1 Closure properties of regular languages
   - Operations on languages
   - $\epsilon$-NFAs
   - Closure under concatenation and Kleene star

2 Regular expressions
   - Regular expressions
   - From regular expressions to regular languages

3 Kleene’s theorem and Kleene algebra
   - Kleene’s theorem
   - Kleene algebra
   - From DFAs to regular expressions
A simple application of NFAs

Consider the following little theorem:

If $L_1$ and $L_2$ are regular languages over $\Sigma$, so is $L_1 \cup L_2$.

This can be shown using DFAs . . . but it’s dead easy using NFAs.

Suppose $N_1 = (Q_1, \Delta_1, S_1, F_1)$ is an NFA for $L_1$, and $N_2 = (Q_2, \Delta_2, S_2, F_2)$ is an NFA for $L_2$.

We may assume $Q_1 \cap Q_2 = \emptyset$ (just relabel states if not).

Now consider the NFA

$$(Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, S_1 \cup S_2, F_1 \cup F_2)$$

This is just $N_1$ and $N_2$ ‘side by side’. Clearly, this NFA recognizes precisely $L_1 \cup L_2$.

(Quite useful in practice — no state explosion!)
We’ve seen that if both $L_1$ and $L_2$ are regular languages, so is $L_1 \cup L_2$.

We sometimes express this by saying that regular languages are closed under the ‘union’ operation. (‘Closed’ used here in the sense of ‘self-contained’.)

We will show that regular languages are closed under other operations too: Concatenation: $L_1 \cdot L_2$ and Kleene star: $L^*$

For these, it will be convenient to work with a minor variation on NFAs: $\epsilon$-NFAs.

All this will lead us to another (and very useful!) way of defining regular languages: via regular expressions.
Concatenation and Kleene star

- **Concatenation**: write $L_1 . L_2$ for the language

  $$\{xy \mid x \in L_1, y \in L_2\}$$

  E.g. if $L_1 = \{aaa\}$ and $L_2 = \{b, c\}$ then $L_1 . L_2$ is the language $\{aaab, aaac\}$.

- **Kleene star**: let $L^*$ denote the language

  $$\{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots$$

  E.g. if $L_3 = \{aaa, b\}$ then $L_3^*$ contains strings like $aaaaaa, bbbbb, baaaaaabaaaa$, etc.

  More precisely, $L_3^*$ contains all strings over $\{a, b\}$ in which the letter $a$ always appears in sequences of length some multiple of 3.
Consider the language over the alphabet \( \{a, b, c\} \)

\[ L = \{ x \mid x \text{ starts with } a \text{ and ends with } c \} \]

Which of the following strings is not valid for the language \( L.L \)?

1. \( abcabc \)
2. \( acacac \)
3. \( abcbcac \)
4. \( abcbacbc \)
Consider the (same) language over the alphabet \( \{a, b, c\} \)

\[ L = \{ x \mid x \text{ starts with } a \text{ and ends with } c \} \]

Which of the following strings is not valid for the language \( L^* \)?

1. \( \epsilon \)
2. \( acaca \)
3. \( abcbc \)
4. \( acacacacac \)
We can vary the definition of NFA by also allowing transitions labelled with the special symbol $\epsilon$ (*not* a symbol in $\Sigma$).

The automaton may (but doesn’t have to) perform an $\epsilon$-transition at any time, without reading an input symbol.

This is quite convenient: for instance, we can turn any NFA into an $\epsilon$-NFA with just one start state and one accepting state:

(Add $\epsilon$-transitions from new start state to each state in $S$, and from each state in $F$ to new accepting state.)
Equivalence to ordinary NFAs

Allowing ϵ-transitions is just a convenience: it doesn’t fundamentally change the power of NFAs.

If \( N = (Q, \Delta, S, F) \) is an ϵ-NFA, we can convert \( N \) to an ordinary NFA with the same associated language, by simply ‘expanding’ \( \Delta \) and \( S \) to allow for silent ϵ-transitions.

Formally, the ϵ-closure of a transition relation \( \Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q \) is the smallest relation \( \overline{\Delta} \) that contains \( \Delta \) and satisfies:

- if \( (q, u, q') \in \overline{\Delta} \) and \( (q', \epsilon, q'') \in \Delta \) then \( (q, u, q'') \in \overline{\Delta} \);
- if \( (q, \epsilon, q') \in \Delta \) and \( (q', u, q'') \in \overline{\Delta} \) then \( (q, u, q'') \in \overline{\Delta} \).

Likewise, the ϵ-closure of \( S \) under \( \Delta \) is the smallest set of states \( \overline{S}_\Delta \) that contains \( S \) and satisfies:

- if \( q \in \overline{S}_\Delta \) and \( (q, \epsilon, q') \in \Delta \) then \( q' \in \overline{S}_\Delta \).

We can then replace the ϵ-NFA \( (Q, \Delta, S, F) \) with the ordinary NFA

\[
(Q, \overline{\Delta} \cap (Q \times \Sigma \times Q), \overline{S}_\Delta, F)
\]
Concatenation of regular languages

We can use $\epsilon$-NFAs to show that regular languages are closed under the concatenation operation:

$$L_1 . L_2 = \{ xy \mid x \in L_1, y \in L_2 \}$$

If $L_1, L_2$ are any regular languages, choose $\epsilon$-NFAs $N_1, N_2$ that define them. As noted earlier, we can pick $N_1$ and $N_2$ to have just one start state and one accepting state.

Now hook up $N_1$ and $N_2$ like this:

```
N1  \epsilon
  ^   ^
  |   |
  v   v
N2
```

Clearly, this NFA corresponds to the language $L_1 . L_2$.

To ponder: do we need the $\epsilon$-transition in the middle?
Similarly, we can now show that regular languages are closed under the Kleene star operation:

\[ L^* = \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots \]

For suppose \( L \) is represented by an \( \epsilon \)-NFA \( N \) with one start state and one accepting state. Consider the following \( \epsilon \)-NFA:

![Diagram](image)

Clearly, this \( \epsilon \)-NFA corresponds to the language \( L^* \).
Regular expressions

We’ve been looking at ways of specifying regular languages via machines (often presented as pictures). But it’s also useful to have more textual ways of defining languages.

A regular expression is a written mathematical expression that defines a language over a given alphabet $\Sigma$.

- The basic regular expressions are

  $\emptyset \quad \epsilon \quad a \quad (\text{for } a \in \Sigma)$

- From these, more complicated regular expressions can be built up by (repeatedly) applying the two binary operations $+$, $\cdot$ and the unary operation $^*$. Example: $(a.b + \epsilon)^* + a$

We use brackets to indicate precedence. In the absence of brackets, $^*$ binds more tightly than $\cdot$, which itself binds more tightly than $+$. So $a + b.a^*$ means $a + (b.(a^*))$

Also the dot is often omitted: $ab$ means $a.b$
How do regular expressions define languages?

A regular expression is itself just a written expression (actually in some context-free ‘meta-language’). However, every regular expression $\alpha$ over $\Sigma$ can be seen as defining an actual language $L(\alpha) \subseteq \Sigma^*$ in the following way:

- $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$.
- $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
- $L(\alpha.\beta) = L(\alpha).L(\beta)$
- $L(\alpha^*) = L(\alpha)^*$

Example: $a + ba^*$ defines the language $\{a, b, ba, baa, baaa, \ldots\}$.

The languages defined by $\emptyset, \epsilon, a$ are obviously regular.

What’s more, we’ve seen that regular languages are closed under union, concatenation and Kleene star.

This means every regular expression defines a regular language. (Proof by induction on the size of the regular expression.)
Consider (again) the language

\[ \{ x \in \{0, 1\}^* \mid x \text{ contains an even number of 0's} \} \]

Which of the following regular expressions is \textit{not} a possible definition of this language?

1. \((1^*01^*01^*)^*\)
2. \((1^*01^*0)^*1^*\)
3. \(1^*(01^*0)^*1^*\)
4. \((1 + 01^*0)^*\)
We’ve seen that every regular expression defines a regular language.

The main goal of today’s lecture is to show the converse, that every regular language can be defined by a regular expression. For this purpose, we introduce Kleene algebra: the algebra of regular expressions.

The equivalence between regular languages and expressions is:

**Kleene’s theorem**

*DFAs and regular expressions give rise to exactly the same class of languages (the regular languages).*

As we’ve already seen, NFAs (with or without $\epsilon$-transitions) also give rise to this class of languages.

So the evidence is mounting that the class of regular languages is mathematically a very ‘natural’ class to consider.
Regular expressions give a **textual** way of specifying regular languages. This is useful e.g. for communicating regular languages to a computer.

Another benefit: regular expressions can be manipulated using algebraic laws (**Kleene algebra**). For example:

\[
\begin{align*}
\alpha + (\beta + \gamma) &= (\alpha + \beta) + \gamma \\
\alpha + \emptyset &= \alpha \\
\alpha(\beta \gamma) &= (\alpha \beta) \gamma \\
\alpha(\beta + \gamma) &= \alpha \beta + \alpha \gamma \\
\emptyset \alpha &= \alpha \emptyset = \emptyset \\
\end{align*}
\]

\[
\begin{align*}
\alpha + \beta &= \beta + \alpha \\
\alpha + \alpha &= \alpha \\
\epsilon \alpha &= \alpha \epsilon = \alpha \\
(\alpha + \beta) \gamma &= \alpha \gamma + \beta \gamma \\
\epsilon + \alpha \alpha^* &= \epsilon + \alpha^* \alpha = \alpha^* \\
\end{align*}
\]

Often these can be used to **simplify** regular expressions down to more pleasant ones.
Let’s write $\alpha \leq \beta$ to mean $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$ (or equivalently $\alpha + \beta = \beta$). Then

$$\alpha \gamma + \beta \leq \gamma \implies \alpha^* \beta \leq \gamma$$

$$\beta + \gamma \alpha \leq \gamma \implies \beta \alpha^* \leq \gamma$$

**Arden’s rule:** Given an equation of the form $X = \alpha X + \beta$, its smallest solution is $X = \alpha^* \beta$.

What’s more, if $\epsilon \not\in \mathcal{L}(\alpha)$, this is the *only* solution.

**Intriguing fact:** The rules on this slide and the last form a complete set of reasoning principles, in the sense that if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, then ‘$\alpha = \beta$’ is provable using these rules. (Beyond scope of Inf2A.)
DFAs to regular expressions

We use an example to show how to convert a DFA to an equivalent regular expression.

For each state $r$, let the variable $X_r$ stand for the set of strings that take us from $r$ to an accepting state. Then we can write some simultaneous equations:

\[
X_p = 1X_p + 0X_q + \epsilon \\
X_q = 1X_q + 0X_p
\]
We solve the equations by eliminating one variable at a time:

\[ X_q = 1^*0X_p \quad \text{by Arden’s rule} \]

So \[ X_p = 1X_p + 01^*0X_p + \epsilon \]
\[ = (1 + 01^*0)X_p + \epsilon \]

So \[ X_p = (1 + 01^*0)^* \quad \text{by Arden’s rule} \]

Since the start state is \( p \), the resulting regular expression for \( X_p \) is the one we are seeking. Thus the language recognised by the automaton is:

\[(1 + 01^*0)^*\]

The method we have illustrated here, in fact, works for arbitrary NFAs (without \( \epsilon \)-transitions).
Theory of regular languages: overview

- DFAs
- subset construction
- NFAs
- solve equations
- reg exps
- closure properties
- remove $\epsilon$-transitions
- $\epsilon$-NFAs
Relevant reading:

- Regular expressions: Kozen chapters 7,8; J & M chapter 2.1. (Both texts actually discuss more general ‘patterns’ — see next lecture.)
- From regular expressions to NFAs: Kozen chapter 8; J & M chapter 2.3.
- Kleene algebra: Kozen chapter 9.
- From NFAs to regular expressions: Kozen chapter 9.

Next time: Some applications of all this theory.

- Pattern matching
- Lexical analysis
Appendix: (non-examinable) proof of Kleene’s theorem

Given an NFA $N = (Q, \Delta, S, F)$ (without $\epsilon$-transitions), we’ll show how to define a regular expression defining the same language as $N$.

In fact, to build this up, we’ll construct a three-dimensional array of regular expressions $\alpha_{uv}^X$: one for every $u \in Q$, $v \in Q$, $X \subseteq Q$.

Informally, $\alpha_{uv}^X$ will define the set of strings that get us from $u$ to $v$ allowing only intermediate states in $X$.

We shall build suitable regular expressions $\alpha_{u,v}^X$ by working our way from smaller to larger sets $X$.

Eventually, the language defined by $N$ will be given by the sum ($+$) of the languages $\alpha_{sf}^Q$ for all states $s \in S$ and $f \in F$. 
Here’s how the regular expressions $\alpha_{uv}^X$ are built up.

- If $X = \emptyset$, let $a_1, \ldots, a_k$ be all the symbols $a$ such that $(u, a, v) \in \Delta$. Two subcases:
  - If $u \neq v$, take $\alpha_{uv}^\emptyset = a_1 + \cdots + a_k$
  - If $u = v$, take $\alpha_{uv}^\emptyset = (a_1 + \cdots + a_k) + \epsilon$

  **Convention:** if $k = 0$, take ‘$a_1 + \ldots + a_k$’ to mean $\emptyset$.

- If $X \neq \emptyset$, choose any $q \in X$, let $Y = X - \{q\}$, and define

  $$\alpha_{uv}^X = \alpha_{uv}^Y + \alpha_{uv}^Y(\alpha_{qq}^Y)^* \alpha_{qv}^Y$$

Applying these rules repeatedly gives us $\alpha_{u,v}^X$ for every $u, v, X$. 
Let’s revisit our old friend:

Here $p$ is the only start state and the only accepting state.

By the rules on the previous slide:

\[
\alpha_{p,p}^{\{p,q\}} = \alpha_{p,p}^{\{p\}} + \alpha_{p,q}^{\{p\}} (\alpha_{q,q}^{\{p\}})^* \alpha_{q,p}^{\{p\}}
\]

Now by inspection (or by the rules again), we have

\[
\begin{align*}
\alpha_{p,p}^{\{p\}} &= 1^* \\
\alpha_{p,q}^{\{p\}} &= 1 + 01^*0 \\
\alpha_{q,q}^{\{p\}} &= 01^*
\end{align*}
\]

So the required regular expression is

\[
1^* + 1^*0(1 + 01^*0)^*01^* \quad (A \text{ bit messy!})
\]