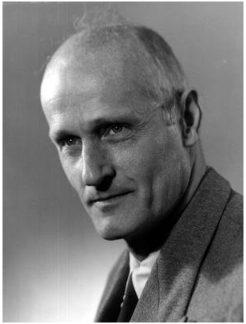
Informatics 1 – Introduction to Computation
Computation and Logic

Julian Bradfield
based on materials by
Michael P. Fourman

Finite State Machines



Stephen Kleene, 1909–1994
Photo: Harold Hone

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,
- and there are finitely many states.

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,
- and there are finitely many states.
- Which state we move to depends on input: a symbol drawn from a finite alphabet.

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,
- and there are finitely many states.
- Which state we move to depends on input: a symbol drawn from a finite alphabet.
- Some states are accepting: if the machine is there at end of input, that's good, otherwise it's bad.

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,
- and there are finitely many states.
- Which state we move to depends on input: a symbol drawn from a finite alphabet.
- Some states are accepting: if the machine is there at end of input, that's good, otherwise it's bad.
- There's an identified start state.

Very Simple Computers

We're going to switch tack, and start looking at more physical models of computation: computation as machinery, rather than computation as logic.

We'll start with the simplest model of computation we can devise:

- the machine will just move from one state to another,
- and there are finitely many states.
- Which state we move to depends on input: a symbol drawn from a finite alphabet.
- Some states are accepting: if the machine is there at end of input, that's good, otherwise it's bad.
- There's an identified start state.

These (give or take a technicality) are Finite Automata, or Finite State Machines.

Applications

FA have countless applications:

- washing machine/central heating/etc. controllers
- traffic light controllers
- parsing programming languages
- CPU controllers
- natural language processing
- ...

Applications

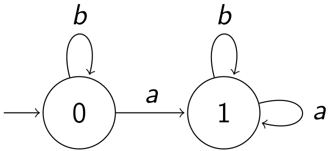
FA have countless applications:

- washing machine/central heating/etc. controllers
- traffic light controllers
- parsing programming languages
- CPU controllers
- natural language processing
- ...

Is your laptop a finite automaton? Is anything not a finite automaton?

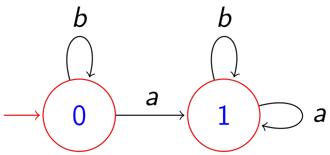
Drawing FAs

We often think about FAs by drawing them:



Drawing FAs

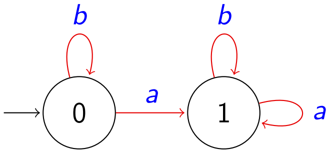
We often think about FAs by drawing them:



The circles are the states, with their names: the set of states is $\{0, 1\}$.

Drawing FAs

We often think about FAs by drawing them:

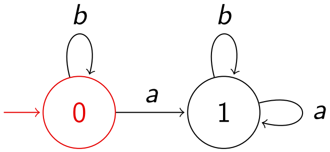


The circles are the states, with their names: the set of states is $\{0, 1\}$.

The connecting arrows are the transitions, with the input letter that activates them: the input alphabet is $\{a, b\}$.

Drawing FAs

We often think about FAs by drawing them:



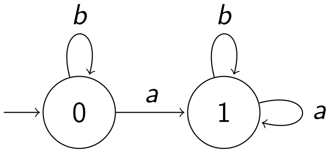
The circles are the states, with their names: the set of states is $\{0, 1\}$.

The connecting arrows are the transitions, with the input letter that activates them: the input alphabet is $\{a, b\}$.

The short arrow marks the initial or start state.

Drawing FAs

We often think about FAs by drawing them:



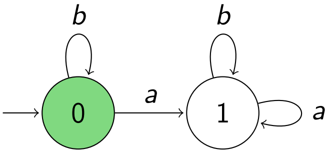
The circles are the states, with their names: the set of states is $\{0, 1\}$.

The connecting arrows are the transitions, with the input letter that activates them: the input alphabet is $\{a, b\}$.

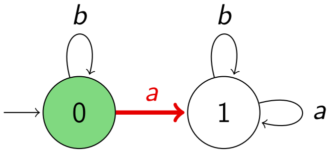
The short arrow marks the initial or start state.

This machine reads b until it reads an a, after which it reads a or b for ever.

FA behaviour



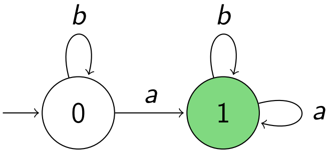
FA behaviour



If we feed the machine *ab*:

- in state 0, *a* fires the right transition

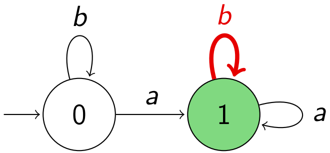
FA behaviour



If we feed the machine *ab*:

- in state 0, *a* fires the right transition and the state changes to 1

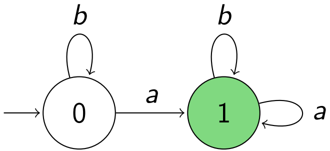
FA behaviour



If we feed the machine *ab*:

- in state 0, *a* fires the right transition and the state changes to 1
- then from state 1, *b* fires the top transition

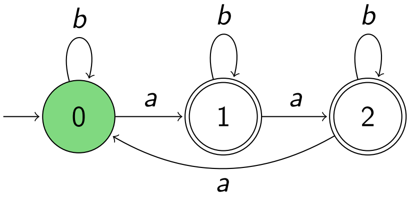
FA behaviour



If we feed the machine *ab*:

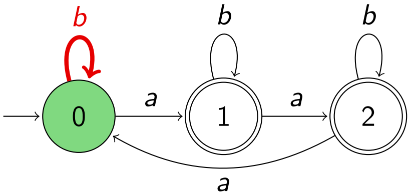
- in state 0, *a* fires the right transition and the state changes to 1
- then from state 1, *b* fires the top transition and the state remains 1

FAs with accepting states



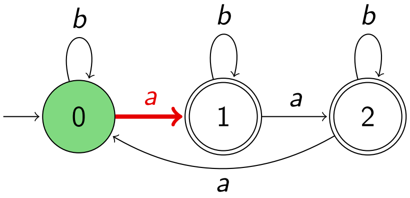
On input *babaaba* we see:

FAs with accepting states



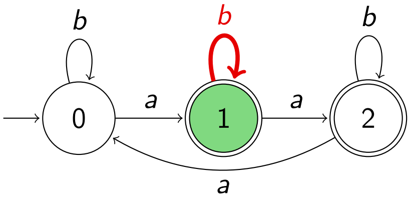
On input *babaaba* we see:

FAs with accepting states



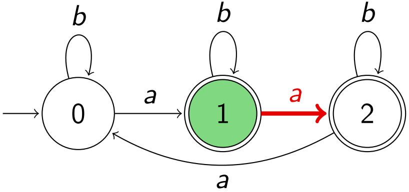
On input *babaaba* we see:

FAs with accepting states



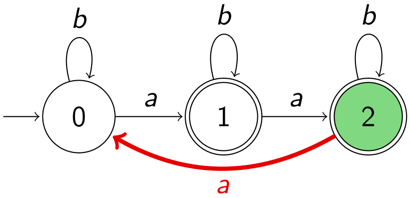
On input *babaaba* we see:

FAs with accepting states



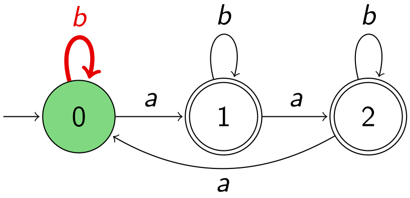
On input *babaaba* we see:

FAs with accepting states



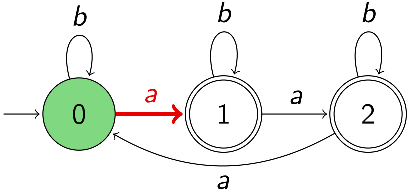
On input *baba**a**ba* we see:

FAs with accepting states



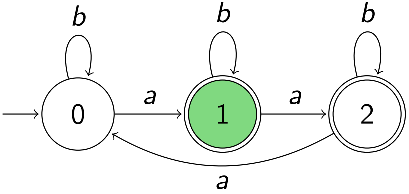
On input *babaaba* we see:

FAs with accepting states



On input *babaaba* we see:

FAs with accepting states



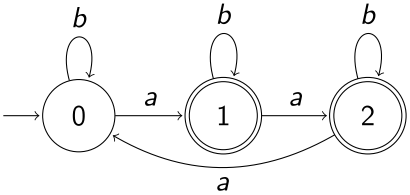
On input *babaaba* we see:

ending up in an accepting state.

We say that the automata has accepted the string *babaaba*.

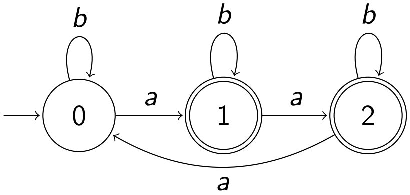
If the automaton ends in a non-accepting state, it has rejected the string. Verify for yourself that this automaton rejects *babbaa*.

Automata accepting languages



What input strings does this automaton accept?

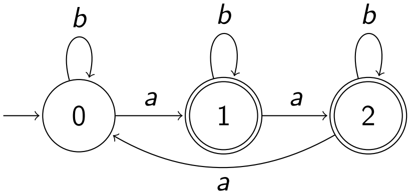
Automata accepting languages



What input strings does this automaton accept?

Any number of *b*s then an *a* and any number of *b*s, then optionally an *a* and any number of *b*s, then two *a*s followed by everything all over again.

Automata accepting languages

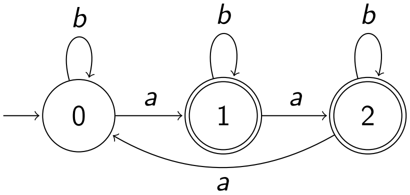


What input strings does this automaton accept?

Any number of *b*s then an *a* and any number of *b*s, then optionally an *a* and any number of *b*s, then two *a*s followed by everything all over again.

That's a little hard to understand: we will see later how to turn this into a precise description.

Automata accepting languages



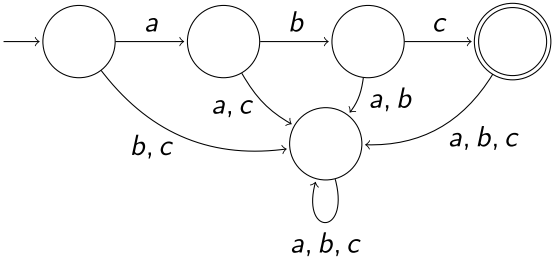
What input strings does this automaton accept?

Any number of *b*s then an *a* and any number of *b*s, then optionally an *a* and any number of *b*s, then two *a*s followed by everything all over again.

That's a little hard to understand: we will see later how to turn this into a precise description.

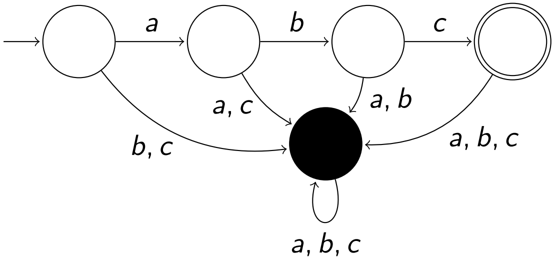
If instead we think about it, we see: the state labels $0, 1, 2$ count how many *a*s we have seen, modulo 3. The automaton accepts any string of *a*s and *b*s where the number of *a*s is not a multiple of 3.

Space saving conventions



Writing multiple labels a, c is shorthand for an a-transition and a c-transition.

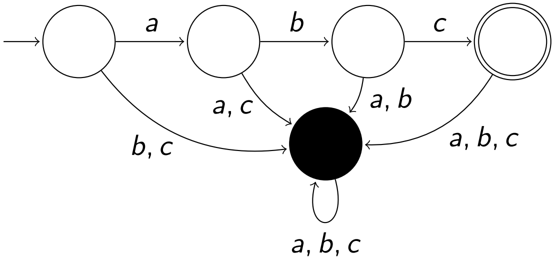
Space saving conventions



Writing multiple labels a, c is shorthand for an a-transition and a c-transition.

The bottom state is a black hole state: once there, the machine never leaves.

Space saving conventions

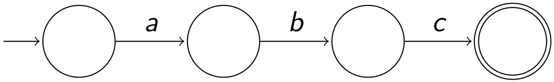


Writing multiple labels a, c is shorthand for an a-transition and a c-transition.

The bottom state is a black hole state: once there, the machine never leaves.

The black hole convention says that if you don't write a transition for letter a from state q, there is an implicit a-transition from q to a non-accepting black hole state.

Space saving conventions



Writing multiple labels a, c is shorthand for an a-transition and a c-transition.

The bottom state is a black hole state: once there, the machine never leaves.

The black hole convention says that if you don't write a transition for letter a from state q, there is an implicit a-transition from q to a non-accepting black hole state.

Determinism

So far, our automata have had

- a single start state
- exactly one transition from each state for each input letter

at most one transition, if we use the black hole convention

Such automata are called deterministic, because their next move is fully determined by the input letter. Later, we'll see non-deterministic automata, but for now we stick with DFAs.

Formalism

There are several ways to mathematize DFAs. Here's one:

A DFA comprises:

- A finite set Q of states
- A finite alphabet Σ of input letters
- A transition function $\delta\colon Q \times \Sigma \to Q$
- A starting state $q_0 \in Q$
- A subset $F \subseteq Q$ of accepting (or final) states

The use of F for 'final' states is traditional.

Building DFAs

The states of a DFA are its memory. Using this fact is the easiest way to construct a DFA: first think about the states, then the transitions. Example:

Building DFAs

The states of a DFA are its memory. Using this fact is the easiest way to construct a DFA: first think about the states, then the transitions. Example:

Build a DFA over the alphabet $\{0, 1\}$ that accepts strings with an even number of zeros and an odd number of ones.

If you already know about regular expressions, can you describe this language by a regexp?

Building DFAs

The states of a DFA are its memory. Using this fact is the easiest way to construct a DFA: first think about the states, then the transitions. Example:

Build a DFA over the alphabet $\{0, 1\}$ that accepts strings with an even number of zeros and an odd number of ones.

We need to track two *bits* of information: have we seen even/odd numbers of zeros/ones. One bit needs two states, two bits needs four states. So:

$$Q = \{ E_0 E_1, E_0 O_1, O_0 E_1, O_0 O_1 \}$$

If you already know about regular expressions, can you describe this language by a regexp?

Building DFAs

The states of a DFA are its memory. Using this fact is the easiest way to construct a DFA: first think about the states, then the transitions. Example:

Build a DFA over the alphabet $\{0, 1\}$ that accepts strings with an even number of zeros and an odd number of ones.

We need to track two *bits* of information: have we seen even/odd numbers of zeros/ones. One bit needs two states, two bits needs four states. So:

$$Q = \{ E_0 E_1, E_0 O_1, O_0 E_1, O_0 O_1 \}$$

Initially, we've read nothing: $q_0 = E_0 E_1$

If you already know about regular expressions, can you describe this language by a regexp?

Building DFAs

The states of a DFA are its memory. Using this fact is the easiest way to construct a DFA: first think about the states, then the transitions. Example:

Build a DFA over the alphabet $\{0, 1\}$ that accepts strings with an even number of zeros and an odd number of ones.

We need to track two *bits* of information: have we seen even/odd numbers of zeros/ones. One bit needs two states, two bits needs four states. So:

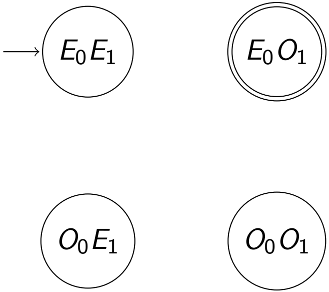
$$Q = \{ E_0 E_1, E_0 O_1, O_0 E_1, O_0 O_1 \}$$

Initially, we've read nothing: $q_0 = E_0 E_1$

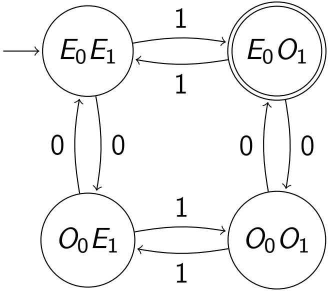
The accepting set is just $F = \{ E_0 O_1 \}$.

If you already know about regular expressions, can you describe this language by a regexp?

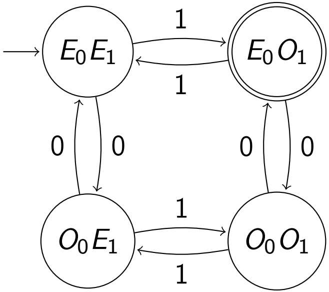
Even 0s and odd 1s



Even 0s and odd 1s



Even 0s and odd 1s



Writing it in symbols rather than diagrams:

$Q = \{\, E_0E_1, E_0O_1, O_0E_1, O_0O_1 \,\}$

$q_0 = E_0E_1$

$F = \{\, E_0O_1 \,\}$

δ is the following table:

	0	1
E_0E_1	O_0E_1	E_0O_1
E_0O_1	O_0O_1	E_0E_1
O_0E_1	E_0E_1	O_0O_1
O_0O_1	E_0O_1	O_0E_1

The book shows how bad things can get if you just try to work from an initial state by following your nose!

More formalism

We need a few notations and terms to talk more about DFAs:

- for any set Σ, Σ^* is the set of strings over Σ. The empty string is written ε. If $s \in \Sigma^*$ and $x \in \Sigma$, then xs is the string comprising x followed by s.

We'll use some lazy conventions on slides: M is $Q, \Sigma, \delta, q_0, F$, and M' is $Q', \Sigma', \delta', q_0', F'$ unless otherwise stated. Similarly M'', M_1, M_2 etc.

More formalism

We need a few notations and terms to talk more about DFAs:

- for any set Σ, Σ^* is the set of strings over Σ. The empty string is written ε. If $s \in \Sigma^*$ and $x \in \Sigma$, then xs is the string comprising x followed by s.
- If $\delta\colon Q \times \Sigma \to Q$ is the transition function, then $\delta^*\colon Q \times \Sigma^* \to Q$ is the string transition function defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, xs) = \delta^*(\delta(q, x), s)$

We'll use some lazy conventions on slides: M is $Q, \Sigma, \delta, q_0, F$, and M' is $Q', \Sigma', \delta', q_0', F'$ unless otherwise stated. Similarly M'', M_1, M_2 etc.

More formalism

We need a few notations and terms to talk more about DFAs:

- for any set Σ, Σ^* is the set of strings over Σ. The empty string is written ε. If $s \in \Sigma^*$ and $x \in \Sigma$, then xs is the string comprising x followed by s.
- If $\delta\colon Q \times \Sigma \to Q$ is the transition function, then $\delta^*\colon Q \times \Sigma^* \to Q$ is the string transition function defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, xs) = \delta^*(\delta(q, x), s)$
- If $\Sigma^* \ni s = a_1 \ldots a_n$, the trace of s is the sequence $q_0 \ldots q_n$ where $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$.

We'll use some lazy conventions on slides: M is $Q, \Sigma, \delta, q_0, F$, and M' is $Q', \Sigma', \delta', q_0', F'$ unless otherwise stated. Similarly M'', M_1, M_2 etc.

More formalism

We need a few notations and terms to talk more about DFAs:

- for any set Σ, Σ^* is the set of strings over Σ. The empty string is written ε. If $s \in \Sigma^*$ and $x \in \Sigma$, then xs is the string comprising x followed by s.
- If $\delta \colon Q \times \Sigma \to Q$ is the transition function, then $\delta^* \colon Q \times \Sigma^* \to Q$ is the string transition function defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, xs) = \delta^*(\delta(q, x), s)$
- If $\Sigma^* \ni s = a_1 \dots a_n$, the trace of s is the sequence $q_0 \dots q_n$ where $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$.
- The language accepted by M is $L(M) = \{s \in \Sigma^* : \delta^*(q_0, s) \in F\}$.

We'll use some lazy conventions on slides: M is $Q, \Sigma, \delta, q_0, F$, and M' is $Q', \Sigma', \delta', q_0', F'$ unless otherwise stated. Similarly M'', M_1, M_2 etc.

More formalism

We need a few notations and terms to talk more about DFAs:

- for any set Σ, Σ^* is the set of strings over Σ. The empty string is written ε. If $s \in \Sigma^*$ and $x \in \Sigma$, then xs is the string comprising x followed by s.
- If $\delta\colon Q \times \Sigma \to Q$ is the transition function, then $\delta^*\colon Q \times \Sigma^* \to Q$ is the string transition function defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, xs) = \delta^*(\delta(q, x), s)$
- If $\Sigma^* \ni s = a_1 \dots a_n$, the trace of s is the sequence $q_0 \dots q_n$ where $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$.
- The language accepted by M is $L(M) = \{s \in \Sigma^* : \delta^*(q_0, s) \in F\}$.
- A language $L \subseteq \Sigma^*$ is regular iff there is some DFA M over Σ that accepts L.

We'll use some lazy conventions on slides: M is $Q, \Sigma, \delta, q_0, F$, and M' is $Q', \Sigma', \delta', q_0', F'$ unless otherwise stated. Similarly M'', M_1, M_2 etc.

Building up automata: complement

We can use automata as building blocks in others (as we build formulae out of formulae. . .).

Start with *complement*: if M accepts L, how do we build a machine that accepts $\bar{L} = \Sigma^* - L$?

There are two common notations for set difference: $A - B$ and $A \setminus B$. They mean $\{x \in A : x \notin B\}$.

Building up automata: complement

We can use automata as building blocks in others (as we build formulae out of formulae...).

Start with *complement*: if M accepts L, how do we build a machine that accepts $\bar{L} = \Sigma^* - L$?

Easy: swap accepting and rejecting states:

- The complement of $M = (Q, \Sigma, \delta, q_0, F)$ is $\overline{M} = (Q, \Sigma, \delta, q_0, Q - F)$.

There are two common notations for set difference: $A - B$ and $A \setminus B$. They mean $\{x \in A : x \notin B\}$.

We must remember to include any black hole states that weren't drawn!

Building up automata: complement

We can use automata as building blocks in others (as we build formulae out of formulae...).

Start with *complement*: if M accepts L, how do we build a machine that accepts $\bar{L} = \Sigma^* - L$?

Easy: swap accepting and rejecting states:

- The complement of $M = (Q, \Sigma, \delta, q_0, F)$ is $\overline{M} = (Q, \Sigma, \delta, q_0, Q - F)$.

Hence we know that the set of regular languages is closed under complement.

There are two common notations for set difference: $A - B$ and $A \setminus B$. They mean $\{x \in A : x \notin B\}$.

We must remember to include any black hole states that weren't drawn!

The term *closed under ...* is common in algebra. Be sure to understand it.

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

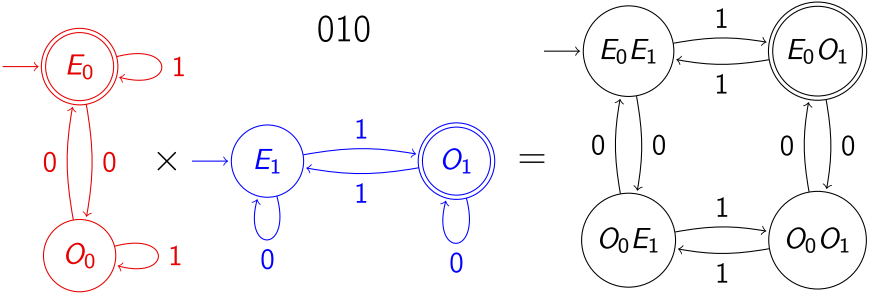
- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

010

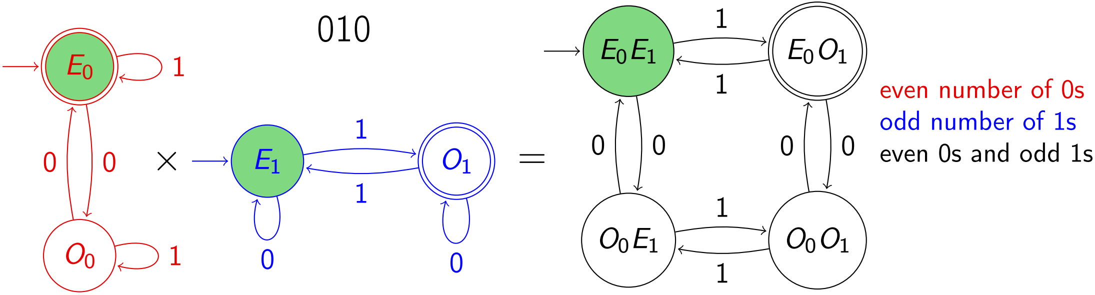


even number of 0s
odd number of 1s
even 0s and odd 1s

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

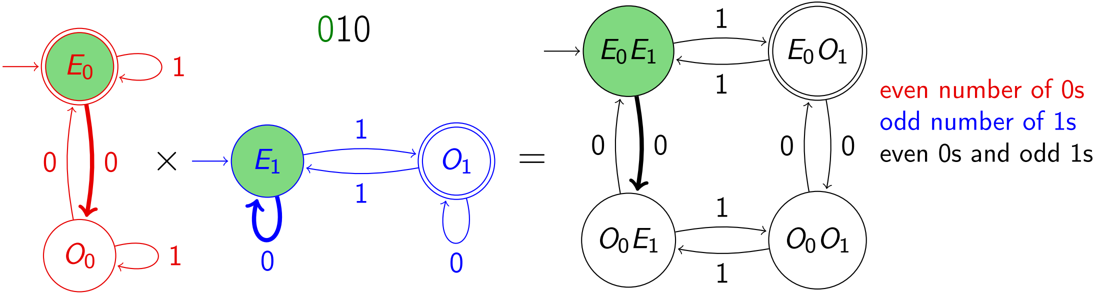
- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.



Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

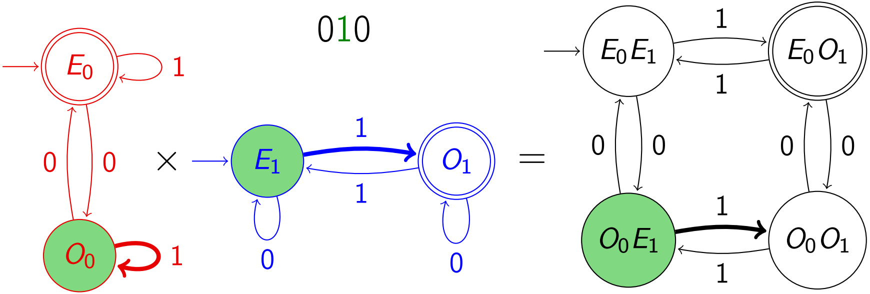
- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.



Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q'_0, F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q'_0), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

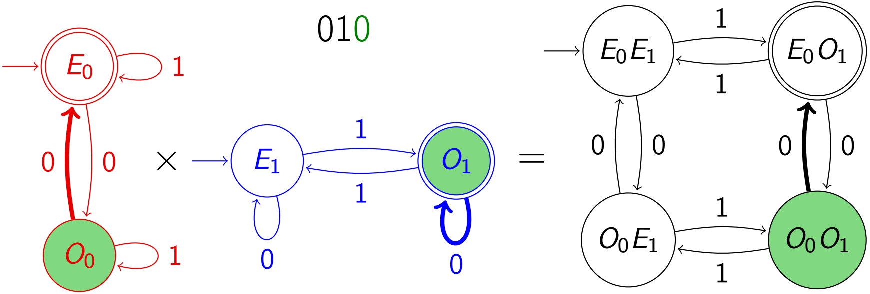


even number of 0s
odd number of 1s
even 0s and odd 1s

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q'_0, F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q'_0), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

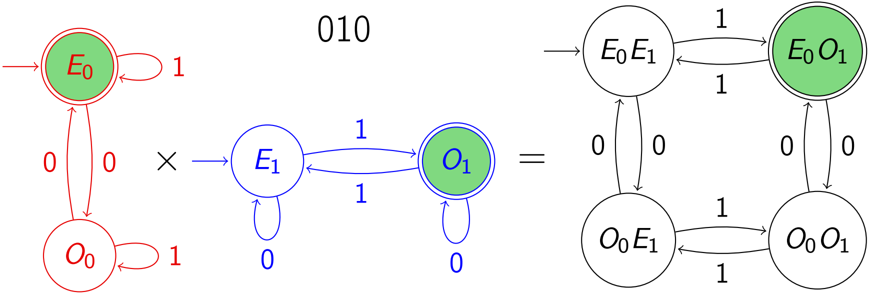


even number of 0s
odd number of 1s
even 0s and odd 1s

Building up automata: product

If M, M' accept L, L', can we make something accepting $L \cap L'$?
It's a bit trickier, but yes: we need somehow to feed input to M and M' at the same time.

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The product $M \times M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), F \times F')$ where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.



Notice that we can run M and M' in parallel without ever constructing all of $M \times M'$. This is *on the fly* construction.

Unfortunately, many things do need the whole automaton.

even number of 0s
odd number of 1s
even 0s and odd 1s

What about black hole states?

Building up automata: sum

If M, M' accept L, L', can we make something accepting $L \cup L'$?

Building up automata: sum

If M, M' accept L, L', can we make something accepting $L \cup L'$?
Yes, with almost the same construction:

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q'_0, F')$.
  The sum $M +_{\mathrm{d}} M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q'_0), Q \times F' \cup F \times Q')$
  where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

The difference is the accepting states: we accept if *either* component accepts. Hence $L(M +_{\mathrm{d}} M') = L(M) \cup L(M')$.

Later we will see a different sum for other automata. I'll write this one as $+_{\mathrm{d}}$ (d for deterministic).

Building up automata: sum

If M, M' accept L, L', can we make something accepting $L \cup L'$?
Yes, with almost the same construction:

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The sum $M +_{\mathrm{d}} M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), Q \times F' \cup F \times Q')$
  where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

The difference is the accepting states: we accept if *either* component accepts. Hence $L(M +_{\mathrm{d}} M') = L(M) \cup L(M')$.

$Q \times F' \cup F \times Q'$ can also be written as
$(Q \times Q') - ((Q - F) \times (Q' - F'))$ which we can notate $\overline{\overline{F} \times \overline{F'}}$.

Does this remind you of something?

Later we will see a different sum for other automata. I'll write this one as $+_{\mathrm{d}}$ (d for deterministic).

Building up automata: sum

If M, M' accept L, L', can we make something accepting $L \cup L'$?
Yes, with almost the same construction:

- Let $M = (Q, \Sigma, \delta, q_0, F)$, and $M' = (Q', \Sigma, \delta', q_0', F')$.
  The sum $M +_{\mathrm{d}} M'$ is $(Q \times Q', \Sigma, \delta'', (q_0, q_0'), Q \times F' \cup F \times Q')$
  where $\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a))$.

The difference is the accepting states: we accept if *either* component accepts. Hence $L(M +_{\mathrm{d}} M') = L(M) \cup L(M')$.

$Q \times F' \cup F \times Q'$ can also be written as
$(Q \times Q') - ((Q - F) \times (Q' - F'))$ which we can notate $\overline{\overline{F} \times \overline{F'}}$.

Does this remind you of something?

So now we know regular languages are closed under complement, intersection, and union.

Later we will see a different sum for other automata. I'll write this one as $+_{\mathrm{d}}$ (d for deterministic).

What about black hole states?