Informatics 1 – Introduction to Computation
Computation and Logic

Julian Bradfield
based on materials by
Michael P. Fourman

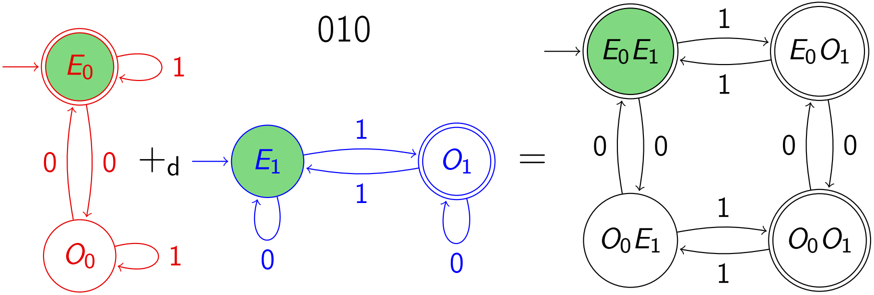
Non-determinism
and Regular Expressions

Michael Rabin, 1931–

Dana Scott, 1932–

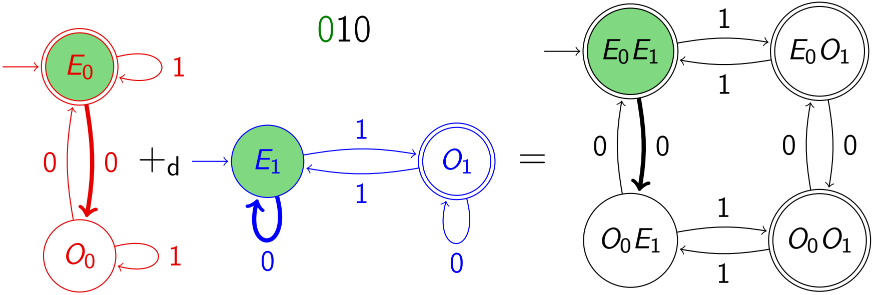
Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



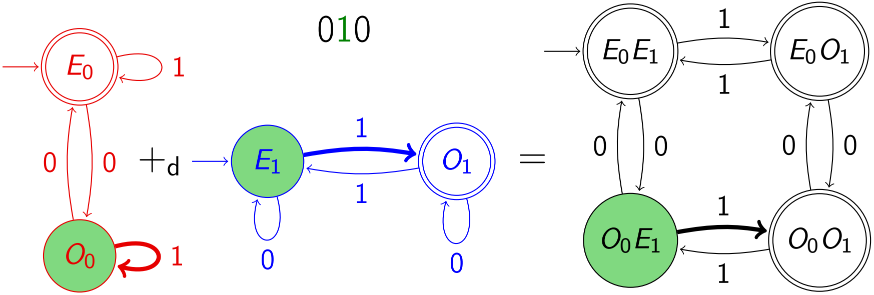
Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



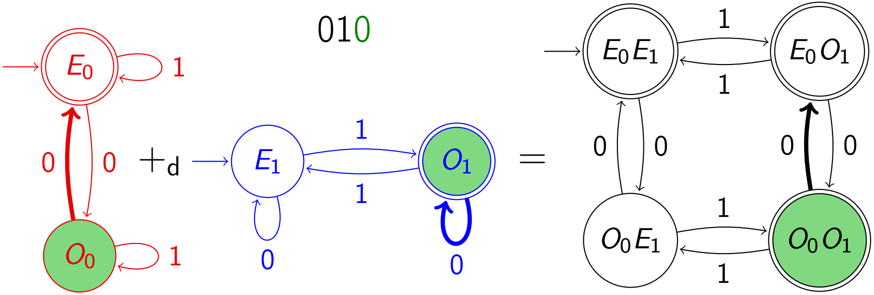
Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



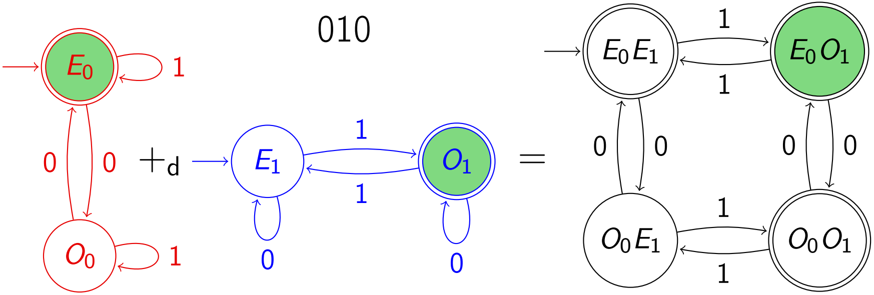
Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



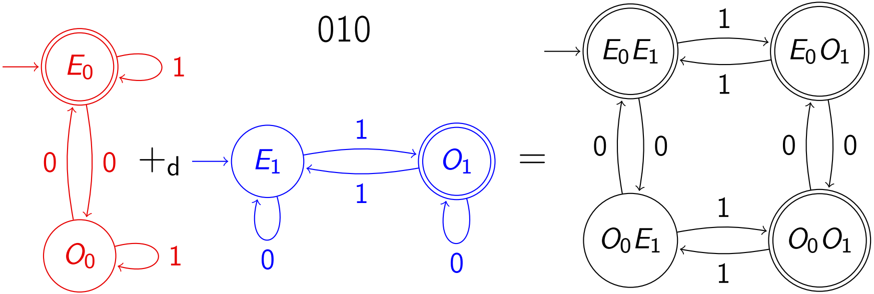
Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



Another way of summing

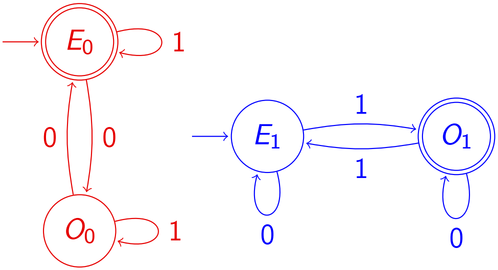
Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



Wouldn't it be nice if instead of building the product, we just ran the components independently ...

Another way of summing

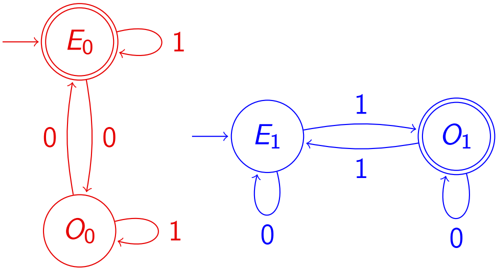
Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



Wouldn't it be nice if instead of building the product, we just ran the components independently . . . as we just did!

Another way of summing

Recall the sum construction from last week: it was the product automaton except with states accepting if *either* component is accepting:



Wouldn't it be nice if instead of building the product, we just ran the components independently . . . as we just did!

But this isn't a DFA – so what is it?

Non-determinism

A non-deterministic finite automaton (NFA) may have:

- any number of start states
- any number of transitions for each letter from each state

Non-determinism

A non-deterministic finite automaton (NFA) may have:

- any number of start states
- any number of transitions for each letter from each state

Formally: An NFA comprises:

- A finite set Q of states
- A finite alphabet Σ of input letters
- A transition relation $\delta \subseteq Q \times \Sigma \times Q$
- A set of starting states $S \subseteq Q$
- A subset $F \subseteq Q$ of accepting (or final) states

Non-determinism

A non-deterministic finite automaton (NFA) may have:

- any number of start states
- any number of transitions for each letter from each state

Formally: An NFA comprises:

- A finite set Q of states
- A finite alphabet Σ of input letters
- A transition relation $\delta \subseteq Q \times \Sigma \times Q$
- A set of starting states $S \subseteq Q$
- A subset $F \subseteq Q$ of accepting (or final) states

Note that we no longer need the black hole convention: we just omit unwanted transitions

The book chooses to define DFA like this with the added constraints 'S is a singleton' and 'δ is functional'. It's a matter of taste.

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).
- at end of input, the NFA accepts if *any* active state is accepting

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).
- at end of input, the NFA accepts if *any* active state is accepting

That's what we saw when we evolved the sum construction in terms of the two components.

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).
- at end of input, the NFA accepts if *any* active state is accepting

That's what we saw when we evolved the sum construction in terms of the two components.

How much memory do we need to run an NFA like this?

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).
- at end of input, the NFA accepts if *any* active state is accepting

That's what we saw when we evolved the sum construction in terms of the two components.

How much memory do we need to run an NFA like this?
One bit for each state. With a DFA we need $\lg n$ bits to track the single current state.

NFA behaviour – parallel understanding

An NFA may have many active/current states:

- all the start states are initially active
- whenever q is active, if input a occurs, then *all* $q' : q \xrightarrow{a} q'$ become active instead of q (hence if there is no such q', the 'activity token' on q dies).
- at end of input, the NFA accepts if *any* active state is accepting

That's what we saw when we evolved the sum construction in terms of the two components.

How much memory do we need to run an NFA like this?
One bit for each state. With a DFA we need $\lg n$ bits to track the single current state. So running an NFA requires exponentially more memory than a DFA.

NFA behaviour – non-deterministic understanding(?)

- At the beginning, the machine guesses/chooses a start state out of S. Then it behaves like a DFA, except that:

NFA behaviour – non-deterministic understanding(?)

- At the beginning, the machine guesses/chooses a start state out of S. Then it behaves like a DFA, except that:
- If at state q and input a there is more than one transition $q \xrightarrow{a}$, then the machine guesses/chooses which one to follow; if there is none, the machine dies (rejects regardless of the rest of the input).

NFA behaviour – non-deterministic understanding(?)

- At the beginning, the machine guesses/chooses a start state out of S. Then it behaves like a DFA, except that:
- If at state q and input a there is more than one transition $q \xrightarrow{a}$, then the machine guesses/chooses which one to follow; if there is none, the machine dies (rejects regardless of the rest of the input).
- If *some sequence of guesses* for a given input string leads to an accepting state, the string is accepted.

This notion of guess/choice is theoretical: it is not physically realizable. In particular, it is *not* probabilistic or chance choice, and it is *not* quantum anything.

NFA behaviour – non-deterministic understanding(?)

- At the beginning, the machine guesses/chooses a start state out of S. Then it behaves like a DFA, except that:
- If at state q and input a there is more than one transition $q \xrightarrow{a}$, then the machine guesses/chooses which one to follow; if there is none, the machine dies (rejects regardless of the rest of the input).
- If *some sequence of guesses* for a given input string leads to an accepting state, the string is accepted.

This notion of guess/choice is theoretical: it is not physically realizable. In particular, it is *not* probabilistic or chance choice, and it is *not* quantum anything.

Another way to think is: if a magic oracle tells you which way to go at each choice, strings in the language are accepted. This way of thinking makes more sense at higher levels of complexity than FSMs.

Using NFAs

Why do we want to use NFAs?

Using NFAs

Why do we want to use NFAs?

Some problems are (much) easier to build NFAs for than DFAs for.

Using NFAs

Why do we want to use NFAs?

Some problems are (much) easier to build NFAs for than DFAs for.

Consider L the language of strings over $\{a, b\}$ that end in ab.

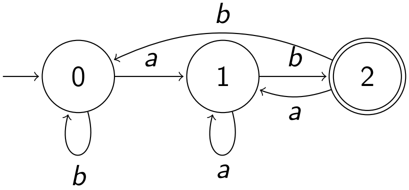
Using NFAs

Why do we want to use NFAs?

Some problems are (much) easier to build NFAs for than DFAs for.

Consider L the language of strings over $\{a, b\}$ that end in ab.

Here is a DFA for L:



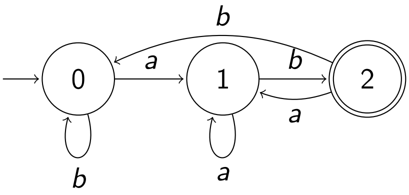
Using NFAs

Why do we want to use NFAs?

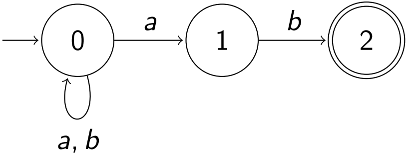
Some problems are (much) easier to build NFAs for than DFAs for.

Consider L the language of strings over $\{a, b\}$ that end in ab.

Here is a DFA for L:



And here is an NFA:



More formalism

We extend some notations and terms to talk about NFAs:

- If $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, then $\hat{\delta}\colon \wp(Q) \times \Sigma \to \wp(Q)$ is the state set transition function defined by $\hat{\delta}(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \{q' : \delta(q, a, q')\}$, and

Here again I am differing slightly from the book. Examine the two sets of definitions carefully. *De gustibus non est disputandum!*

More formalism

We extend some notations and terms to talk about NFAs:

- If $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, then $\hat{\delta}\colon \wp(Q) \times \Sigma \to \wp(Q)$ is the state set transition function defined by $\hat{\delta}(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \{q' : \delta(q, a, q')\}$, and
- $\hat{\delta}^*\colon \wp(Q) \times \Sigma^* \to \wp(Q)$ is the string transition function defined by $\hat{\delta}^*(\hat{Q}, \varepsilon) = \hat{Q}$ and $\hat{\delta}^*(\hat{Q}, xs) = \hat{\delta}^*(\hat{\delta}(\hat{Q}, x), s)$

Here again I am differing slightly from the book. Examine the two sets of definitions carefully. *De gustibus non est disputandum!*

More formalism

We extend some notations and terms to talk about NFAs:

- If $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, then $\hat{\delta} \colon \wp(Q) \times \Sigma \to \wp(Q)$ is the state set transition function defined by $\hat{\delta}(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \{q' : \delta(q, a, q')\}$, and
- $\hat{\delta}^* \colon \wp(Q) \times \Sigma^* \to \wp(Q)$ is the string transition function defined by $\hat{\delta}^*(\hat{Q}, \varepsilon) = \hat{Q}$ and $\hat{\delta}^*(\hat{Q}, xs) = \hat{\delta}^*(\hat{\delta}(\hat{Q}, x), s)$
- If $\Sigma^* \ni s = a_1 \dots a_n$, the trace of s is the sequence $Q_0 \dots Q_n$ where $Q_0 = S$ and $Q_{i+1} = \delta^*(Q_i, a_i)$

Here again I am differing slightly from the book. Examine the two sets of definitions carefully. *De gustibus non est disputandum!*

More formalism

We extend some notations and terms to talk about NFAs:

- If $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, then $\hat{\delta}\colon \wp(Q) \times \Sigma \to \wp(Q)$ is the state set transition function defined by $\hat{\delta}(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \{q' : \delta(q, a, q')\}$, and
- $\hat{\delta}^*\colon \wp(Q) \times \Sigma^* \to \wp(Q)$ is the string transition function defined by $\hat{\delta}^*(\hat{Q}, \varepsilon) = \hat{Q}$ and $\hat{\delta}^*(\hat{Q}, xs) = \hat{\delta}^*(\hat{\delta}(\hat{Q}, x), s)$
- If $\Sigma^* \ni s = a_1 \ldots a_n$, the trace of s is the sequence $Q_0 \ldots Q_n$ where $Q_0 = S$ and $Q_{i+1} = \delta^*(Q_i, a_i)$
- The language accepted by M is $L(M) = \{s \in \Sigma^* : \hat{\delta}^*(S, s) \cap F \neq \varnothing\}$.

Here again I am differing slightly from the book. Examine the two sets of definitions carefully. *De gustibus non est disputandum!*

More formalism

We extend some notations and terms to talk about NFAs:

- If $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, then $\hat{\delta} : \wp(Q) \times \Sigma \to \wp(Q)$ is the state set transition function defined by $\hat{\delta}(\hat{Q}, a) = \bigcup_{q \in \hat{Q}} \{q' : \delta(q, a, q')\}$, and
- $\hat{\delta}^* : \wp(Q) \times \Sigma^* \to \wp(Q)$ is the string transition function defined by $\hat{\delta}^*(\hat{Q}, \varepsilon) = \hat{Q}$ and $\hat{\delta}^*(\hat{Q}, xs) = \hat{\delta}^*(\hat{\delta}(\hat{Q}, x), s)$
- If $\Sigma^* \ni s = a_1 \ldots a_n$, the trace of s is the sequence $Q_0 \ldots Q_n$ where $Q_0 = S$ and $Q_{i+1} = \delta^*(Q_i, a_i)$
- The language accepted by M is $L(M) = \{s \in \Sigma^* : \hat{\delta}^*(S, s) \cap F \neq \varnothing\}$.
- **Theorem**: A language $L \subseteq \Sigma^*$ is regular iff there is some NFA M over Σ that accepts L.

Here again I am differing slightly from the book. Examine the two sets of definitions carefully. *De gustibus non est disputandum!*

NFAs = DFAs

I just said:

Theorem: A language $L \subseteq \Sigma^*$ is regular iff there is some NFA M over Σ that accepts L.

NFAs = DFAs

I just said:

Theorem: A language $L \subseteq \Sigma^*$ is regular iff there is some NFA M over Σ that accepts L.

So NFAs don't give us anything more than DFAs. How can this be?

NFAs = DFAs

I just said:

Theorem: A language $L \subseteq \Sigma^*$ is regular iff there is some NFA M over Σ that accepts L.

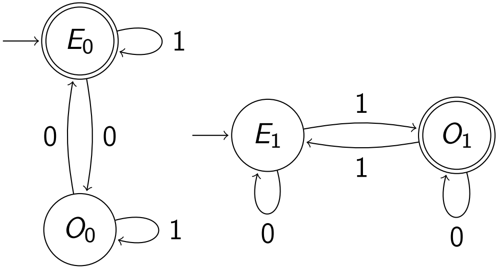
So NFAs don't give us anything more than DFAs. How can this be?

Theorem: For any NFA, we can build a DFA that accepts the same language.

And it's easy – in fact, we've already seen how it's done.

The subset construction

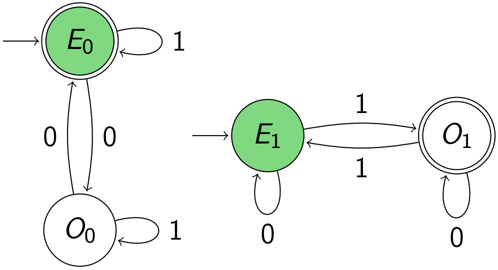
Here's an NFA with two start states:



We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

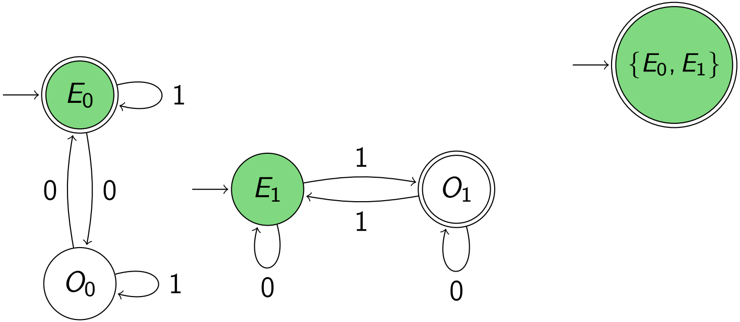
Here's an NFA with two start states:



We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

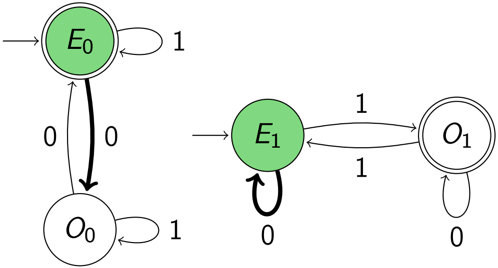
Here's an NFA with two start states:

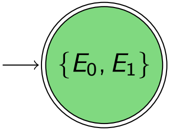


We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

Here's an NFA with two start states:

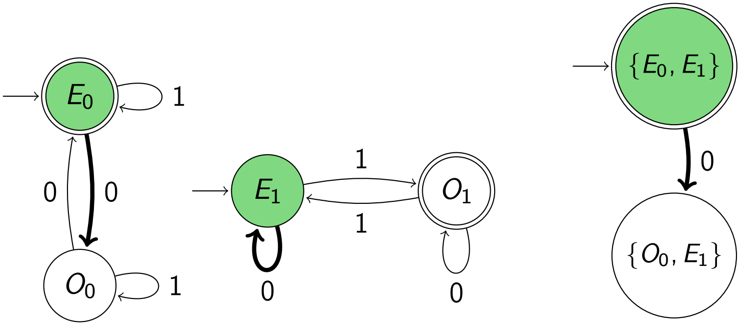




We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

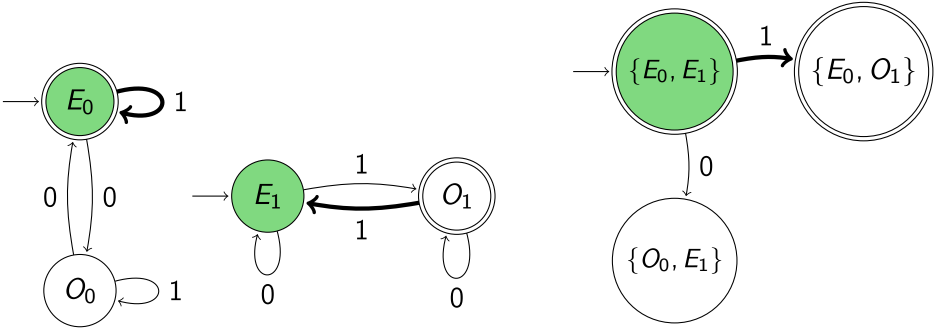
Here's an NFA with two start states:



We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

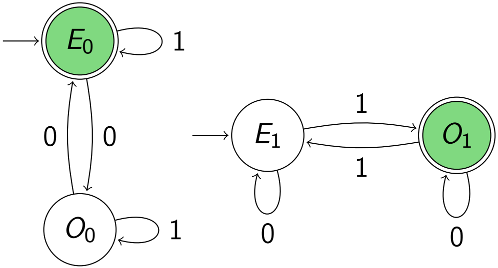
Here's an NFA with two start states:

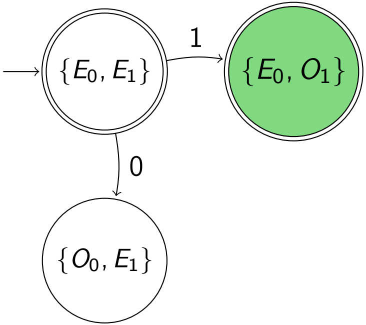


We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

Here's an NFA with two start states:

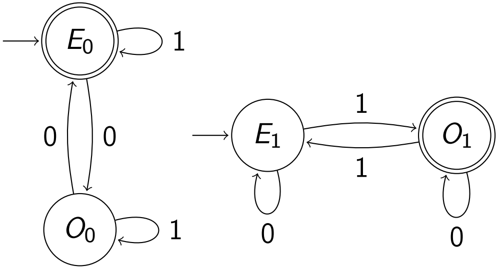


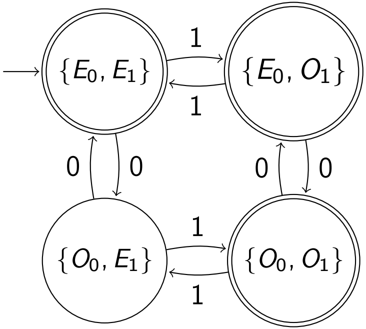


We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

The subset construction

Here's an NFA with two start states:

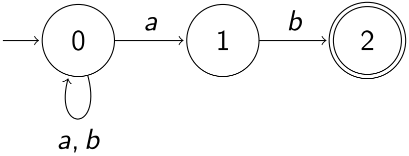




We'll build a DFA just by tracking which NFA states are active and how that changes by transitions in the 'parallel' run.

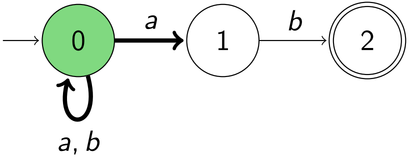
The subset construction

Another example from earlier, with non-det transitions:



The subset construction

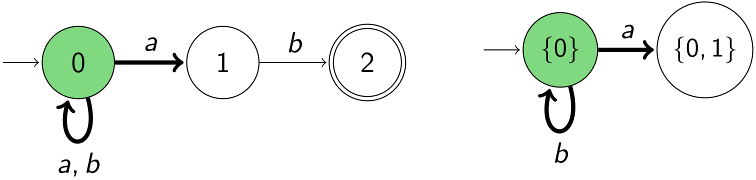
Another example from earlier, with non-det transitions:



If a happens, 0 stays active *and* activates 1. If b happens, 0 just stays active.

The subset construction

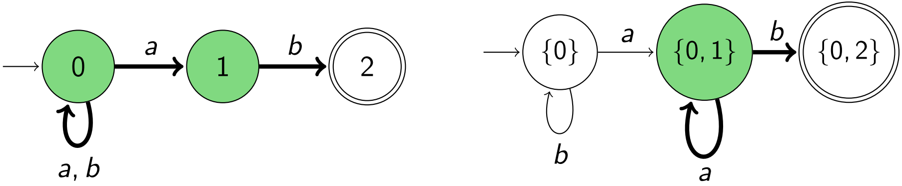
Another example from earlier, with non-det transitions:



If *a* happens, 0 stays active *and* activates 1. If *b* happens, 0 just stays active.

The subset construction

Another example from earlier, with non-det transitions:



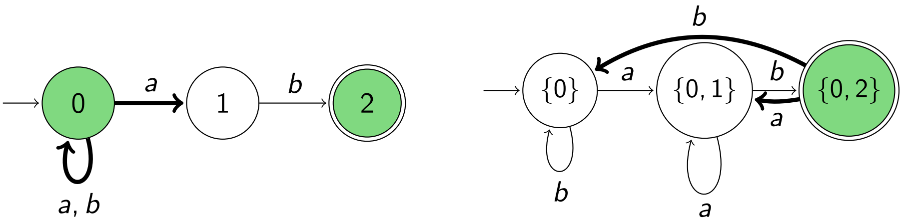
If *a* happens:

- There's no *a*-transition from 1, so 1 dies.
- But 0 stays active *and* (re-)activates 1.

If *b* happens, 0 stays active and the activity on 1 moves to 2.

The subset construction

Another example from earlier, with non-det transitions:



If *a* happens:

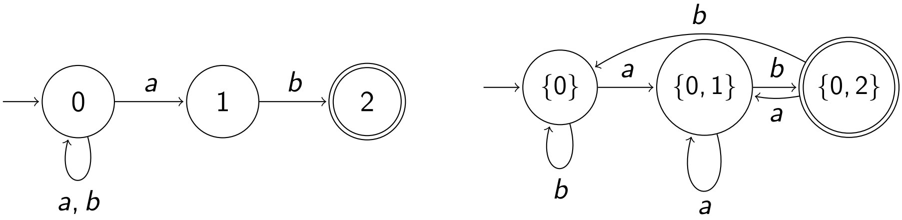
- There's no *a*-transition from 2, so it dies.
- 0 stays active *and* activates 1.

If *b* happens:

- There's no *b*-transition from 2, so it dies.
- 0 stays active.

The subset construction

Another example from earlier, with non-det transitions:



We have reconstructed the original DFA from slide 6! This is a happy coincidence.

The subset construction, formally

What we've seen is a dynamic or *on-the-fly* construction of a DFA.
If we do it mathematically, at one fell swoop, it looks like this:
Given NFA $M = (Q, \Sigma, \delta, S, F)$, define DFA $\hat{M}$ by:

- $\hat{M} = (\wp(Q), \Sigma, \hat{\delta}, S, \mathscr{F})$ where:

The subset construction, formally

What we've seen is a dynamic or *on-the-fly* construction of a DFA.
If we do it mathematically, at one fell swoop, it looks like this:
Given NFA $M = (Q, \Sigma, \delta, S, F)$, define DFA $\hat{M}$ by:

- $\hat{M} = (\wp(Q), \Sigma, \hat{\delta}, S, \mathscr{F})$ where:
- $\hat{\delta}$ is the state set transition function (slide 7) and

The subset construction, formally

What we've seen is a dynamic or *on-the-fly* construction of a DFA.
If we do it mathematically, at one fell swoop, it looks like this:

Given NFA $M = (Q, \Sigma, \delta, S, F)$, define DFA $\hat{M}$ by:

- $\hat{M} = (\wp(Q), \Sigma, \hat{\delta}, S, \mathscr{F})$ where:
- $\hat{\delta}$ is the state set transition function (slide 7) and
- $\mathscr{F} = \{Q' \subseteq Q : Q' \cap F \neq \varnothing\}$

In many cases, most of the superstates in $\wp(Q)$ can't be reached from the starting superstate S, so on-the-fly construction is almost always the right thing in practice.

Now convince yourself (using the book if necessary) that $L(M) = L(\hat{M})$.

It's always annoyed me that we have to write $A \cap B \neq \varnothing$ to say that A and B overlap. Somebody on reddit suggests $A \between B$. What do you think?

ϵ-NFAs

In practice, it's very useful to have a slightly extended notion of NFA.

An ϵ-NFA is an NFA which has an additional special symbol $\epsilon \notin \Sigma$, and a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

If $q \xrightarrow{\epsilon} q'$, then the machine can move from q to q' *without reading any input*.

This makes it much easier to *concatenate* machines or build loops. (We'll see examples later.)

Everything we've done can be adjusted to ϵ-NFAs with a little work – see the book for details. In particular, the subset construction still works. (**Read** the book section on this, p. 331–3 on the draft pdf.)

The Greek letter lower-case epsilon has two common forms: standard ε and *lunate* ϵ. I like to use ε for the empty string, and ϵ for the silent transition, but that's just me ...

Building (ϵ-)NFAs: complement and product

The product construction works on NFAs just as it does on DFAs (with a little work for ϵ).

Building (ϵ-)NFAs: complement and product

The product construction works on NFAs just as it does on DFAs (with a little work for ϵ).

Complement does **not** work: an NFA accepts if *any* run leads to F, so its complement would have to accept only if *all* runs lead to $Q - F$, and that's not an NFA.

Building (ϵ-)NFAs: complement and product

The product construction works on NFAs just as it does on DFAs (with a little work for ϵ).

Complement does **not** work: an NFA accepts if *any* run leads to F, so its complement would have to accept only if *all* runs lead to $Q - F$, and that's not an NFA.

To complement an NFA, first convert to DFA and then complement: exponential blow-up in states.

Building (ϵ-)NFAs: sum

In compensation, the sum or union becomes much easier:

- Let $M = (Q, \Sigma, \delta, S, F)$, and $M' = (Q', \Sigma, \delta', S', F')$, *where* $Q \cap Q' = \varnothing$.
- The sum is $M + M' = (Q \cup Q', \Sigma, \delta \cup \delta', S \cup S', F \cup F')$.

In other words, just put the two automata side by side, as we did at the beginning of this week.

Some people write $M + M'$, some $M \cup M'$.

Building ϵ-NFAs: concatenation

The big win from ϵ is concatenation:

Given $L = L(M)$ and $L' = L(M')$, can we build a machine that accepts $LL' = \{ss' : s \in L, s' \in L'\}$?

Hence we know regular languages are closed under concatenation.

Building ϵ-NFAs: concatenation

The big win from ϵ is concatenation:

Given $L = L(M)$ and $L' = L(M')$, can we build a machine that accepts $LL' = \{ss' : s \in L, s' \in L'\}$?

- The concatenation of M and M' (where $\Sigma' = \Sigma$ and $Q \cap Q' = \varnothing$) is $MM' = (Q \cup Q', \Sigma, \delta \cup \delta' \cup \delta'', S, F')$, where $\delta'' = \{(q, \epsilon, q') : q \in F, q' \in S'\}$.

Hence we know regular languages are closed under concatenation.

Building ϵ-NFAs: concatenation

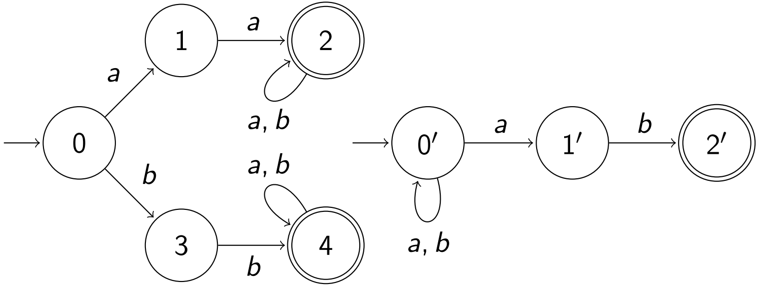
The big win from ϵ is concatenation:

Given $L = L(M)$ and $L' = L(M')$, can we build a machine that accepts $LL' = \{ss' : s \in L, s' \in L'\}$?

- The concatenation of M and M' (where $\Sigma' = \Sigma$ and $Q \cap Q' = \varnothing$) is $MM' = (Q \cup Q', \Sigma, \delta \cup \delta' \cup \delta'', S, F')$, where $\delta'' = \{(q, \epsilon, q') : q \in F, q' \in S'\}$.

In other words, put M and M' side by side, and connect the end of M to the start of M' with ϵ transitions.

Hence we know regular languages are closed under concatenation.



Building ϵ-NFAs: concatenation

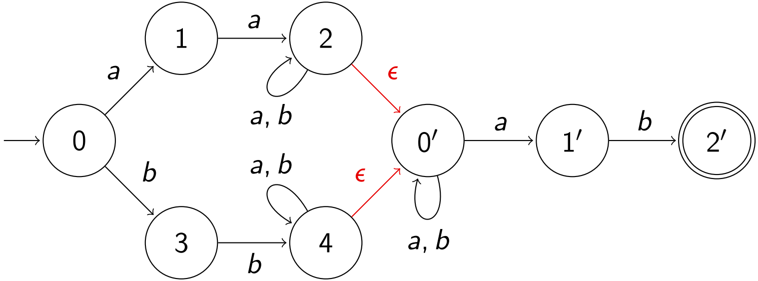
The big win from ϵ is concatenation:

Given $L = L(M)$ and $L' = L(M')$, can we build a machine that accepts $LL' = \{ss' : s \in L, s' \in L'\}$?

- The concatenation of M and M' (where $\Sigma' = \Sigma$ and $Q \cap Q' = \varnothing$) is $MM' = (Q \cup Q', \Sigma, \delta \cup \delta' \cup \delta'', S, F')$, where $\delta'' = \{(q, \epsilon, q') : q \in F, q' \in S'\}$.

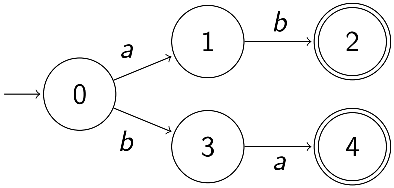
In other words, put M and M' side by side, and connect the end of M to the start of M' with ϵ transitions.

Hence we know regular languages are closed under concatenation.



Building ϵ-NFAs: looping

A generalization of concatenation is concatenating a machine *with itself*:

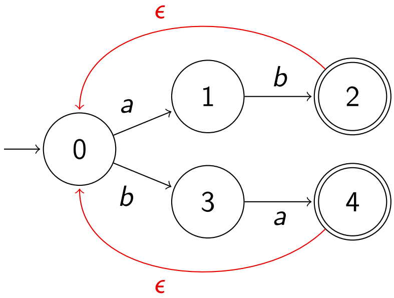


Note that the initial and final states remain such.

This machine accepts $\{ab, ba\}$.

Building ϵ-NFAs: looping

A generalization of concatenation is concatenating a machine *with itself*:



Note that the initial and final states remain such.

This machine accepts strings made up of sequences of *ab* and *ba*.

Regular Expressions

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

'regexp' vs 'regex' is one of those religious wars.

Regular Expressions

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

Given an alphabet Σ, we define the class $\mathscr{R}$ of regular expressions over Σ, and the languages $L(R)$ accepted by them, thus:

- if $a \in \Sigma$, then $a \in \mathscr{R}$, and $L(a) = \{a\}$

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

Given an alphabet Σ, we define the class $\mathscr{R}$ of regular expressions over Σ, and the languages $L(R)$ accepted by them, thus:

- if $a \in \Sigma$, then $a \in \mathscr{R}$, and $L(a) = \{a\}$
- $\varepsilon \in \mathscr{R}$, and $L(\varepsilon) = \{\varepsilon\}$

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

Regular Expressions

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

Given an alphabet Σ, we define the class $\mathscr{R}$ of regular expressions over Σ, and the languages $L(R)$ accepted by them, thus:

- if $a \in \Sigma$, then $a \in \mathscr{R}$, and $L(a) = \{a\}$
- $\varepsilon \in \mathscr{R}$, and $L(\varepsilon) = \{\varepsilon\}$
- if $R, S \in \mathscr{R}$, then $RS \in \mathscr{R}$ and $L(RS) = L(R)L(S)$

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

Regular Expressions

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

Given an alphabet Σ, we define the class $\mathscr{R}$ of regular expressions over Σ, and the languages $L(R)$ accepted by them, thus:

- if $a \in \Sigma$, then $a \in \mathscr{R}$, and $L(a) = \{a\}$
- $\varepsilon \in \mathscr{R}$, and $L(\varepsilon) = \{\varepsilon\}$
- if $R, S \in \mathscr{R}$, then $RS \in \mathscr{R}$ and $L(RS) = L(R)L(S)$
- if $R, S \in \mathscr{R}$, then $R|S \in \mathscr{R}$ and $L(R|S) = L(R) \cup L(S)$

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

Regular Expressions

ϵ-NFAs are not very convenient for writing in programs!

Regular expressions (regexes, regexps) are a simple and universally used way of describing string languages. Any program that does anything with text probably uses them.

There are two styles of notation used for regexps: traditional CS theory, and programming language. We'll use programming.

Given an alphabet Σ, we define the class $\mathscr{R}$ of regular expressions over Σ, and the languages $L(R)$ accepted by them, thus:

- if $a \in \Sigma$, then $a \in \mathscr{R}$, and $L(a) = \{a\}$
- $\varepsilon \in \mathscr{R}$, and $L(\varepsilon) = \{\varepsilon\}$
- if $R, S \in \mathscr{R}$, then $RS \in \mathscr{R}$ and $L(RS) = L(R)L(S)$
- if $R, S \in \mathscr{R}$, then $R|S \in \mathscr{R}$ and $L(R|S) = L(R) \cup L(S)$
- if $R \in \mathscr{R}$, then $R^* \in \mathscr{R}$, and $L(R^*) = L(R)^*$

'regexp' vs 'regex' is one of those religious wars.

The main difference is $\cup$ vs $|$, and the plentiful *syntactic sugar* in PL notation.

If $L \subseteq \Sigma^*$, L^* means $\bigcup_{n>=0} L^n$, where $L^0 = \{\varepsilon\}$.

Regexp examples

- $(ab|ba)^*$ is the language from slide 16.2.
- $(aa|bb)(a|b)^*ab$ is the language from slide 15.4
- $1^*(1^*01^*01^*)^*$ is the language of strings with an even number of 0s. (Why? Why is this so much more complex than the DFA from last week?)

Regular expressions to ϵ-NFAs

The constructors for regexps are exactly the operators with easy ϵ-NFA constructions. So it is very easy to convert regexps to ϵ-NFAs, starting with the automata for ε and a.

So easy it's not even worth having a slide!

We can conclude that for $R \in \mathscr{R}$, $L(R)$ is a regular language.

If you need to see details, they're in the book.

Regular expressions to ϵ-NFAs

The constructors for regexps are exactly the operators with easy ϵ-NFA constructions. So it is very easy to convert regexps to ϵ-NFAs, starting with the automata for ε and a.

So easy it's not even worth having a slide!

We can conclude that for $R \in \mathscr{R}$, $L(R)$ is a regular language.

If you use the ϵ-NFA constructors, what is the automaton for ab?

If you need to see details, they're in the book.

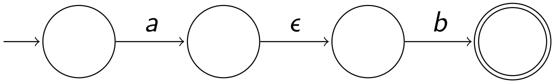
Regular expressions to ϵ-NFAs

The constructors for regexps are exactly the operators with easy ϵ-NFA constructions. So it is very easy to convert regexps to ϵ-NFAs, starting with the automata for ε and a.

So easy it's not even worth having a slide!

We can conclude that for $R \in \mathscr{R}$, $L(R)$ is a regular language.

If you use the ϵ-NFA constructors, what is the automaton for ab?



If you need to see details, they're in the book.

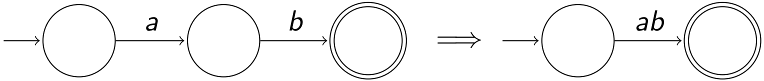
ϵ-NFAs to regular expressions

Unsurprisingly, every regular language is described by a regexp: any ϵ-NFA can be converted to a regexp. This is harder!

ϵ-NFAs to regular expressions

Unsurprisingly, every regular language is described by a regexp: any ϵ-NFA can be converted to a regexp. This is harder!

The basic technique is to remove *internal* states, and combine the transitions through it into transitions labelled by regexps:

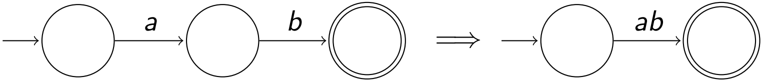


Internal means neither initial nor accepting.

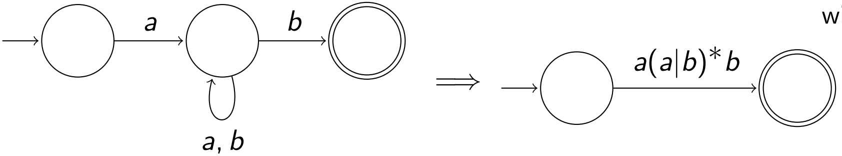
ϵ-NFAs to regular expressions

Unsurprisingly, every regular language is described by a regexp: any ϵ-NFA can be converted to a regexp. This is harder!

The basic technique is to remove *internal* states, and combine the transitions through it into transitions labelled by regexps:



If the state to be removed has self loops, that's still easy:



Internal means neither initial nor accepting.

We've combined multiple (a, b) transitions into one with $|$.

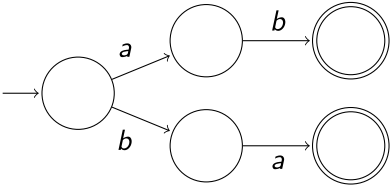
ϵ-NFAs to regular expressions, cont.

So far so good, but what about initial and accepting states with loops etc.?

ϵ-NFAs to regular expressions, cont.

So far so good, but what about initial and accepting states with loops etc.?

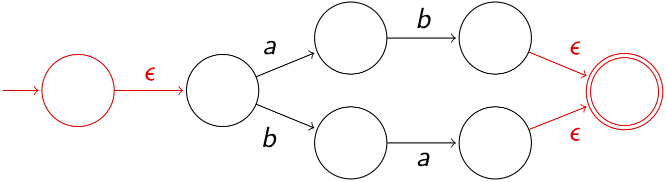
Before doing the previous, first convert to a new machine by adding a single initial and single final:



ϵ-NFAs to regular expressions, cont.

So far so good, but what about initial and accepting states with loops etc.?

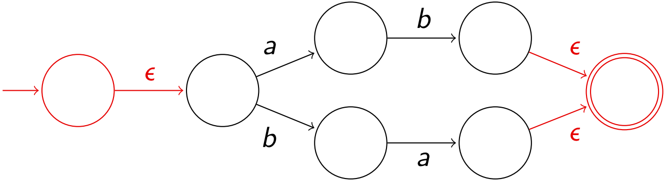
Before doing the previous, first convert to a new machine by adding a single initial and single final:



ϵ-NFAs to regular expressions, cont.

So far so good, but what about initial and accepting states with loops etc.?

Before doing the previous, first convert to a new machine by adding a single initial and single final:



Now repeating the previous procedure will bring you to a big | regexp on one transition between the red states.

Hence: regular expressions describe exactly the regular languages.

The book describes a different method, solving equations and using *Arden's Rule*. It is essentially equivalent, and perhaps easier to program, but less easy to understand intuitively.

If you are feeling strong, try our technique on the 'even 0s and odd 1s' machine from last week.

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a\text{–}f]$ meaning $(a|b|c|d|e|f)$.

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a–f]$ meaning $(a|b|c|d|e|f)$.
- *Negated character classes* $[\hat{}abc]$ meaning any character *except* a, b, c.

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a\text{–}f]$ meaning $(a|b|c|d|e|f)$.
- *Negated character classes* $[\hat{}abc]$ meaning any character *except* a, b, c.
- *Optional subexpressions* $R?$ meaning $(\varepsilon|R)$.

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a\text{–}f]$ meaning $(a|b|c|d|e|f)$.
- *Negated character classes* $[\hat{}abc]$ meaning any character *except* a, b, c.
- *Optional subexpressions* $R?$ meaning $(\varepsilon|R)$.
- *At least one* R^+ meaning RR^*.

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a–f]$ meaning $(a|b|c|d|e|f)$.
- *Negated character classes* $[\hat{}abc]$ meaning any character *except* a, b, c.
- *Optional subexpressions* $R?$ meaning $(\varepsilon|R)$.
- *At least one* R^+ meaning RR^*.
- *Wildchard* . meaning any character (usually except newline).

Regexps in real life

Actual regexps have many more constructors, to make them easier to use. Some examples:

- *Character classes* $[abc]$ meaning $(a|b|c)$, and *ranges* $[a–f]$ meaning $(a|b|c|d|e|f)$.
- *Negated character classes* $[\hat{}abc]$ meaning any character *except* a, b, c.
- *Optional subexpressions* $R?$ meaning $(\varepsilon|R)$.
- *At least one* R^+ meaning RR^*.
- *Wildchard* . meaning any character (usually except newline).

and many more. The older ones are syntactic sugar, but modern languages may add constructors that are no longer regular.

'Syntactic sugar' refers to syntax that doesn't increase the power of a language, but makes it easier and shorter to write.

Matching regexps in real life

Usually, programming languages assume you want to see if R matches any *substring* of input s. I.e. R implicitly means $(.^*R.^*)$.
To avoid this, you can *anchor* to the beginning and/or end of s using $\hat{}R\$$.

The language Perl introduced extremely powerful 'regexps', which have been taken up by other language as 'PCRE's. Perl's regexps are not at all regular. I have an entire talk about them!

Matching regexps in real life

Usually, programming languages assume you want to see if R matches any *substring* of input s. I.e. R implicitly means $(.^*R.^*)$. To avoid this, you can *anchor* to the beginning and/or end of s using $\hat{}R\$$.

In reality, you want to know not only *whether* R matched a substring of s, but *which* substring. You might also want to know which subexpressions of R matched which sub-substrings.

The language Perl introduced extremely powerful 'regexps', which have been taken up by other language as 'PCRE's. Perl's regexps are not at all regular. I have an entire talk about them!

Matching regexps in real life

Usually, programming languages assume you want to see if R matches any *substring* of input s. I.e. R implicitly means $(.^*R.^*)$. To avoid this, you can *anchor* to the beginning and/or end of s using ^R\$.

In reality, you want to know not only *whether* R matched a substring of s, but *which* substring. You might also want to know which subexpressions of R matched which sub-substrings.

So given input $s = aaa$, and $R = $ ^$(a^*)(a^*)$\$, which bits of s are matched by the two parenthesized parts? (Remember NFAs are non-deterministic!)

The language Perl introduced extremely powerful 'regexps', which have been taken up by other language as 'PCRE's. Perl's regexps are not at all regular. I have an entire talk about them!

Matching regexps in real life

Usually, programming languages assume you want to see if R matches any *substring* of input s. I.e. R implicitly means $(.^*R.^*)$. To avoid this, you can *anchor* to the beginning and/or end of s using $\hat{}R\$$.

In reality, you want to know not only *whether* R matched a substring of s, but *which* substring. You might also want to know which subexpressions of R matched which sub-substrings.

So given input $s = aaa$, and $R = \hat{}(a^*)(a^*)\$$, which bits of s are matched by the two parenthesized parts? (Remember NFAs are non-deterministic!)

Programming languages *determinize* regexps: they say that * is *greedy*, i.e. matches as much as possible. So s would be matched as $(aaa)()$.

The language Perl introduced extremely powerful 'regexps', which have been taken up by other language as 'PCRE's. Perl's regexps are not at all regular. I have an entire talk about them!