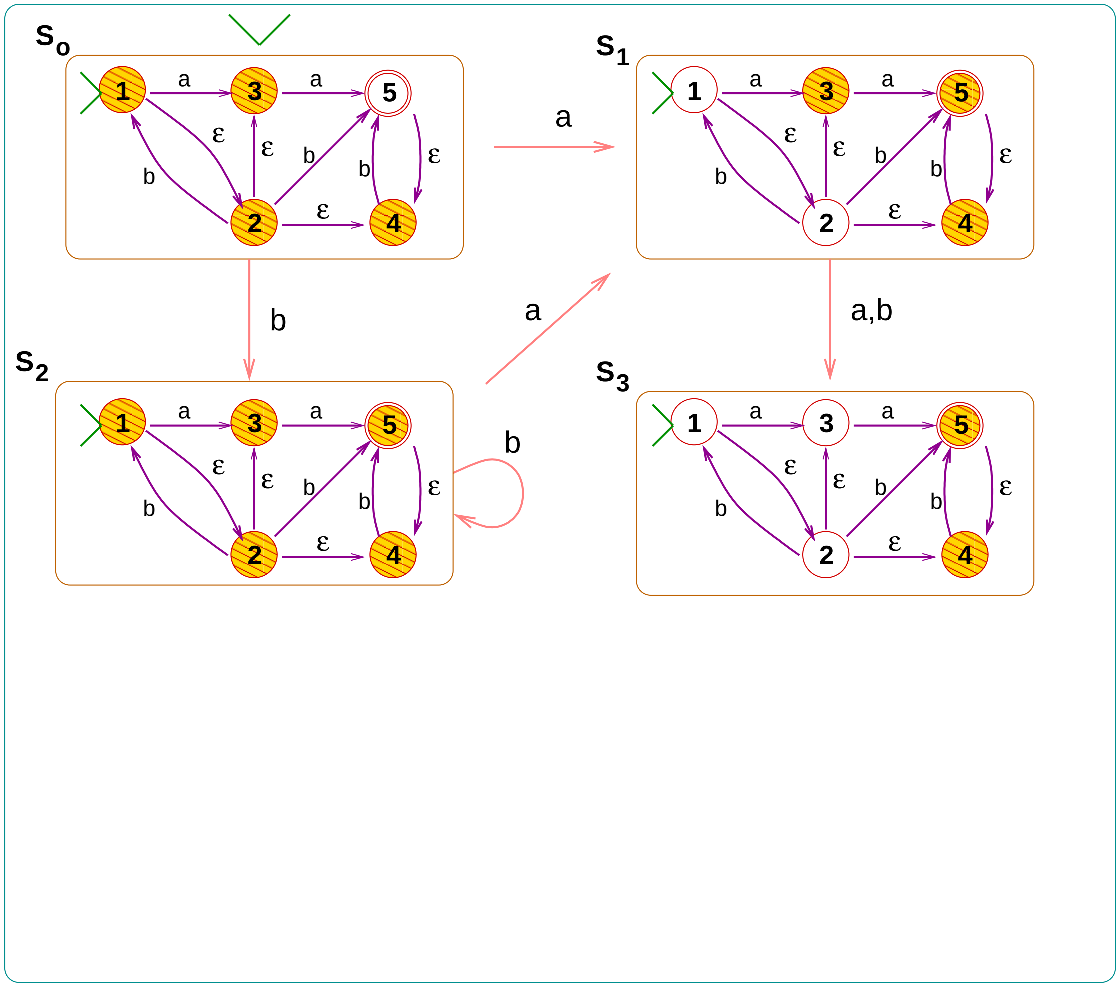

S o
1
3
5
2
4
a
a
ε
ε
b
b
b
ε
ε
a
S 1
1
3
5
2
4
a
a
ε
ε
b
b
b
ε
ε
b
a
S 2
1
3
5
2
4
a
a
ε
ε
b
b
b
ε
ε
b


S0
S1
S2
S3
a
b
a
a,b
b
1
2
3
4
5
ε


S0
1
3
5
2
4
a
a
ε
ε
b
b
b
ε
ε
a
S1
b
a
a,b
S2
b
S3
b
a
S4

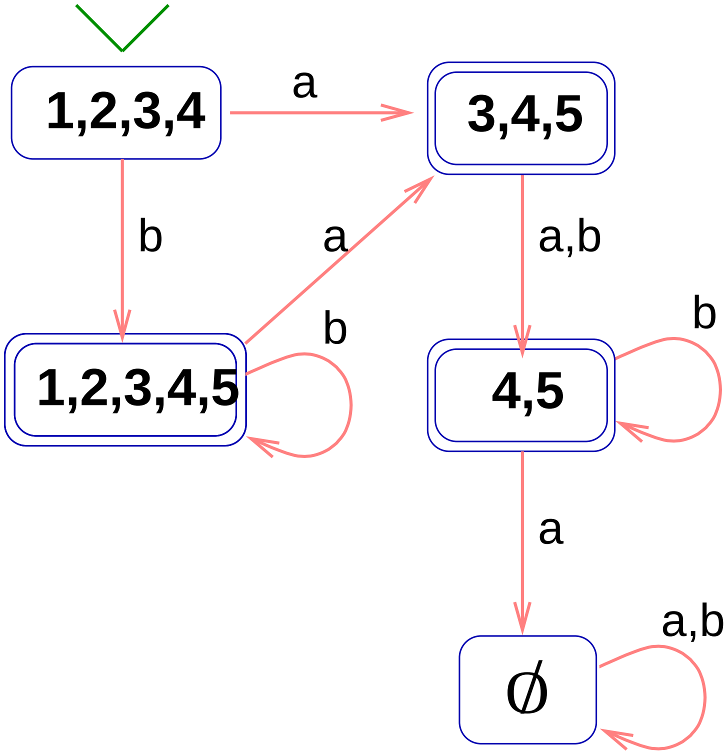
- Complete the transition for the final setup.

- The setups are the states of the new, deterministic, automaton.
- A setup is ***accepting*** if it contains an accepting state of N:


S0
1
a
3
a
5
ε
ε
b
b
b
ε
2
ε
4
a
S1
1
a
3
a
5
ε
ε
b
b
b
ε
2
ε
4
b
a
a,b
S2
1
a
3
a
5
ε
ε
b
b
b
ε
2
ε
4
b
S3
1
a
3
a
5
ε
ε
b
b
b
ε
2
ε
4

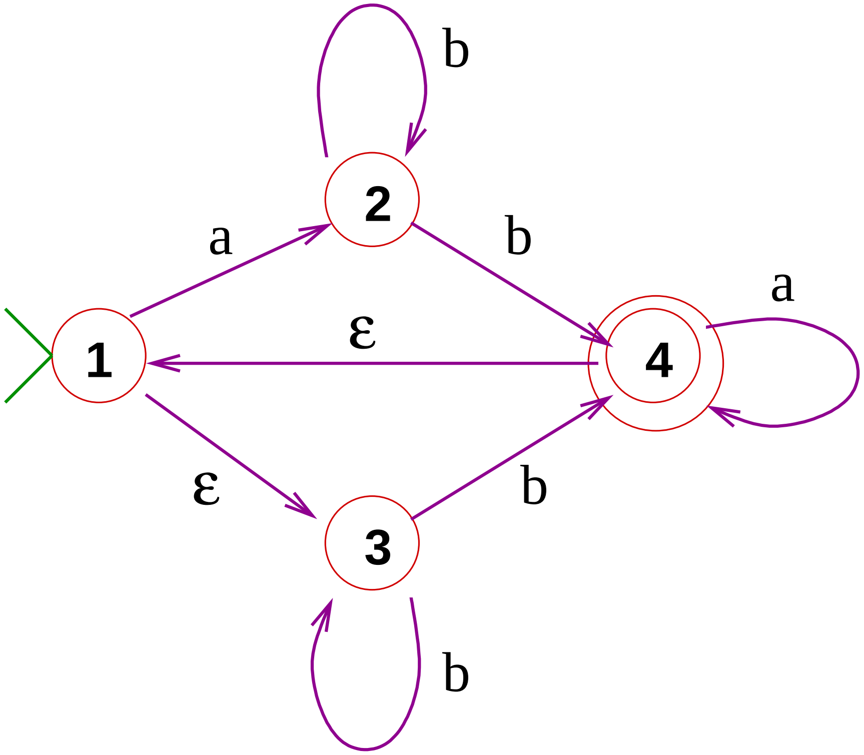
The resulting DFA

- Each state of the DFA obtained is a setup of N 's states:

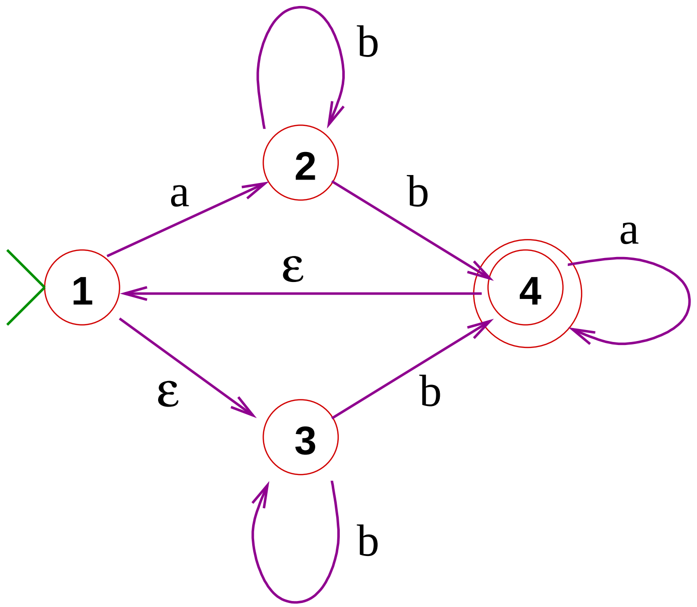


- We have constructed from an NFA N an equivalent DFA M.

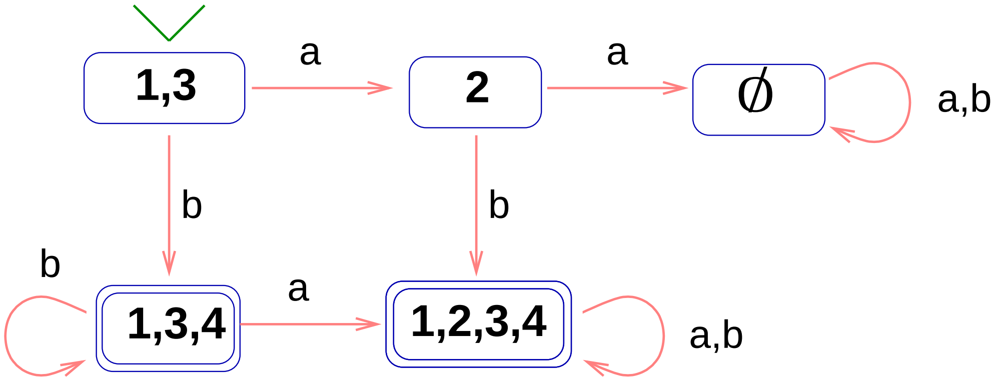
Another example



Another example

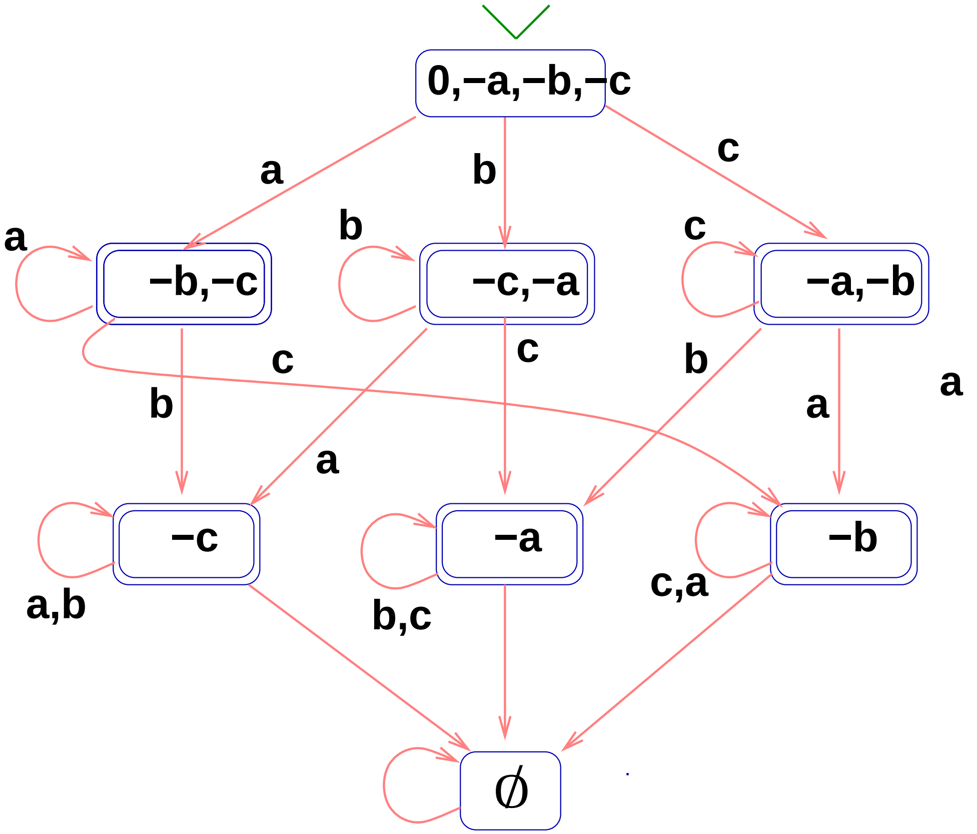


Another example

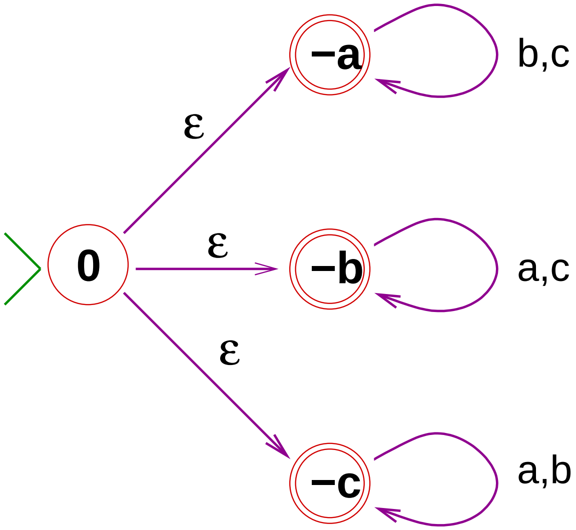


An exponential explosion

- If N has n states, then the DfA obtained
  may have up to 2^n states.
- Is that really necessary?
  Could we have a more efficient construction?
- No! Consider the language of strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that miss at least one letter.
- The smallest DFA recognizing it is

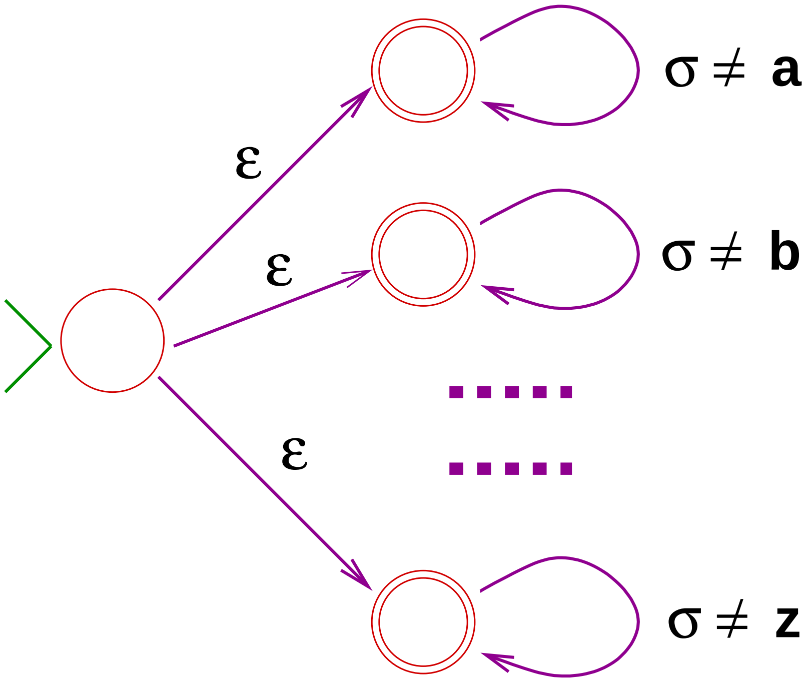

0,−a,−b,−c
a
b
c
a
−b,−c
b
−c,−a
c
−a,−b
c
b
c
b
a
a
a
−c
−a
−b
a,b
b,c
c,a
∅

- But here is a 4-state NFA recognizing it:



- For “missed-som” language over the Latin alphabet
  the smalles recognizing automaton has $2^{26} > 67$ million states!

- But here is a 27 state NFA recognizing it:


ε
ε
ε
σ ≠ **a**
σ ≠ **b**
σ ≠ **z**

Next ...

	Descriptive		Operational
Narrow	STRICT-REG		DFA
			=
Broad	REGULAR	$\Longrightarrow$	NFA

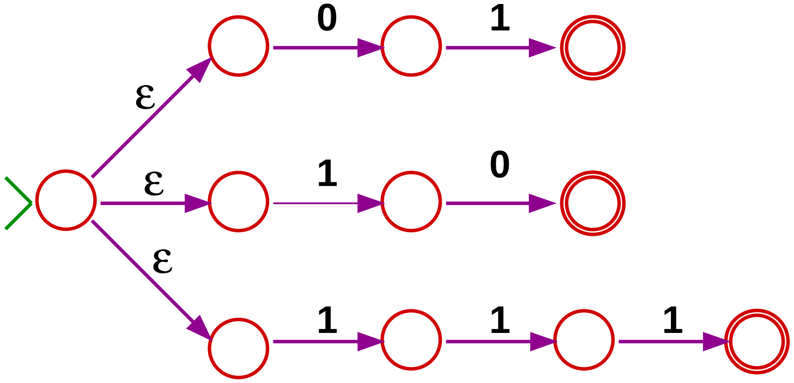
Reminder: Generating the regular languages

1. Every finite language is regular.
2. If L, K are regular, then so are their union, intersection, complement, concatenation, star, and plus.

- We show that all regular languages are recognized by NFAs (and therefore by DFAs).
- The proof is by induction on the generative dfn of the regular languages.

Finite languages are recognized

- For example $\{01, 10, 111\}$ is recognized by



- We know that it suffices to take the finite languages with 0 or 1 elements, each a string of size 0 or 1.

  By this construction, what would be the NFA recognizing

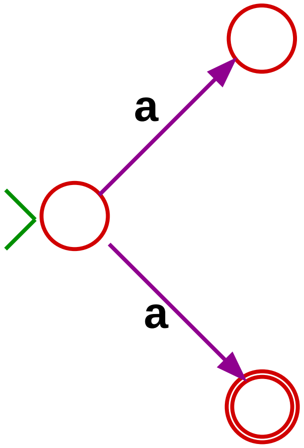
  $\{0\}$? $\{\varepsilon\}$? $\emptyset$?

Complement of recognized is recognized

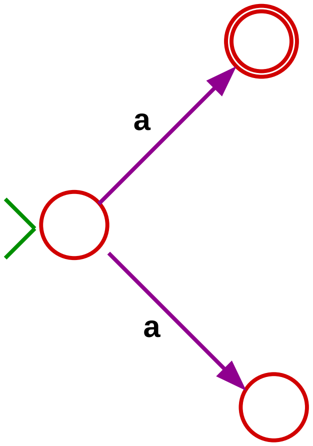
- We have seen:
  A language recognized by an NFA is recognized by a DFA M,
  so its complement is recognized by the DFA $\bar{M}$
  obtained by replacing in M acceptance and non-acceptance.

- Note: This idea doesn't work for NFAs:

N:



$\bar{\mathbf{N}}$**:**



NFA N accepts a and so does $\bar{N}$.

The $\cup$ and $\cap$ of recognized is recognized

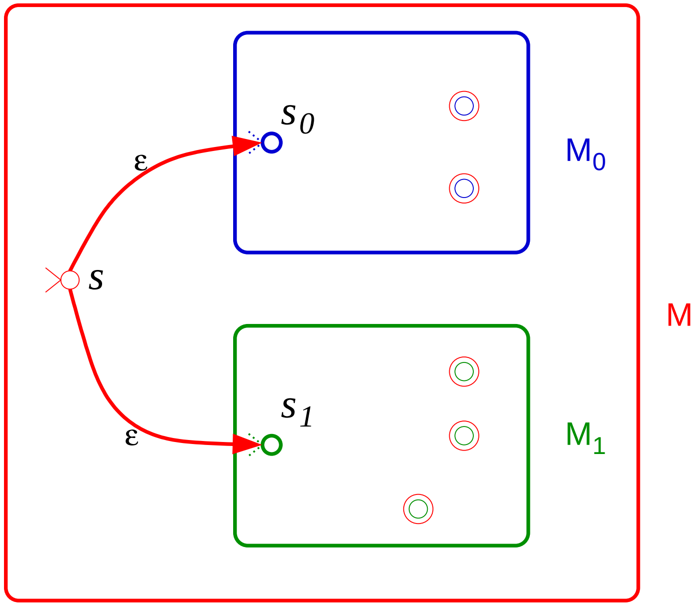
- We already showed this for DFAs.

The ∪ and ∩ of recognized is recognized

- We already showed this for DFAs.
- An alternative approach for union:

  Given $L_0 = \mathcal{L}(M_0)$ and $L_1 = \mathcal{L}(M_1)$,

  here's an NFA M that recognizes $L_0 \cup L_1$



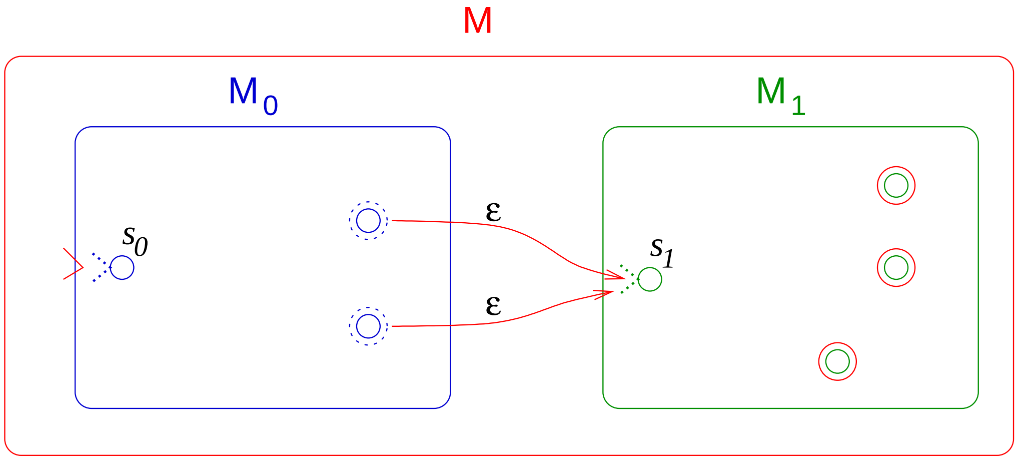
- Once we have closure under union and complement, we obtain closure under intersection:
- 3-If L and K are both recognized, then so are $\bar{L}$ and $\bar{K}$, and therefore $\bar{L} \cup \bar{K}$, as well as its complement which is $= L \cap K$.

- Once we have closure under union and complement,
  we obtain closure under intersection:
- We have $\overline{L \cap K} = \bar{L} \cup \bar{K}$,
  so by complementing both sides we get $L \cap K = \overline{\bar{L} \cup \bar{K}}$
- 3-If L and K are both recognized,
  then so are $\bar{L}$ and $\bar{K}$,
  and therefore $\bar{L} \cup \bar{K}$, as well as its complement which is $= L \cap K$.

- Once we have closure under union and complement, we obtain closure under intersection:
- We have $\overline{L \cap K} = \bar{L} \cup \bar{K}$, so by complementing both sides we get $L \cap K = \overline{\bar{L} \cup \bar{K}}$
- 3-If L and K are both recognized, then so are $\bar{L}$ and $\bar{K}$, and therefore $\bar{L} \cup \bar{K}$, as well as its complement which is $= L \cap K$.

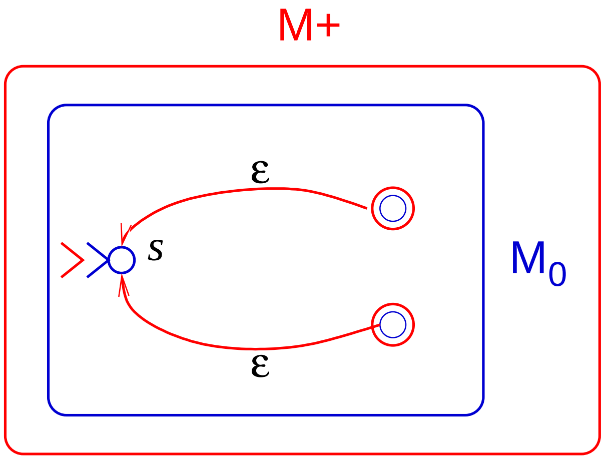
Concatenation of recognized is recognized

- Given $L_0 = \mathcal{L}(M_0)$ and $L_1 = \mathcal{L}(M_1)$,
  here’s an NFA M that recognizes their concatenagion:



Plus and star of recognized are recognized

- Given $L = \mathcal{L}(M)$ here's an NFA M^+ recognizing L^+:



- Since $L^* = L^+ \cup \{\varepsilon\}$ we conclude that L^* is also recognized.

Graphs with reg-exps as labels

- ⋆ Starting with the given NFA,

  Collapse labels: eg, replace $q \xrightarrow{a,b,\epsilon} p$ by $q \xrightarrow{a \cup b \cup \epsilon} p$

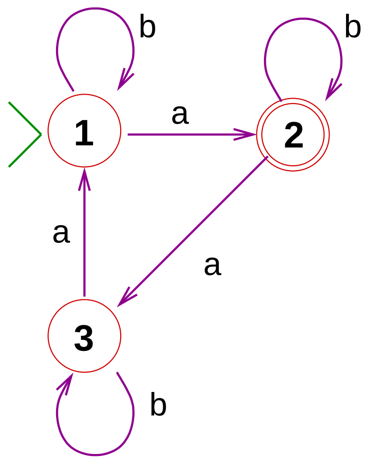
- ⋆ Create a new start state s_0

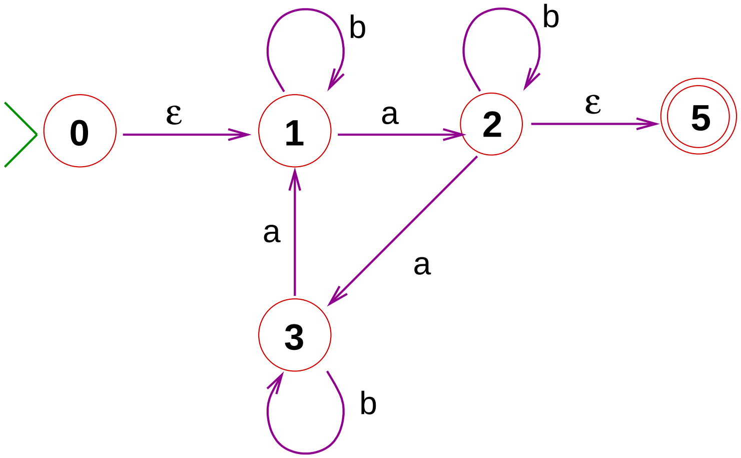
  with an ε-transition to the original start state of N.

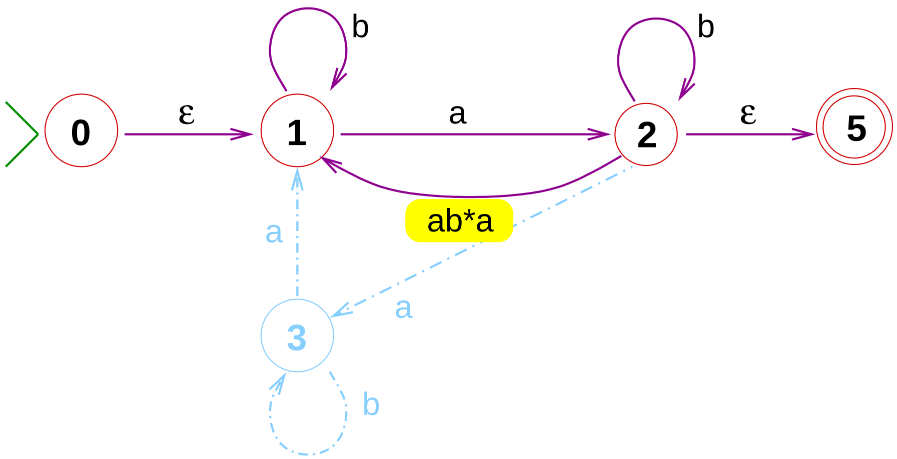
- ⋆ Create a new state a_0 as the only accepting state,

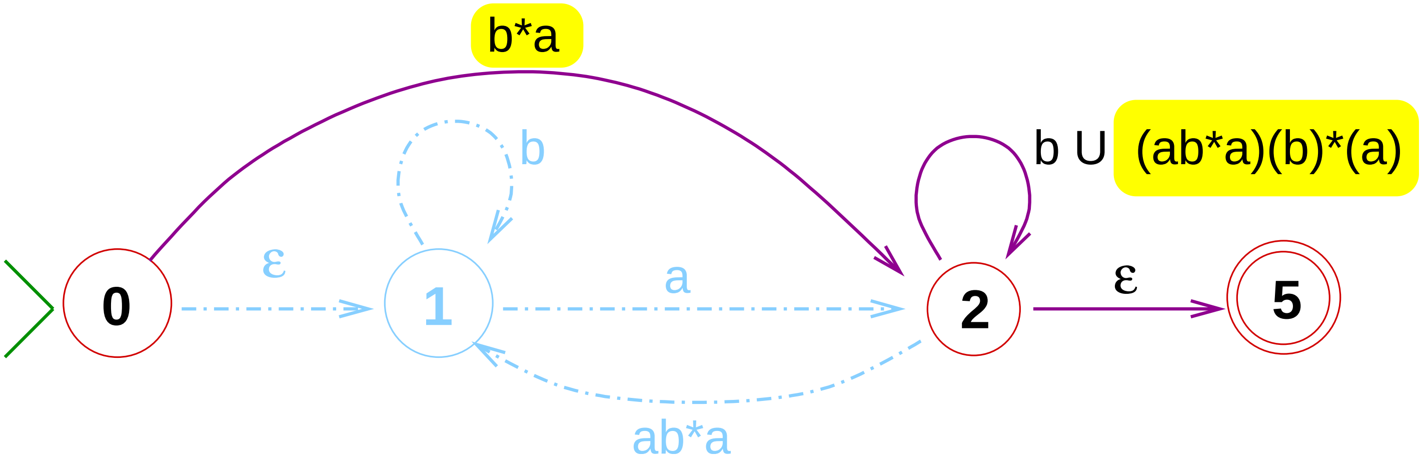
  and create an ε-transition from each accepting state of N to a_0.

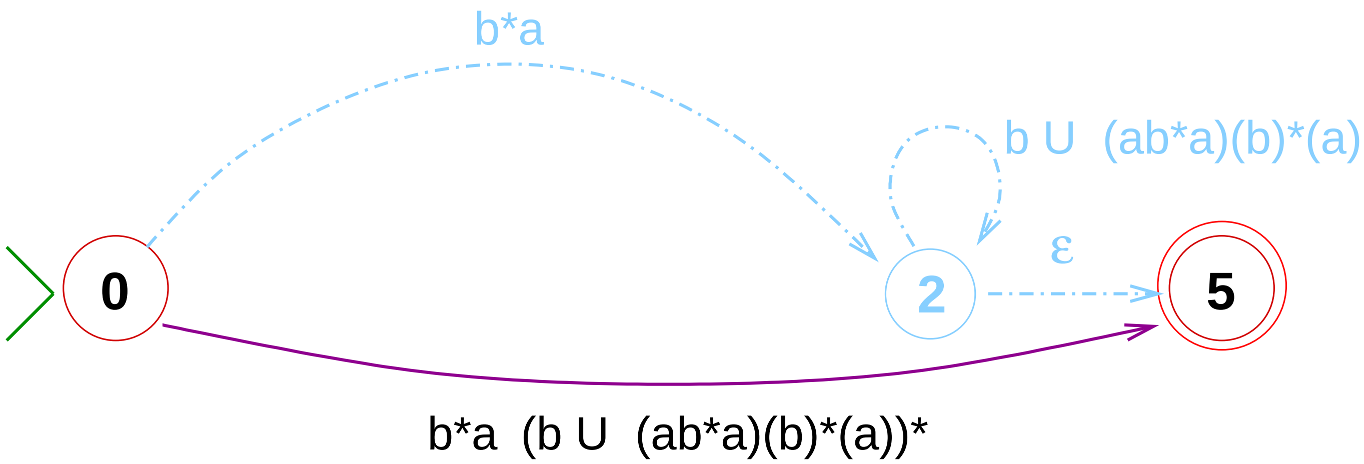
A working example




0
ε
1
b
a
2
b
ε
5
a
a
3
b

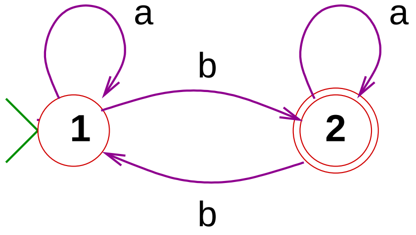

b
b
0
ε
1
a
2
ε
5
a
ab*a
a
3
b

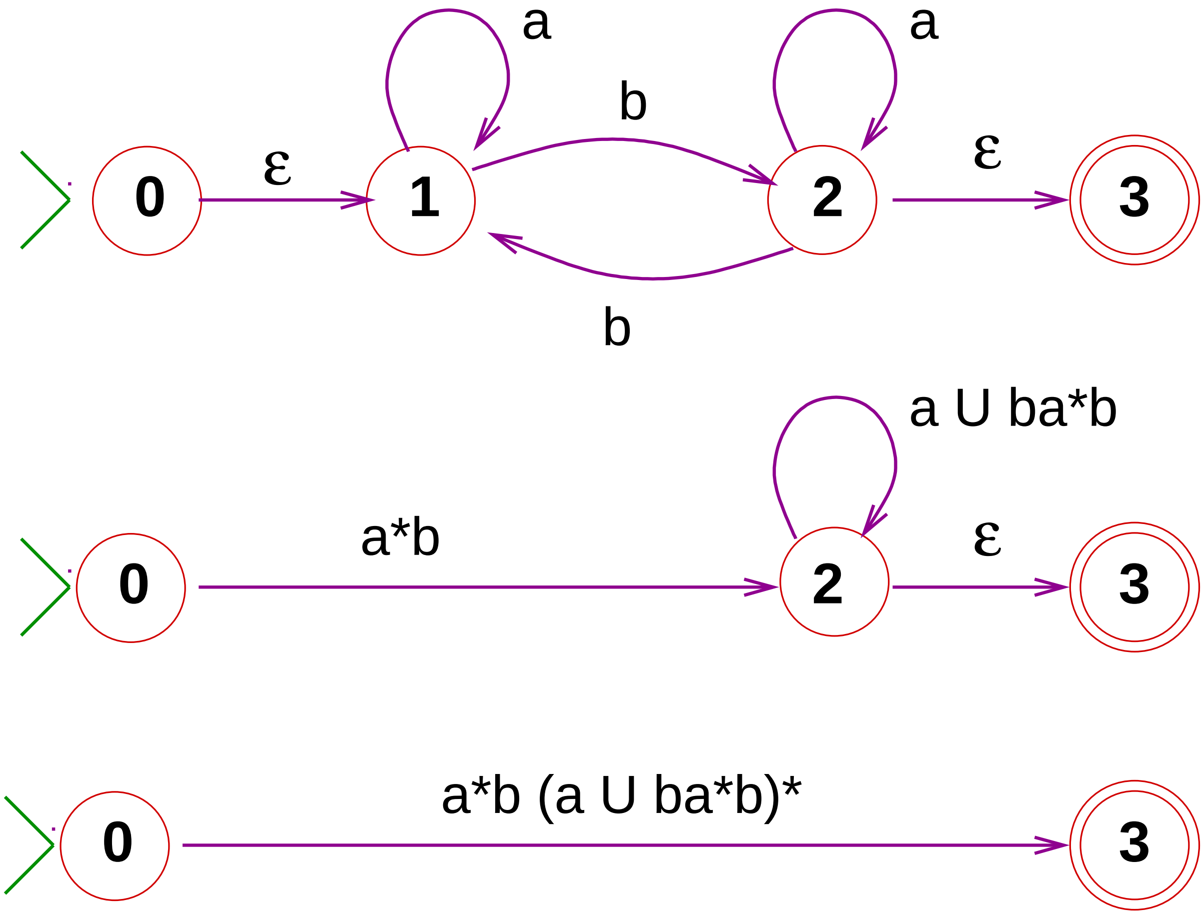

b*a
b
b U (ab*a)(b)*(a)
ε
a
ε
0
1
2
5
ab*a



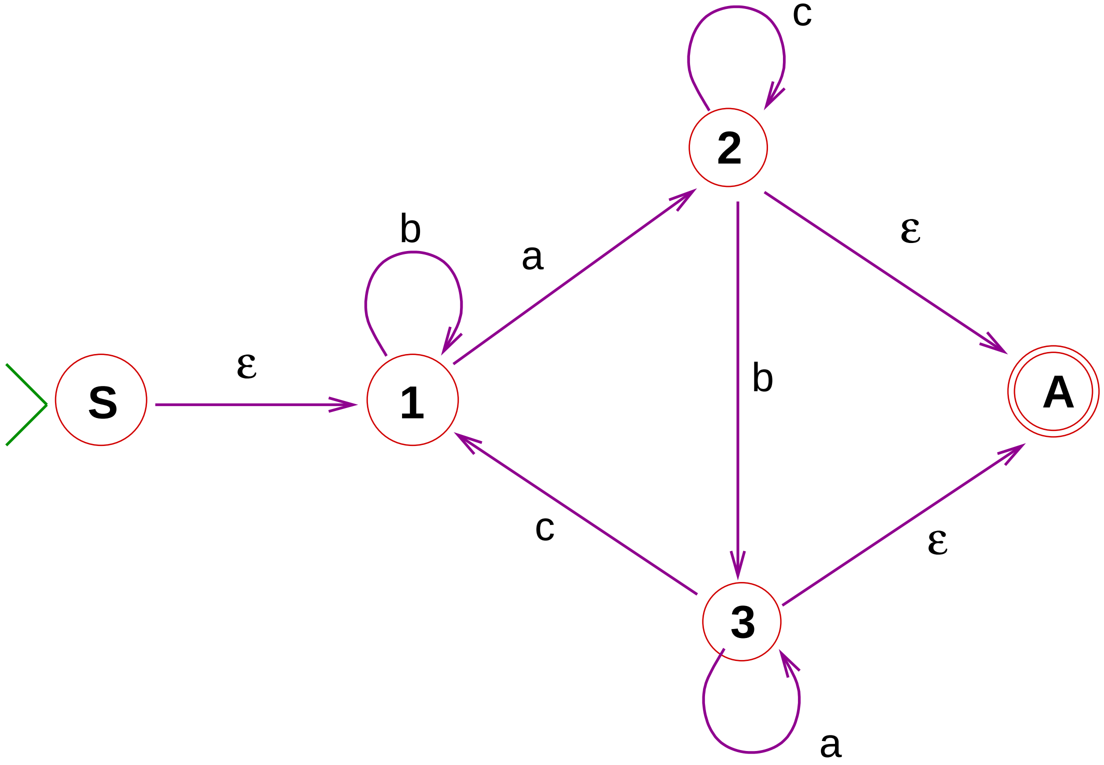
$$\mathcal{L}(N) = \mathcal{L}(\mathtt{b}^{*}\cdot\mathtt{a}\cdot(\mathtt{b}\cup(\mathtt{a}\cdot\mathtt{b}^{*}\cdot\mathtt{a})\cdot(\mathtt{b})^{*}\cdot(\mathtt{a}))^{*})$$

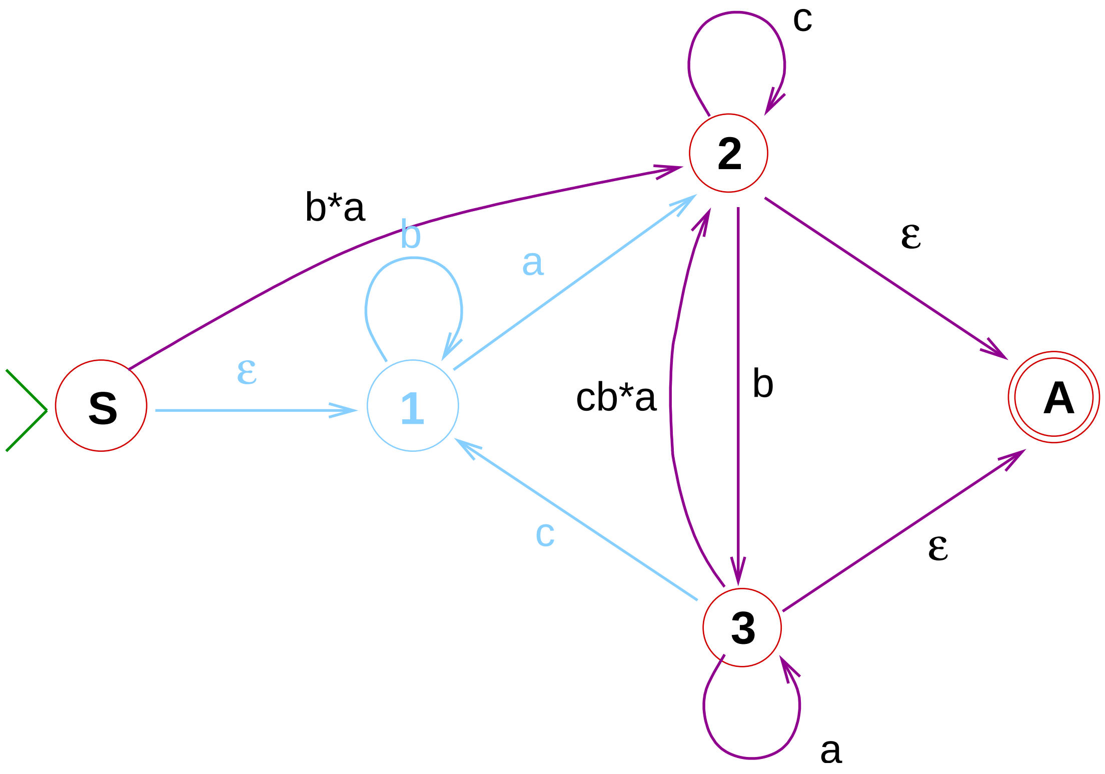
Another example

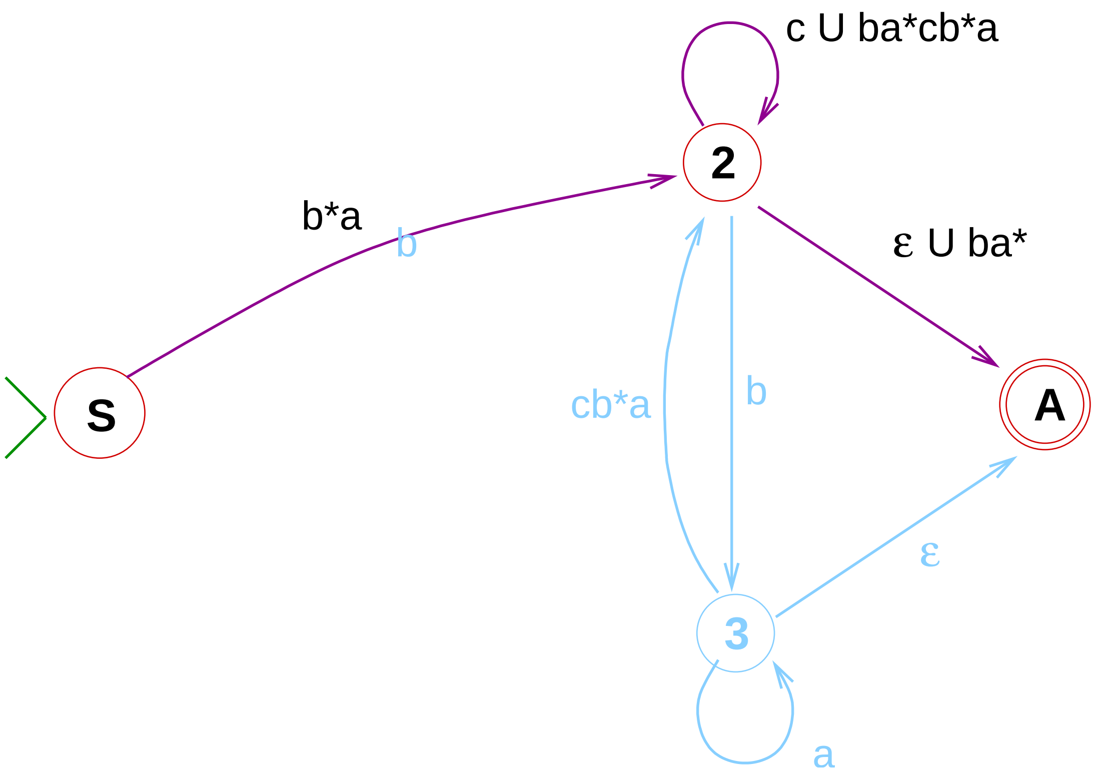


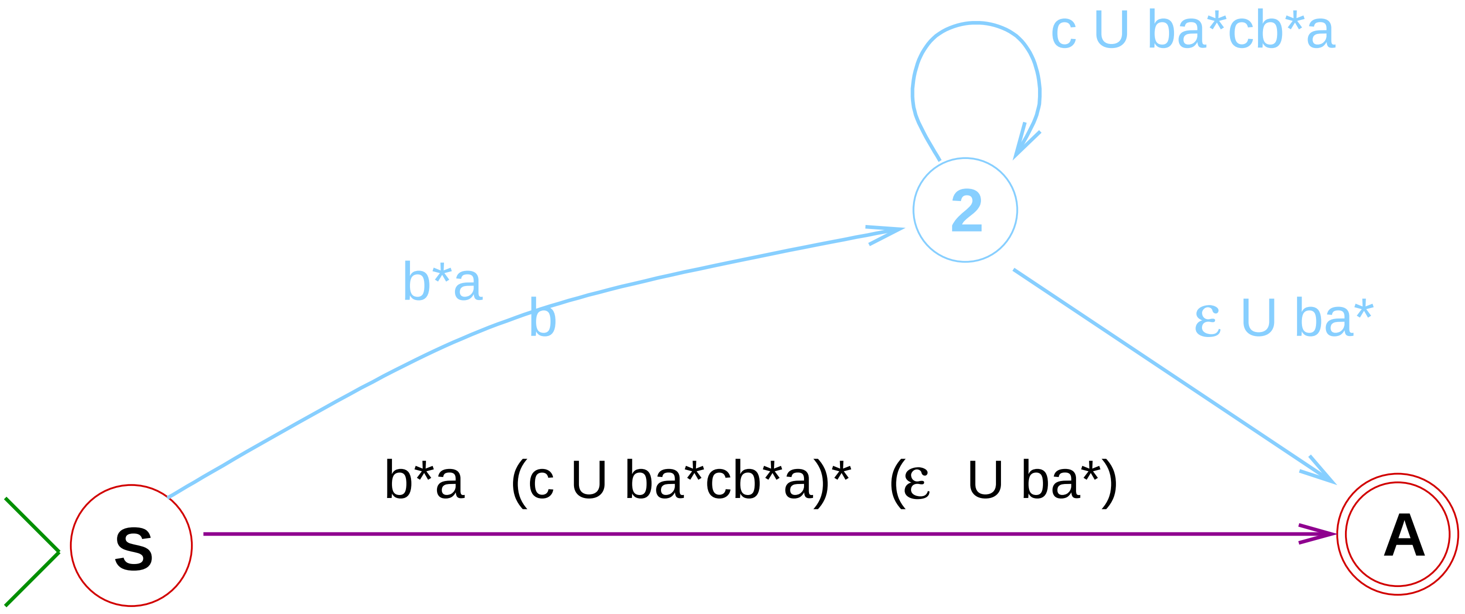

a
a
b
ε
ε
0
1
2
3
b
a U ba*b
a*b
ε
0
2
3
a*b (a U ba*b)*
0
3

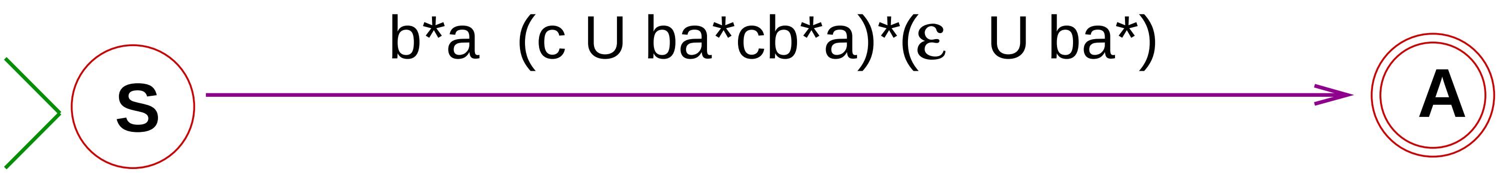
Yet another example




c
2
b*a
b
a
ε
S
ε
1
cb*a
b
A
c
ε
3
a


c U ba*cb*a
2
b*a
b
ε U ba*
S
cb*a
b
A
ε
3
a


c U ba*cb*a
2
b*a
b
ε U ba*
b*a (c U ba*cb*a)* (ε U ba*)
S
A


b*a (c U ba*cb*a)*(ε U ba*)
S
A

Summary

- The collection of DFA-recognized languages is closed under set operations (complement and product constructions)
- A language is NFA-recognized IFF it is DFA-recognized (Powerset construction)
- The collection of recognized languages is closed under all set/language operations.
- Therefore every regular language is recognized.
- Every recognized language is regular (state-elimination construction)

Two-way DFAs

Additional deterministic read-only algorithms

- Consider the language L over $[\mathtt{a}-\mathtt{z}]$ of words that include all letters. No English word is in L, but probably every book.
- L is a regular language: it is the intersection of the 26 languages $\{w \mid w$ has $\sigma\}$ for $\sigma = \mathtt{a}, \mathtt{b}\ldots$.
- The smallest DFA that recognizes L has $> 2^{26} > 67,000,000$ states.
- The smallest NFA recognizing L has 27 states.
- Is there a *deterministic* *algorithm* that does it with a manageable number of states?

A deterministic algorithm for the all-letters problem

- Algorithm: Scan for each digit separately, and repeat.

- This cannot be done if we only read forward!
  The cursor would have to be scrolled back (or repositioned).

- SO let’s imagine a device that behaves just like an automaton,
  but can move the cursor both ways.

Some challenges

- Symbol read determines not only next state,
  but also next move: forward or backward.
- To detect the ends of the input string it must have end-markers,
  say $>$ (the ***gate***) on the left,
  and $\sqcup$ (the ***blank***) on the right.
- Termination is not by reading through,
  but needs to be declared by a final accept state.
  (We need not guarantee termination.)

Two-way automata

A ***two-way automaton (2DFA)*** over an alphabet Σ:

- Finite set of states Q
- $s \in Q$, the *initial state*
- $a \in S$, the *accepting state*
- Transition partial-function: $\delta : Q \times \Gamma \rightharpoonup Q \times \mathbf{Act}$
  where $\Gamma = \Sigma \cup \{>, \sqcup\}$ and $\mathbf{Act} = \{+, -\}$.
- Write $q \xrightarrow{\sigma(\alpha)} p$ for $\delta(q, \sigma) = \langle p, \alpha \rangle$

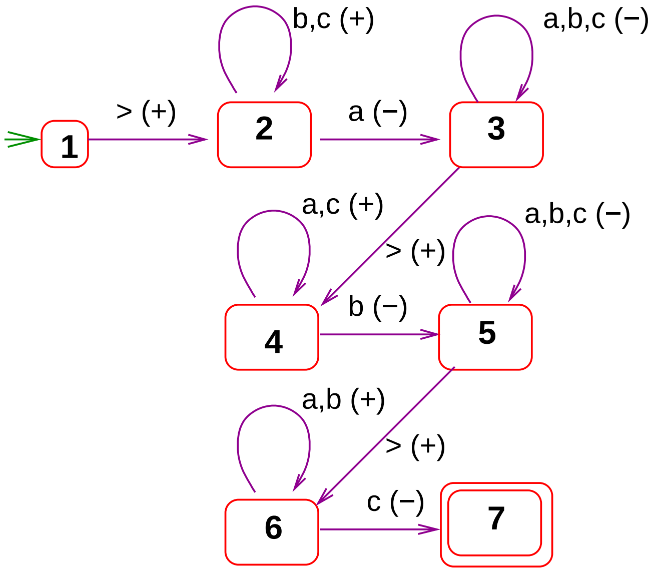
Two-way automata

- $\delta: \; Q \times \Gamma \;\rightharpoonup\; Q \times \mathbf{Act}$
  where $\Gamma = \Sigma \cup \{>, \sqcup\}$ and $\mathbf{Act} = \{+, -\}$.
- Write $q \overset{\sigma(\alpha)}{\rightarrow} p$ for $\delta(q, \sigma) = \langle p, \alpha \rangle$

The intent:

- Γ end-markers $\boxed{> \textit{ (gate)}}$ and $\boxed{\sqcup \textit{ (blank)}}$ added to Σ
- Example: Input 001201 appears as $>001201\sqcup$
- The $\boxed{\textit{actions}}$ $+$ and $-$ stand for “step forward” and “step back.”

Example: The strings using all of `a,b,c`



- With 26 in place of 3 we'd have 53 states,
  as opposed to $> 67,000,000$ states in the smallest DFA!

Operation of 2DFAs: configurations

- For DFAs we could generate the relation $p \xrightarrow{w} q$ inductively, as a function of w.
- This is no longer the case for 2DFAs: here we *must* account for the cursor position and keep record of the entire input for future use.
- A ***cursored-string*** over Σ is a $\Sigma-$string with one underlined symbol-position.
- A ***configuration (cfg)*** is a pair $(q, \check{w})$ where
  - ⋆ q is a state, and
  - ⋆ $\check{w}$ is a cursored-string,

    That is, $($ state, cursored-string $)$.
- Example: $(q, >0101\underline{1}00\sqcup)$
- The ***initial cfg for input w*** is the cfg $(s, \underline{>}w\sqcup)$.

The YIELD relation

- The ***Yield*** relation $\Rightarrow$
  (or $\Rightarrow_M$ when it matters which M) is obtained by:
-
  - ⋆ If $q \overset{\gamma(+)}{\longrightarrow} p$
    then $(q, u\underline{\gamma}\tau v) \;\Rightarrow\; (p, u\gamma\underline{\tau}v)$
  - ⋆ If $q \overset{\gamma(-)}{\longrightarrow} p$
    then $(q, u\tau\underline{\gamma}v) \;\Rightarrow\; (p, u\underline{\tau}\gamma v)$
  - ⋆ Nothing else
- If the given cfg is $(q, 01101\underline{0})$,
  and $q \overset{0(+)}{\longrightarrow} p$, then the transition above does not apply.

  The same holds when invoking a transition $q \overset{0(-)}{\longrightarrow} p$
  for a configuration with a cursor at the head of the string, such as $(q, \underline{0}11010)$.

Traces, acceptance, recognition

- A cfg $c = (q, u\gamma v)$ is ***terminal*** if no transition applies (no yield). It is a ***accepting*** its state is accepting state a.
- A ***trace*** of M for input w is a sequence of

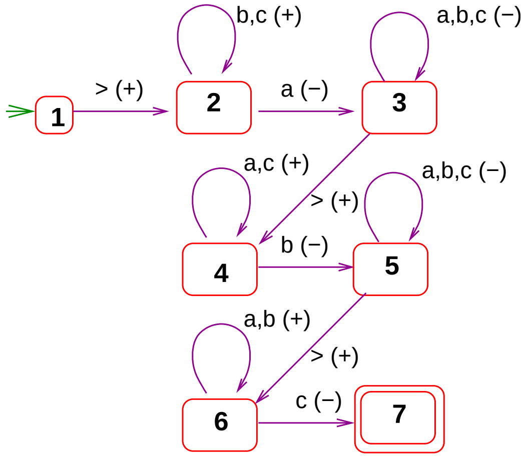
  $$c_0 \Rightarrow c_1 \Rightarrow \cdots$$

  where c_0 is initial for w, and either

  1. the sequence is infinite; or
  2. the sequence is finite, and its last cfg is terminal.
- The trace is ***accepting*** if it is finite and its last cfg is accepting.
- M ***accepts*** $w \in \Sigma^*$ if it its trace for input w is accepting.
- The language ***recognized*** by M is

  $$\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

Example



Accepting trace for trace of M above for $w = \mathbf{bcab}$:

$$(1, \underline{>}\mathbf{bcab}\sqcup)$$

$\Rightarrow (2, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$

$\Rightarrow (2, >\mathtt{b}\underline{\mathtt{c}}\mathtt{ab}\sqcup)$

$\Rightarrow (2, >\mathtt{bc}\underline{\mathtt{a}}\mathtt{b}\sqcup)$

$\Rightarrow (3, >\mathtt{b}\underline{\mathtt{c}}\mathtt{ab}\sqcup)$

$\Rightarrow (3, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$

$\Rightarrow (3, \underline{>}\mathtt{bcab}\sqcup)$

$\Rightarrow (4, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$

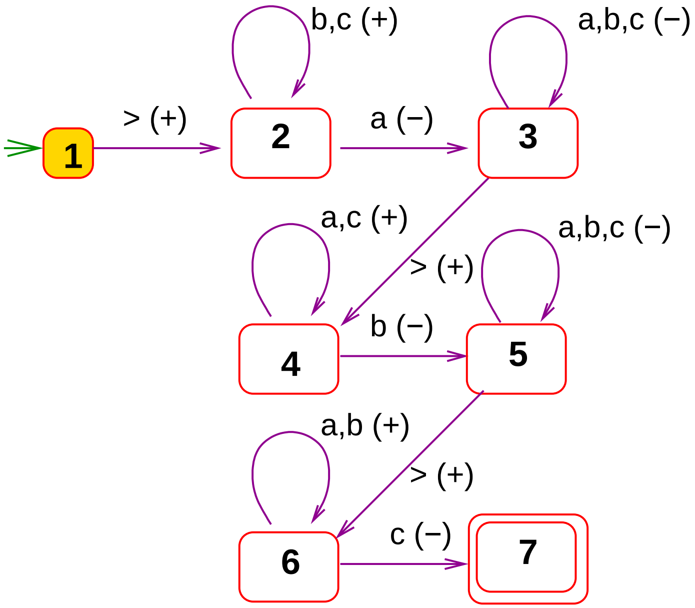
$\Rightarrow (5, \underline{>}\mathtt{bcab}\sqcup)$

$\Rightarrow (6, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$

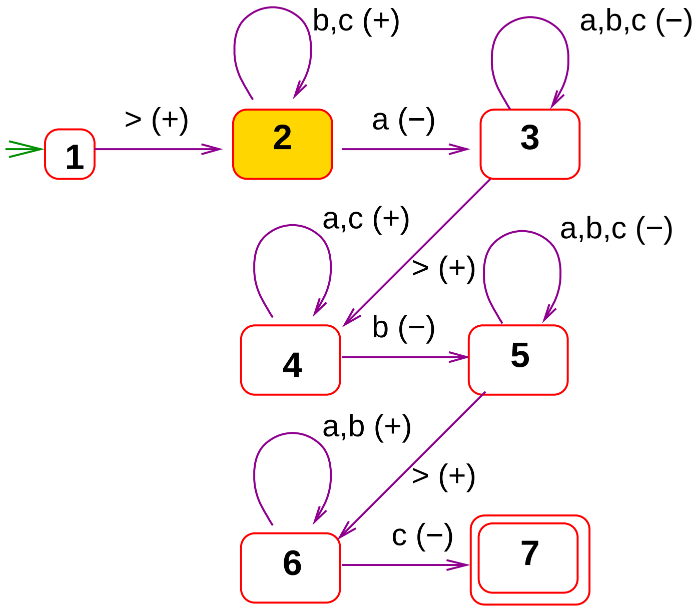
$\Rightarrow (6, >\mathtt{b}\underline{\mathtt{c}}\mathtt{ab}\sqcup)$

$\Rightarrow (7, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$

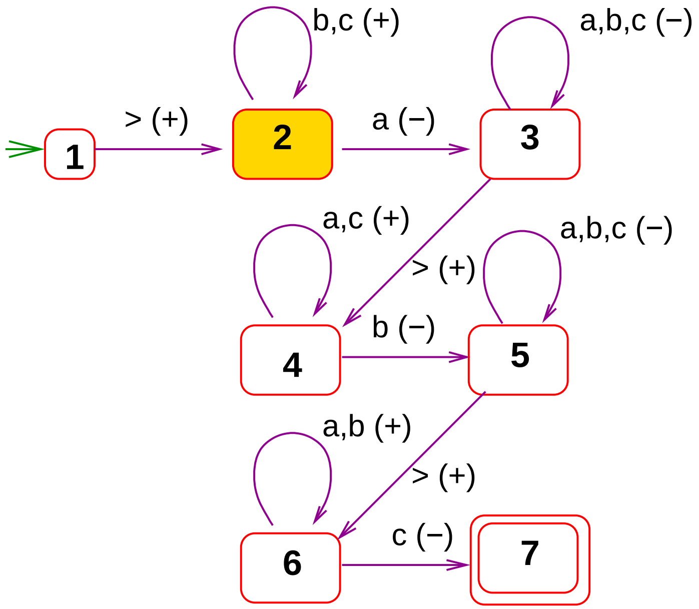
$(1, \underline{>}\mathbf{bcab}\sqcup)$



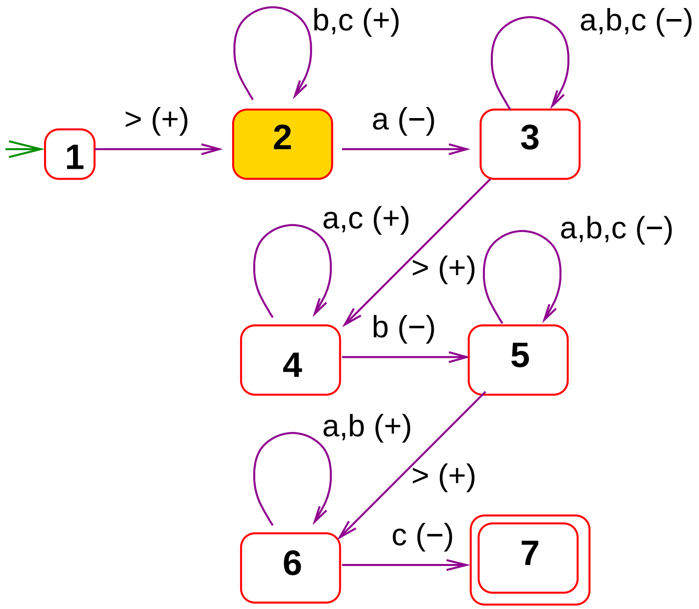
$(2,\ >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$



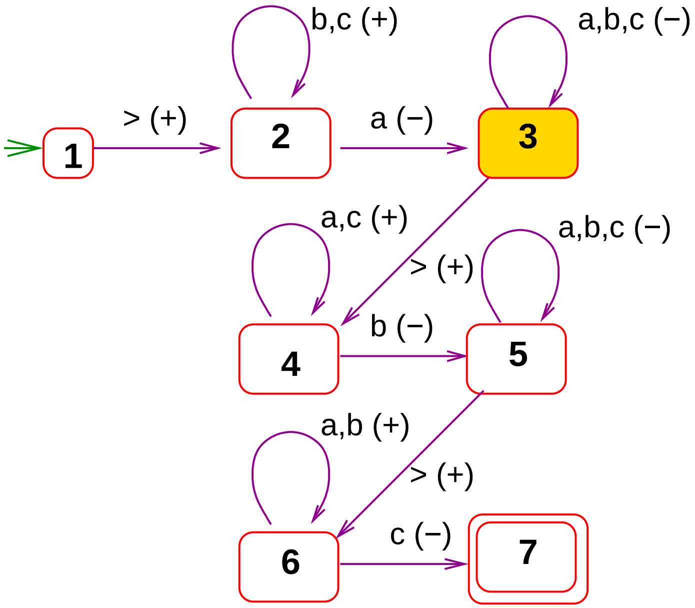
$(2, >\mathrm{b}\underline{\mathrm{c}}\mathrm{ab}\sqcup)$



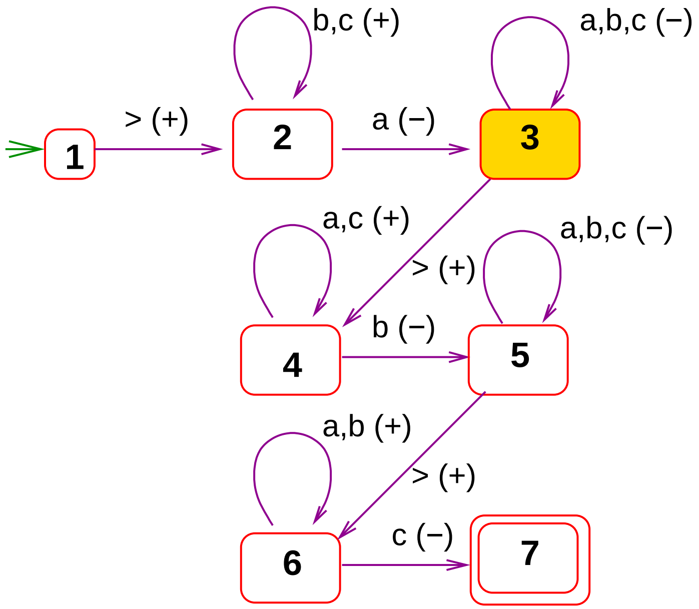
$(2, >\mathrm{bc}\underline{\mathrm{a}}\mathrm{b}\sqcup)$



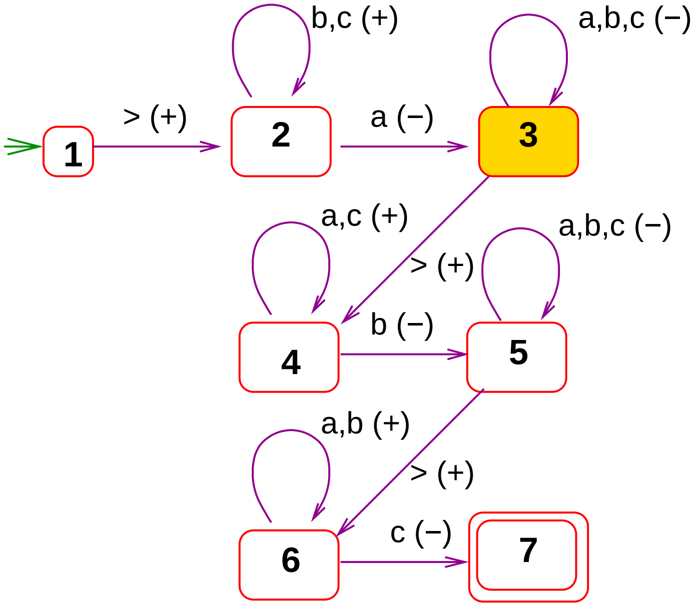
$(3,\ >b\underline{c}ab\sqcup)$



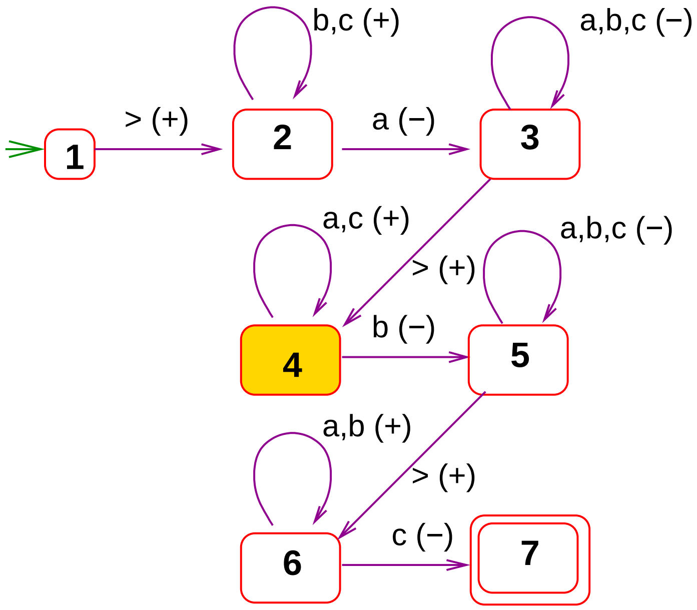
$(3, >\underline{\mathtt{b}}\mathtt{cab}\sqcup)$



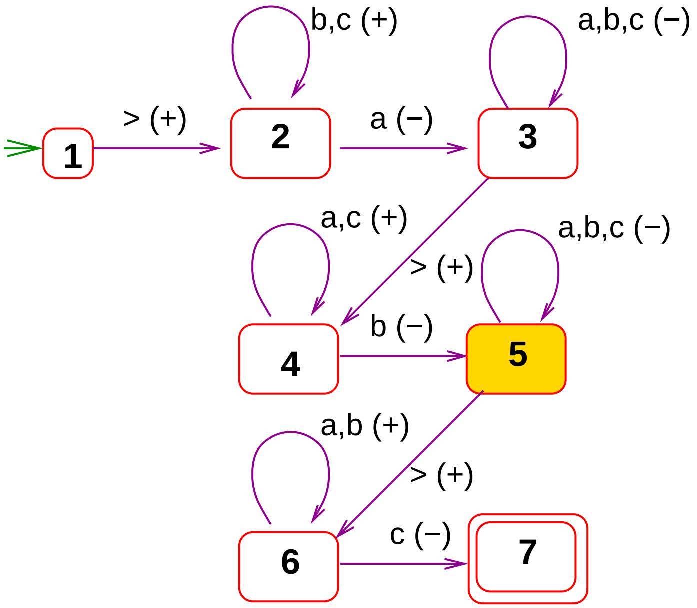
$(3, \underline{>}\mathtt{bcab}\sqcup)$



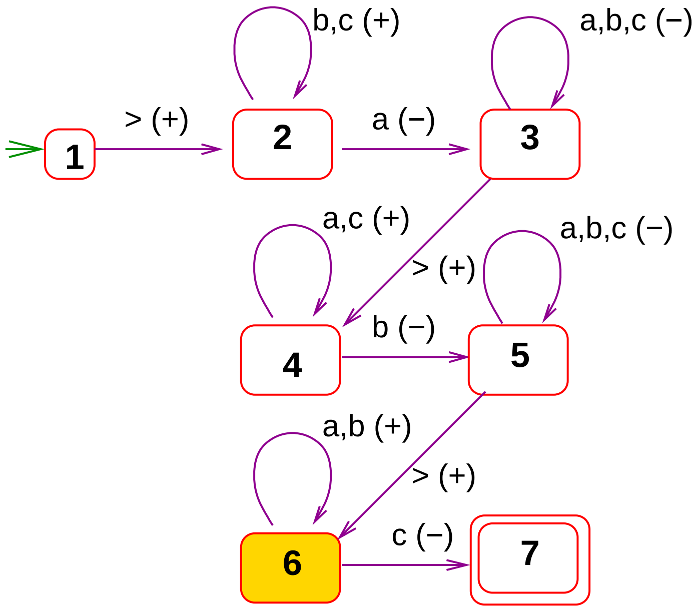
$(4,\ >\underline{b}cab\sqcup)$



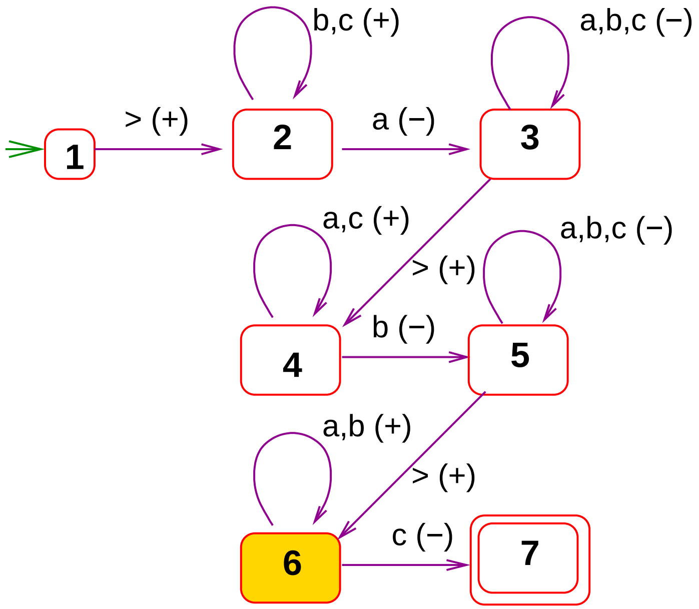
$(5, \underline{>}\text{bcab}\sqcup)$



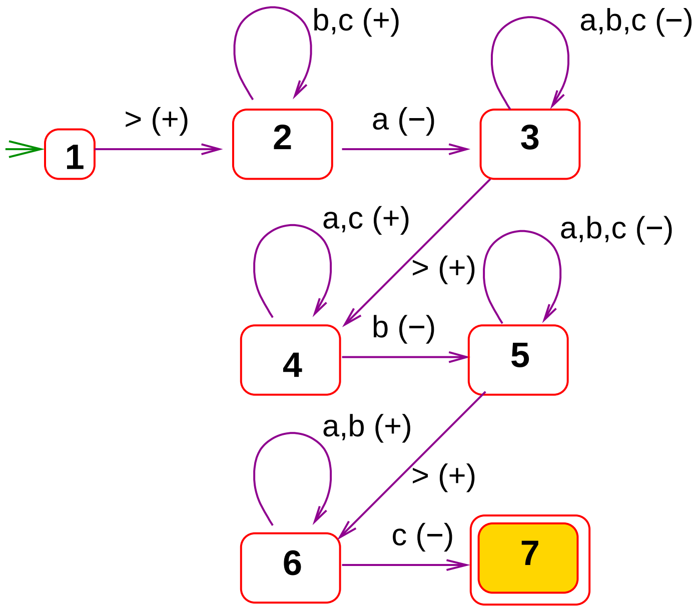
$(6, >\underline{b}cab\sqcup)$



$(6, >\mathrm{b}\underline{\mathrm{c}}\mathrm{ab}\sqcup)$



$(7, >\underline{b}cab\sqcup)$

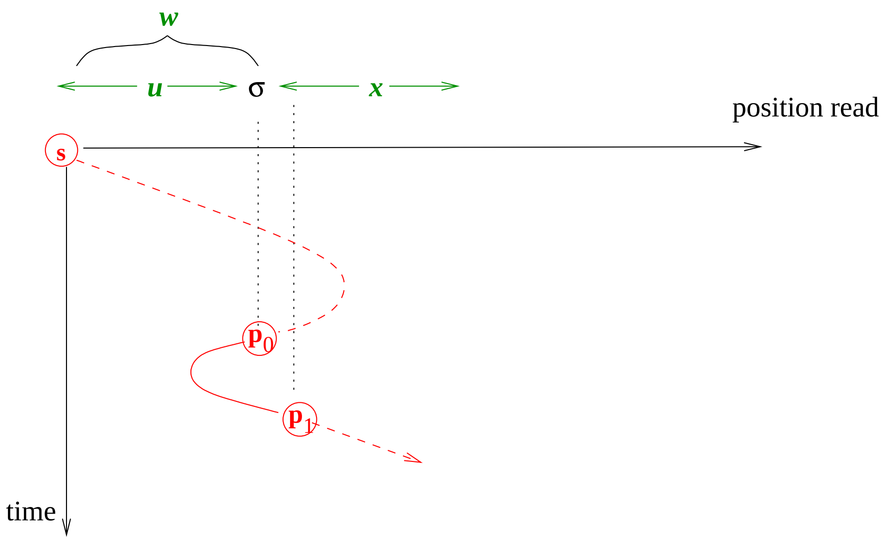


Two-way automata recognize just regular languages!

- Yet another characterization of regular languages!
- Adding nondeterminism to 2DFA still recognizes just regular languages!
- We still avoid extensible memory, so this is not a big surprise.

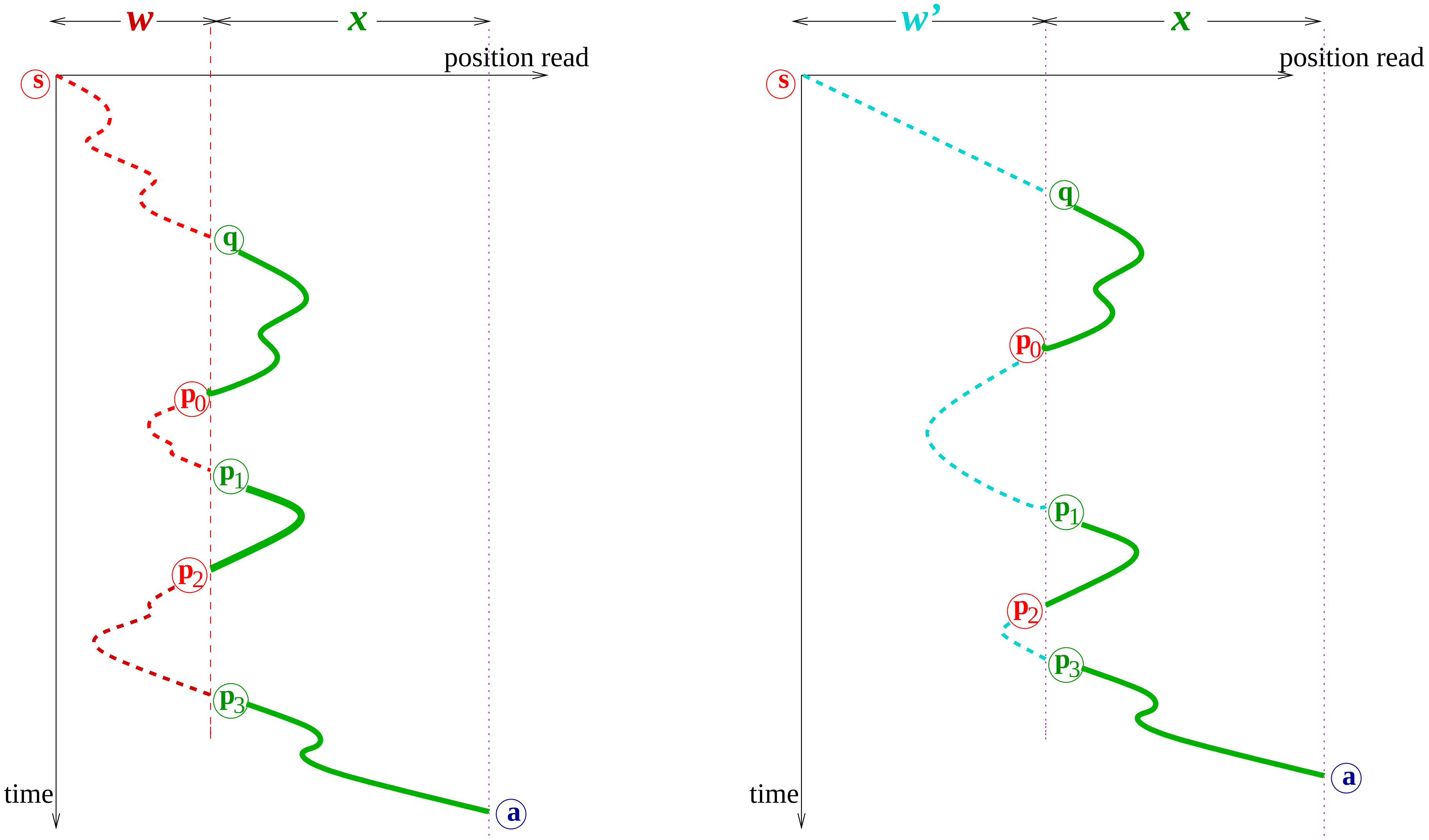
Proof outline

- DFA recognize languages with finitely many residues L/w.
- For each w a finite amount of info suffices to decide $x \in L/w$.
- For DFA the info is the state q reached: $s \xrightarrow{w} q$.
- For 2DFA the scan might cross out of w and into x .
  back in, and then out again into x.
- This is the info needed about w:
  If the reading cross back into w in a state
- The extra info:
  the pairs $(\textit{in}, \textit{out})$ of states
  s.t. crossing back into w in state *in*
  leads to crossing back out in state *out*.


w
u
σ
x
position read
s
p$_0$
p$_1$
time

Language recognized is regular!

- Say that $\langle p_0, p_1 \rangle$ is a *back-crossing pair.*
- L/w is determined by q reached by reading w,
  plus the set of back-crossing pairs for w:
  if w, w' reach the same state,
  and have the same crossing pairs, then $L/w = L/w'$.



x **in** L/w **IFF** x **in** L/w'

- For M with k states
  there are k^2 potential back-crossing pairs,
  and so 2^{k^2} possible descriptions of the situation at the border.
- Finitely many residues, albeit a lot, but still
  recognizing a regular language!

REGULARITY

The many facets of regularity

- Big equivalence of language properties, relating definitional to structural as well as computational properties.

The many facets of regularity

- Big equivalence of language properties, relating definitional to structural as well as computational properties.

$$\begin{aligned}\text{Regular} &\iff \text{Strictly-Regular}\\ &\iff \text{DFA-recognized}\\ &\iff \text{2DFA-recognized}\\ &\iff \text{NFA-recognized}\\ &\iff \text{has finitely many residues}\end{aligned}$$

- Another important characterization of regular languages is related to our automata-construction method.

The many facets of regularity

- Big equivalence of language properties, relating definitional to structural as well as computational properties.

  Regular $\iff$ Strictly-Regular
  $\iff$ DFA-recognized
  $\iff$ 2DFA-recognized
  $\iff$ NFA-recognized
  $\iff$ has finitely many residues

- Another important characterization of regular languages is related to our automata-construction method.
- One disappointment: It's all about languages and acceptors.
  What about functions and transducers?

The many facets of regularity

- Big equivalence of language properties, relating definitional to structural as well as computational properties.

Regular ⟺ Strictly-Regular
⟺ DFA-recognized
⟺ 2DFA-recognized
⟺ NFA-recognized
⟺ has finitely many residues

- Another important characterization of regular languages is related to our automata-construction method.
- One disappointment: It's all about languages and acceptors. What about functions and transducers?

FINITE STATE TRANSDUCERS

Finite-state transducers

- In a 2DFA the transition mapping indicates a choice of action: step forward or backward.

  In a deterministic ***finite-state transducer (DFT)*** the choice of action is an output string to be appended to an output device.

-

Finite-state transducers

- In a 2DFA the transition mapping indicates a choice of action: step forward or backward.

  In a deterministic **finite-state transducer (DFT)** the choice of action is an output string to be appended to an output device.

-

- Examples.

  - Double zeros: Input alphabet: 0,1.
    The DFS outputs 00 for 0 and 1 for 1.

Finite-state transducers

- In a 2DFA the transition mapping indicates
  a choice of action: step forward or backward.

  In a deterministic ***finite-state transducer (DFT)*** the choice
  of action is an output string to be appended to an output device.

-

- Examples.

  - Double zeros: Input alphabet: 0,1.
    The DFS outputs 00 for 0 and 1 for 1.

  - Input alphabet: English words
    Output: phonetic text.
    DFS outputs for each word its pronunciation.

Finite-state transducers

- In a 2DFA the transition mapping indicates
  a choice of action: step forward or backward.

  In a deterministic ***finite-state transducer (DFT)*** the choice
  of action is an output string to be appended to an output device.

-

- Examples.

  - ▸ Double zeros: Input alphabet: 0,1.
    The DFS outputs 00 for 0 and 1 for 1.
  - ▸ Input alphabet: English words
    Output: phonetic text.
    DFS outputs for each word its pronunciation.
  - ▸ Input alphabet: Latin.
    Output: Blanks replaced by ASCII $<$ *newline* $>$.

Formal definition of DFTs

- A ***deterministic finite-state transducer (DFT)*** consists of
  - Two alphabets Σ and Γ (possibly the same);

Formal definition of DFTs

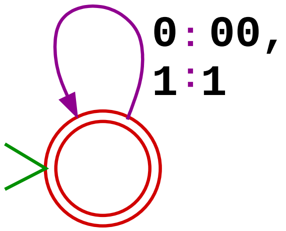
- A ***deterministic finite-state transducer (DFT)*** consists of
  - Two alphabets Σ and Γ (possibly the same);
  - A finite non-empty set Q of ***states***;
  - An ***initial*** (or "***start***") state $s \in Q$;

Formal definition of DFTs

- A ***deterministic finite-state transducer (DFT)*** consists of
  - Two alphabets Σ and Γ (possibly the same);
  - A finite non-empty set Q of ***states***;
  - An ***initial*** (or ***"start"***) state $s \in Q$;
  - A partial-function $\delta : Q \times \Sigma \rightharpoonup \Gamma^* \times Q$.

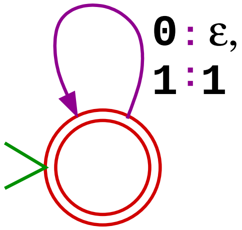
Examples

- Double zeros: The input is a binary string.
  Output: 00 for each 0 read and 1 for 1.



Examples

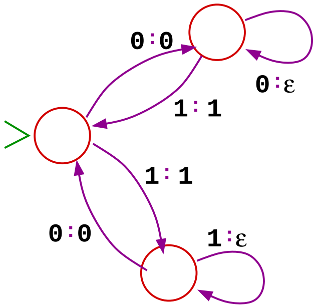
- Delete zeros: The input is a binary string.
  Output: ε for each 0 read and 1 for 1.



Examples

- Delete duplicate letters: The input is binary.

  Output: Remove duplicates, e.g. **001110** ↦ **010**.



Computing over streams

- A Given a set S a $\boxed{\textit{stream over } \Sigma}$ (or Σ***-stream***) is function $f : \mathbb{N} \to S$, i.e. an infinite sequence $a_0, a_1, \ldots$ where $a_i \in S$.
  (Alternative names: ω -strings, ω -words.)

Computing over streams

- A Given a set S a **stream over Σ** (or Σ-***stream***) is function $f : \mathbb{N} \to S$, i.e. an infinite sequence $a_0, a_1, \ldots$ where $a_i \in S$.
  (Alternative names: ω-strings, ω-words.)
- Example, every real number $a \in [0..1]$ has a decimal expansion as a stream $.a_0 a_1 a_2 \ldots$ over the decimal digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Computing over streams

- A Given a set S a $\boxed{\textit{stream over } \Sigma}$ (or Σ***-stream***) is function $f : \mathbb{N} \to S$, i.e. an infinite sequence $a_0, a_1, \ldots$ where $a_i \in S$. (Alternative names: ω -strings, ω -words.)
- Example, every real number $a \in [0..1]$ has a decimal expansion as a stream $.a_0a_1a_2\ldots$ over the decimal digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. E.g. 1 is $.9999\ldots$, $\sqrt{2}/2$ is $.70710678118\ldots$ and $\pi/10$ is $.3141592653\cdots$.

Running DFA’s on streams: Büchi acceptors

- Running DFT’s on streams is obvious,
  since termination plays no direct role in their running.
  But what about DFA’s?
  How is an input stream to be ***“accepted”***?

Running DFA's on streams: Büchi acceptors

- Running DFT's on streams is obvious,
  since termination plays no direct role in their running.
  But what about DFA's?
  How is an input stream to be ***"accepted"***?

- How about considering input stream α
  to be "accepted" by M if the execution of M on α
  has an accepting state?

Running DFA's on streams: Büchi acceptors

- Running DFT's on streams is obvious,
  since termination plays no direct role in their running.
  But what about DFA's?
  How is an input stream to be ***"accepted"***?
- How about considering input stream α
  to be "accepted" by M if the execution of M on α
  has an accepting state?
- Bad idea: It goes counter to the accepance of strings!

Running DFA’s on streams: Büchi acceptors

- Running DFT’s on streams is obvious,
  since termination plays no direct role in their running.
  But what about DFA’s?
  How is an input stream to be ***“accepted”***?
- What about M being in an accepting state from a certain step and on?

Running DFA's on streams: Büchi acceptors

- Running DFT's on streams is obvious,
  since termination plays no direct role in their running.

  But what about DFA's?
  How is an input stream to be ***"accepted"***?
- What about M being in an accepting state from a
  certain step and on?
- Also bad:
  Acceptance is then determined by a prefix of the input.

Running DFA’s on streams: Büchi acceptors

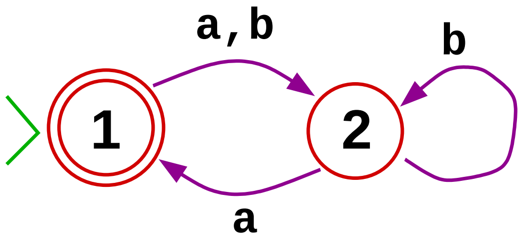
- Running DFT’s on streams is obvious,
  since termination plays no direct role in their running.

  But what about DFA’s?
  How is an input stream to be ***“accepted”***?

- The right idea (Büchi, 1962):
  Accept an input if its state-trace is in a “good” state
  infinitely many times.

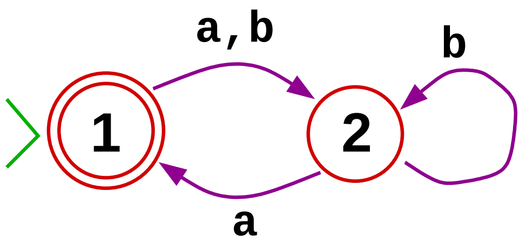
Example 1

Here’s a DFA.



Example 1

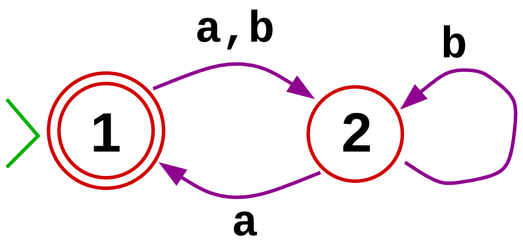
Here's a DFA.



- What language does it recognize?

Example 1

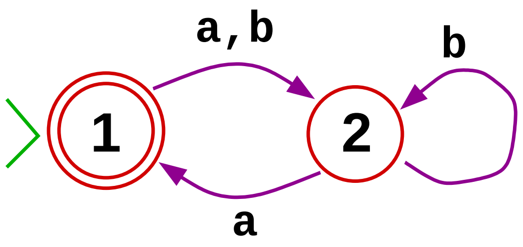
Here's a DFA.



- What language does it recognize?
- $((\mathtt{a} \cup \mathtt{b}) \cdot \mathtt{b}^* \cdot \mathtt{a})^*$.

Example 1

Here's a DFA.

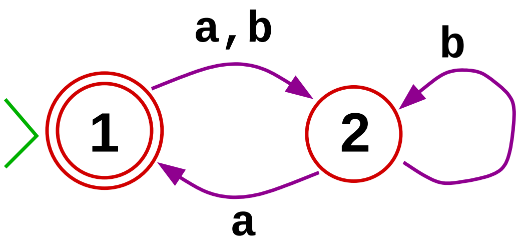


- What language does it recognize?
- $((\mathtt{a} \cup \mathtt{b}) \cdot \mathtt{b}^* \cdot \mathtt{a})^*$.

What streams are accepted?

Example 1

Here's a DFA.

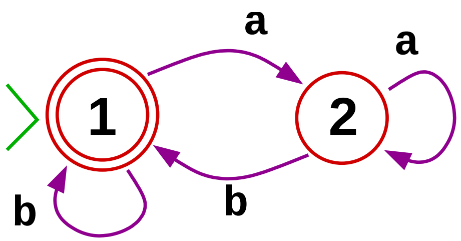


- What language does it recognize?
- $((a \cup b) \cdot b^* \cdot a)^*$.

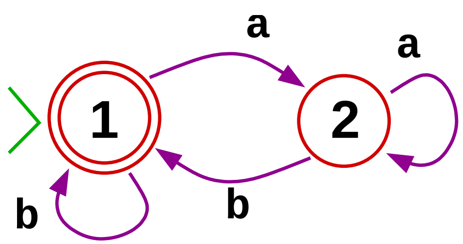
What streams are accepted?

With infinitely many a's.

Example 2

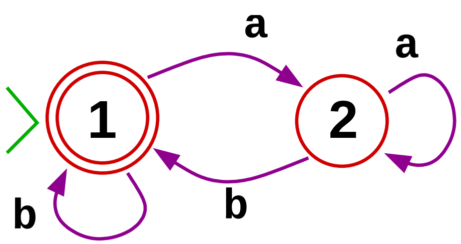


Example 2



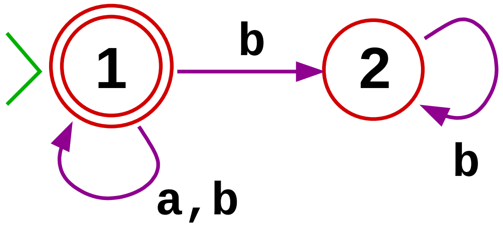
- What streams are accepted?

Example 2

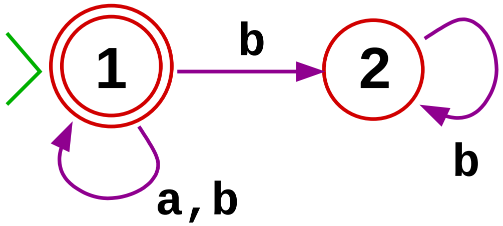


- What streams are accepted?
- Where every `a` is followed by some `b`.

Example 3

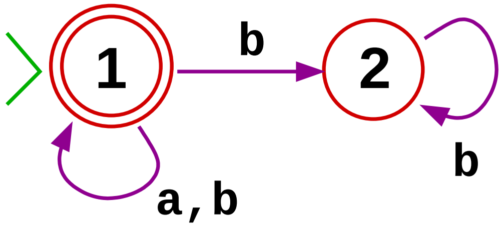


Example 3



- What stream are accepted?

Example 3



- What stream are accepted?
- With finitely many a’s.