Regular Expressions and Finite Automata

Lecture 7
24 September 2019
Fun:

Assignment 1 corrections due today!
Assignment 2 out today!

Previously:

Equivalent computational models: DFAs and NFAs
Closure properties of regular languages
Regular expressions

Today:

What do these three topics have to do with one another?
Regular expressions
The *language of a regular expression* is the language described by that regular expression.

Formally:

\[ L(\varepsilon) = \{\varepsilon\} \]
\[ L(\emptyset) = \emptyset \]
\[ L(a) = \{a\} \]
\[ L(R_1 R_2) = L(R_1)L(R_2) \]
\[ L(R_1 \cup R_2) = L(R_1) \cup L(R_2) \]
\[ L(R^*) = L(R)^* \]
\[ L((R)) = L(R) \]
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$
Designing regular expressions

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

Let $L = \{ w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$

$$(a \cup b)^*aa(a \cup b)^*$$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w$ contains $aa$ as a substring$\}$

$(\text{a} \cup \text{b})*\text{aa}(\text{a} \cup \text{b})*$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa\text{ as a substring}\}$

$$(au b)^*aa(au b)^*$$

$\text{bbabbaabab}$

$\text{aaaa}$

$\text{bbbbbbabbbbaabbbbb}$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w$ contains $aa$ as a substring$\}$

$$(a \cup b)^* aa (a \cup b)^*$$

$bbabbaaabab$

$aaaa$

$bbbbbbabbbbaabbbbbb$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w$ contains $aa$ as a substring$\}$

$\Sigma^*aa\Sigma^*$

A convenient shorthand

bbabbbbaabab

aaaa

bbbbbabaabbbbaabbbb
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

*Recall: $|w|$ denotes the length of string $w$.***
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

$\Sigma \Sigma \Sigma \Sigma$

aaaaa
baba
bbbb
baaa
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$

\[ \Sigma^4 \]

aaaa
baba
bbbb
baaa
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

Here are some candidates regular expressions for $L$. Which are correct?

- $\Sigma^* a \Sigma^*$
- $b^* a b^* \cup b^* b^*$
- $b^* (a^* b^* \cup \varepsilon) b^*$
- $b^* a^* b^* b^* \cup b^* (a^* \cup \varepsilon) b^*$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w$ contains at most one $a\}$

$b^*(a \cup \varepsilon)b^*$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* | w$ contains at most one $a\}$

$b^* (a \cup \epsilon) b^*$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w$ contains at most one $a\}$

$b^*(a \cup \varepsilon)b^*$

$bbbbabbb$

$bbbbbb$

$abbb$

$a$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

\[b^*(a \cup \varepsilon)b^*\]

- $b$$b$$b$$a$$b$$b$$b$$b$
- $b$$b$$b$$b$$b$$b$$b$$b$
- $a$$b$$b$$b$$b$$b$
- $a$$b$$b$$b$$b$
- $a$
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

\[ b^* a? b^* \]

- bbbbbabbbb
- bbbbbbb
- abbb
- a
Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$

Another shorthand

b*a?b*
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.
Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

\[ aa^* \]

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
   aa*
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(\cdot aa^*)^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

`aa*(.aa*)*`
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa*(.aa*)*@$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@$$

- mvassar@vassar.edu
- matthew.vassar@vassarbrewery.com
- matt@cs.vassar.edu
A more elaborate design

Let \( \Sigma = \{a, ., @\} \), where \( a \) represents “any letter”.

Let’s make a regular expression for email addresses.

\[
\text{aa}^*(\text{.aa}^*)^*\text{@}\text{aa}^*.\text{aa}^*
\]

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa*(.aa* )*@aa* .aa*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@aa^*.aa^*(.aa^*)*$$

$\text{mvassar@vassar.edu}$
$\text{matthew.vassar@vassarbrewery.com}$
$\text{matt@cs.vassar.edu}$
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(\.aa*)*@aa*\.aa*\(\.aa*\)*
```

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(\text{.aa}*)*@\text{aa}*.\text{aa}*(\text{.aa}*)*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, \., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

You guessed it – another shorthand

$$a^+.(.aa*)*@aa^*.aa^*(.aa*)*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)*@a^+.a^+(.a^+)*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)*@a^+(.a^+)^+$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
For comparison

\[ a^+ (\cdot a^+) \ast @ a^+ (\cdot a^+) ^+ \]
Convenient shorthands

Σ is a shorthand for “any character in Σ”

$R^n$ is a shorthand for $RR \ldots R$ ($n$ times)

$R?$ is shorthand for $(R \cup \varepsilon)$ — that is, zero or one copies of $R$.

$R^+$ is a shorthand for $RR^*$ — that is, one or more copies of $R$. 
Regular expressions in the real world
UNIX regular expressions

From the beginning (of time), UNIX has used regular expressions in many places, including the grep command.

\texttt{grep} = \texttt{global} (search for a) \texttt{regular expression} and \texttt{print}

Many UNIX commands use an extended RE notation, but it still expresses only the regular languages.
UNIX RE notation

\[ [a_1a_2\ldots a_n] \] is shorthand for \( a_1u a_2u \cdots u a_n \).

Ranges are indicated by first-dash-last and brackets, using ASCII character order, e.g.,

- \[ [a-z] \] = any lowercase letter
- \[ [a-zA-Z] \] = any letter

Dot ( . ) = any character (like our shorthand \( \Sigma \))
UNIX RE notation, continued

Since characters like brackets, dashes, and dots have special meaning, if you want to match them, you need to quote with backslash (\).

Union operator is represented with a bar (|)

Includes our + shorthand for “one or more”, e.g.,

\[a-z]+ = one or more lowercase letter
Perl, Python, Emacs, …

Include additional extensions, notably character classes like \b for word boundary characters, \w for word characters, etc.

With each implementation of regular expressions, they become less standard, so what you write for one language or application won’t work in another.
grep

`grep` lets you search files for text

```
$ grep bananas foo.txt
```

Here are some of my favourite `grep` command line arguments!

- `-i` case insensitive
- `-E` aka `egrep`
- `-v` invert match: find all lines that don't match
- `-l` only show the filenames of the files that matched
- `-o` only print the matching part of the line (not the whole line)
- `-a` search binaries: treat binary data like it's text instead of ignoring it!
- `-A` show context for your search.
  `$ grep -A 3 foo` will show 3 lines of context after a match
- `-F` don't treat the match string as a regex
  eg `$ grep -F ...`
- `-r` recursive! Search all the files in a directory.
- `-c` grep alternatives
  `ack`, `ag`, `ripgrep`
  (better for searching code!)

https://twitter.com/b0rk/status/991880504805871616
Lexical analysis

The first thing a compiler does is break a program into tokens, which are substrings that together represent a unit, e.g.,

- identifiers
- reserved words like “if”
- meaningful single characters like “;” or “+”
- multi-character operators like “<=“
Lexical analysis, continued

There are tools like `lex` or `flex` that let you write a regular expression for each kind of token.

E.g., in UNIX notation, identifiers are something like `[A–Za–z][A–Za–z0–9_]*`

Each RE has an associated action like returning a code for the token found or adding it to a symbol table.
Kleene’s theorem
Kleene’s Theorem: Regular expressions and finite automata have equivalent descriptive power.

That is, the languages recognized by DFAs and NFAs and those described by REs are exactly the regular languages.
NFA, NFA-ε, Regular Expression

DFA

((a ∪ ba*) ∪ ca*) *  \( ab^*(cub)^* \)
Proof

We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton
Constructing an NFA-ε from a regular expression
Cover the six cases in the formal (recursive) definition of REs.

Base cases:

1. \( R = a \)

2. \( R = \varepsilon \)

3. \( R = \emptyset \)
The class of regular languages is closed under the union operation.

For languages represented by $R_1$ and $R_2$, take their NFAs $N_1$ and $N_2$ and combine them into one new NFA $N$.

$N$ must accept input if either $N_1$ or $N_2$ accepts input.

The new machine guesses non-deterministically which of the two machines accepts the input.
The new machine guesses non-deterministically where to split the input in order to have a first part accepted by $N_1$ and a second part accepted by $N_2$.

The class of regular languages is closed under the concatenation operation.

For languages represented by $R_1$ and $R_2$, take their NFAs $N_1$ and $N_2$ and combine them sequentially into one new NFA $N$. 

$R = (R_1 \circ R_2)$
The class of regular languages is closed under the star operation.

For a language represented by $R_1$, modify $N_1$ to accept $(R_1)^*$. The new machine has the option of jumping back to the start state to read another piece that $N_1$ accepts.

Why not just make the start state of $N_1$ a final state?
The construction we’re using in this proof is, approximately, *Thompson’s Algorithm*, which is used in practice by many regular expression matchers to convert regular expressions into NFAs (and, from there, DFAs).

The “Thompson” is Ken Thompson, co-inventor of UNIX.

Since the 1970s, `grep` has converted each regular expression into a finite automaton that it runs to do the search.
Example

(\text{abua})^\ast
Example

\((ab \cup a)^*\)

That ends the first part of the proof of Theorem 1.54, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

**Example 1.56**

We convert the regular expression \((ab \cup a)^*\) to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram.

Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

**Solution from Sipser**
Proof

We will prove this set of equivalences by

- ✔ Showing how to construct a DFA from an NFA-\(\varepsilon\). (Already done!)
- ✔ Showing how to construct an NFA-\(\varepsilon\) from a regular expression
- ✔ Showing how to construct a regular expression from a finite automaton
Constructing a regular expression from a DFA
Generalizing NFAs
Generalizing NFAs
Generalizing NFAs

These are all regular expressions!
Generalizing NFAs

Note: NFAs aren’t allowed to have transitions like these. This is just a thought experiment.
Generalizing NFAs

\[ q_0 \xrightarrow{ab \cup b} q_1 \]

\[ q_0 \xrightarrow{a} q_2 \]

\[ q_2 \xrightarrow{a^*b?a^*} q_3 \]

\[ q_1 \xrightarrow{ab^*} q_3 \]
Generalizing NFAs
Generalizing NFAs
Generalizing NFAs
Generalizing NFAs

\( q_0 \) \( \xrightarrow{ab \cup b} q_1 \)
\( q_2 \) \( \xrightarrow{a} q_2 \) \( \xrightarrow{ab^*} q_3 \)
\( q_2 \) \( \xrightarrow{a^*b?a^*} q_3 \)

Start symbol: \( q_0 \)
Generalizing NFAs

\begin{align*}
q_0 & \xrightarrow{ab \cup b} q_1 \\
q_0 & \xrightarrow{a} q_2 \\
q_2 & \xrightarrow{a\ast b\ast?a\ast} q_3 \\
q_3 & \xrightarrow{ab\ast} q_2
\end{align*}
Generalizing NFAs

\[ q_0 \rightarrow q_1 \text{ via } ab \cup b \]

\[ q_2 \rightarrow q_3 \text{ via } a^*b?a^* \]

\[ a^*b?b \]

\[ start \rightarrow q_0 \]

\[ q_2 \rightarrow q_3 \text{ via } ab^* \]
Generalizing NFAs

```
a a a b a a b b b b
```
Generalizing NFAs

\[ \text{start} \rightarrow q_0 \rightarrow q_1 \]
Generalizing NFAs
Note: We haven’t said that DFAs or NFAs can have regular expressions on arcs in their transition diagrams; they can’t.

As an intermediate model in this construction, we’re using a generalized nondeterministic finite automaton (GNFA).

A GNFA is an NFA where the transition arrows can have a regular expression as the label.

It can read blocks of symbols from the input rather than just one symbol at a time.
**Key idea 1:** Imagine that we can label transitions in an NFA with arbitrary regular expressions.
Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Key idea 2: If we can convert an NFA into a generalized NFA that looks like this,

\[ \text{start} \rightarrow q_0 \xrightarrow{\text{some-regex}} q_1 \]

then we can easily read off a regular expression for the original NFA.
From GNFAs to regular expressions

$R_{00}$, $R_{01}$, $R_{11}$, and $R_{10}$ are arbitrary regular expressions.
From GNFAs to regular expressions

Can we get a clean regular expression from this NFA?
From GNFAs to regular expressions

Key idea 3: Transform a GNFA so it looks like this:
From GNFAs to regular expressions

**First add new start and accept states**

```
start

q_s

q_0

q_1

q_f

R_{00}

R_{01}

R_{10}

R_{11}
```
First add new start and accept states
From GNFAs to regular expressions

First add new start and accept states
Then, for each of the original states, ask could we eliminate this state from the GNFA?
From GNFAs to regular expressions
From GNFAs to regular expressions

We can use concatenation and Kleene closure to skip this state
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions

We can use union to combine these transitions.
From GNFA\textsc{\textbf{s}} to regular expressions

could we eliminate this state from the GNFA?
could we eliminate this state from the GNFA?
From GNFA's to regular expressions

\[ q_s \xrightarrow{R_{00} \cdot R_{12}} q_1 \xrightarrow{\varepsilon} q_f \]

\[ R_{11} \cup R_{10}R_{00} \cdot R_{01} \]
What should we put on this transition?
From GNFAs to regular expressions

\[ R_{00}R_{12} (R_{11} \cup R_{10}R_{00}R_{01})^* \varepsilon \]
From GNFAs to regular expressions

\( R_{00}^* R_{12} \ (R_{11} \cup R_{10} R_{00}^* R_{01})^* \varepsilon \)
From GNFAs to regular expressions

\[ R_{00}^* R_{12} \ (R_{11} \cup R_{10}R_{00}^*R_{01})^* \varepsilon \]
From GNFAs to regular expressions

\[ R_{00}R_{12} (R_{11} \cup R_{10}R_{00}R_{01})^* \]
From GNFAs to regular expressions

\[ R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \]

Diagram:
- Start state: \( q_s \)
- Final state: \( q_f \)
- Transitions:
  - From \( q_s \) to \( q_f \):
    - \( R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \)
  - From \( q_0 \) to \( q_1 \):
    - \( R_{00} \)
    - \( R_{01} \)
  - From \( q_1 \) to \( q_0 \):
    - \( R_{10} \)
  - From \( q_0 \) to \( q_1 \):
    - \( R_{11} \)
The construction at a glance

1. Start with a DFA $M$ for the language $L$, which we’ll use as a GNFA.

2. Add a new start state $q_s$ and accept state $q_f$ to $M$
   - Add an $\varepsilon$-transition from $q_s$ to the old start state of $M$.
   - Add $\varepsilon$-transitions from each accepting state of $M$ to $q_f$, then mark them as not accepting.

3. Repeatedly remove states other than $q_s$ and $q_f$ from the NFA by “shortcutting” them until only $q_s$ and $q_f$ remain.

4. The transition from $q_s$ to $q_f$ is now a regular expression equivalent to the original DFA.
Eliminating a state

Before

\[ R_{\text{other}} \]

\[ R_{\text{in}} \]

\[ R_{\text{out}} \]

\[ q_i \rightarrow q_j \]

\[ q_{\text{rip}} \]

\[ R_{\text{stay}} \]

After

\[(R_{\text{in}})(R_{\text{stay}})^*(R_{\text{out}}) \cup (R_{\text{other}})\]

\[ q_i \rightarrow q_j \]
Eliminating a state

To eliminate a state $q_{\text{rip}}$ from the automaton, do the following for each pair of states $q_i$ and $q_j$, where there’s a transition from $q_0$ into $q_{\text{rip}}$ and a transition from $q_{\text{rip}}$ into $q_j$:

Let $R_{\text{in}}$ be the regex. on the transition from $q_i$ to $q_{\text{rip}}$.

Let $R_{\text{out}}$ be the regex. on the transition from $q_{\text{rip}}$ to $q_j$.

If there is a regular expression $R_{\text{stay}}$ on a transition from $q_{\text{rip}}$ to itself,

Add a new transition from $q_i$ to $q_j$ labeled $((R_{\text{in}})(R_{\text{stay}})^*(R_{\text{out}}))$.

Otherwise,

Add a new transition from $q_i$ to $q_j$ labeled $((R_{\text{in}})(R_{\text{out}}))$.

If a pair of states has multiple transitions between them labeled $R_1$, $R_2$, $\ldots$, $R_k$, replace them with a single transition labeled $R_1 \cup R_2 \cup \cdots \cup R_k$. See diagram on previous slide.
Example 1

\[ 
\begin{array}{c}
\text{1} \\
\hspace{2cm}
\text{2}
\end{array} 
\]

\[ 
\begin{array}{c}
a \\
\text{a, b}
\end{array} 
\]

\[ 
\begin{array}{c}
b \\
\text{a, b}
\end{array} 
\]
Example 1

Add new start and end state

$S \in A \in \varepsilon$
Example 1

Add new start and end state

Remove state 2
Example 1

- Add new start and end state
- Remove state 2
Example 1

Add new start and end state

Remove state 2
Example 1

Add new start and end state

Remove state 2
Example 1

Add new start and end state

Remove state 2

Remove state 1
Example 1

- Add new start and end state
- Remove state 2
- Remove state 1

\[ S \in A \]

\[ b(a \cup b)^* \]
Example 1

1. Add new start and end state:

2. Remove state 2:

3. Remove state 1:
Example 1

Add new start and end state

Remove state 2

Remove state 1
Example 2

Original FA
Example 2

Original FA

Add new start and end state
Example 2

Original FA

Add new start and end state
Example 2

Original FA

Add new start and end state

q1

q2

q3

q4

q1

q2

q3

q4

q5

q1

q2

q3

q4

q5

a

b

a

b

a

b

a

b

a

b

a

b

a

b

a

b

ε

ε

ε
Example 2

Original FA

Add new start and end state

Eliminate q1
Example 2

Original FA

Add new start and end state

Eliminate $q_1$
Example 2

Original FA

Add new start and end state

Eliminate q₁
Example 2

Original FA

Add new start and end state

Eliminate q_1

Eliminate q_2
Example 2

Original FA

Add new start and end state

Eliminate q₂

Eliminate q₁
Example 2

Original FA

Add new start and end state

Eliminate q₂

Eliminate q₁
Example 2

Original FA

Add new start and end state

Eliminate q_2

Eliminate q_1

Eliminate q_3
Example 2

Original FA

Add new start and end state

Eliminate q₂

Eliminate q₃

Eliminate q₁

(a*b ∪ a*ba*b)(a ∪ ba*b)*
Example 3
Step 1: Modify to create a unique start and end state:
Step 2: Eliminate state $q_0$. 
Step 3: Eliminate state $q_1$. 
Step 4: Eliminate state $q_2$. 

$S \xrightarrow{\text{start}} \quad a^*bb^* \cup (a^*bb^*a ((b \cup aa^*b)b^*)^*(((b \cup aa^*b)b^* \cup \epsilon))) \xrightarrow{\epsilon} F$
Proof

We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton
The regular languages are recognized by NFA-\(\varepsilon\)s, NFAs, DFAs, and REs (…and GNFAs)

\[
\text{Regular Expression}
\]

\[
((a \cup b)^* \cup (c^* a^*) \cup (a \cup b a^* c)^*)^* \cup (a b^* (c^* a^* b)^*)^*
\]

- **Thompson’s algorithm**
- **Subset construction**
- **State elimination**

NFA, NFA-\(\varepsilon\), DFA
The following are all equivalent:

$L$ is a regular language.

There is a DFA $D$ such that $L(D) = L$.

There is an NFA $N$ such that $L(N) = L$.

There is a regular expression $R$ such that $L(R) = L$. 
Why this matters

The equivalence of regular expressions and finite automata has practical relevance.

Tools like grep and flex that use regular expressions capture all the power available via DFAs and NFAs.

This also is hugely theoretically significant: The regular languages can be assembled “from scratch” using a small number of operations!
Acknowledgments

This lecture incorporates material from:

Nancy Ide
Keith Schwarz
Michael Sipser
Jeffrey Ullman