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:

\[ L(\varepsilon) = \{ \varepsilon \} \]

\[ L(\emptyset) = \emptyset \]

\[ L(a) = \{ a \} \]

\[ L(R_1R_2) = L(R_1)L(R_2) \]

\[ L(R_1 \cup R_2) = L(R_1) \cup L(R_2) \]

\[ L(R^*) = L(R)^* \]

\[ L((R)) = L(R) \]
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 $L = \{w \in \Sigma^* | w \text{ contains } aa \text{ as a substring}\}$

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

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

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

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

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

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

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

$bbabbbbaabab$

$aaaa$

$bbbbbbabbbbbbaabbbbbb$
Designing regular expressions

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

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

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

$bbabbbbaabab$

$aaaa$

$bbbbbbabbbbaabbbbb$
Designing regular expressions

Let \( \Sigma = \{a, b\} \)

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

\[ \Sigma^*aa\Sigma^* \]

\( \text{A convenient shorthand} \)

bbabbbbaaabab
aaaa
bbbbbabbbaabbbbbbb
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 $L = \{w \in \Sigma^* \mid |w| = 4\}$

$\Sigma \Sigma \Sigma \Sigma$

aaaa
baba
bbbb
baaa
Designing regular expressions

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

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

$\Sigma \Sigma \Sigma \Sigma$

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

Another shorthand
Designing regular expressions

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

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

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

- $\Sigma*a\Sigma*$
- $b*ab*ub*b*$
- $b*(a\cup\varepsilon)b*$
- $b*a*bub*b$
- $b*(a\cup\varepsilon)b*$
Designing regular expressions

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

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

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

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

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

$b^*(a \cup \epsilon) b^*$
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 $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$  

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

$bbbbabbb$

$bbbbbbbb$

$abbb$

$a$
Designing regular expressions

Let \( \Sigma = \{a, b\} \)

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

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

\[ bbbbbabbb \]

\[ bbbbbbb \]

\[ abbb \]

\[ a \]
Designing regular expressions

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

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

Another shorthand

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

\[
\begin{align*}
\text{bbbbabbb} \\
\text{bbbbbb} \\
\text{abbb} \\
\text{a}
\end{align*}
\]
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.

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.

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

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.

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

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.

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

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.

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

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.

\[
\text{aa}^* (\text{aa}^*)^* @ \text{aa}^* . \text{aa}^*(\text{aa}^*)^* \\
mvassar@vassar.edu \\
matthew.vassar@vassarbrewery.com \\
matt@cs.vassar.edu
\]
A more elaborate design

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

Let’s make a regular expression for email addresses.

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

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.

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

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.

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

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

$\Sigma$ is a shorthand for “any character in $\Sigma$”

$R^n$ 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.

`grep` = global (search for a) regular expression and 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 \(a_1 \cup a_2 \cup \cdots \cup a_n\).

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

- \(\text{[a–z]}\) = any lowercase letter
- \(\text{[a–zA–Z]}\) = any letter

Dot (\(\cdot\)) = 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 lets you search files for text

$ grep bananas foo.txt

Here are some of my favourite grep command line arguments!

-E aka egrep

Use if you want regexps like ".+" to work. Otherwise you need to use "\." 

-V

Invert match: find all lines that don't match

-l

Only show the filenames of the files that matched

-i case insensitive

Only print the matching part of the line (not the whole line)

-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 ...

-a

Search binaries: treat binary data like it's text instead of ignoring it!

 grep alternatives

ack, eg, ripgrep (better for searching code!)

Julia Evans @b0rk
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.
\[
\text{NFA, NFA-}\varepsilon, \quad \text{Regular Expression } \quad ab^* (c^* b^*)^* \quad \text{DFA}
\]
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. $R = a$
   - Start state
   - Transition labeled $a$

2. $R = \epsilon$
   - Start state

3. $R = \emptyset$
   - Start state
4 \( R = (R_1 \cup R_2) \)

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 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 $N$. 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 $N_2$. 

$R = (R_1 \circ R_2)$
6 \( R = (R_1)^* \)

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

$$(abua)^*$$
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
Generalizing NFAs

\begin{align*}
& \text{start} \\
q_0 & \xrightarrow{ab \cup b} q_1 \\
q_2 & \xrightarrow{a} q_2 \xrightarrow{a*b?a*} q_3 \xrightarrow{ab*} q_3
\end{align*}

\text{Sample input: } a a a b a a b b b b
Generalizing NFAs

\[ q_0 \xrightarrow{\text{start}} q_1 \]

\[ q_0 \xrightarrow{a} q_2 \]

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

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

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

Input: \[ a\ a\ a\ a\ b\ a\ a\ b\ b\ b\ b\ b\ ]
Generalizing NFAs

- start
- \( q_0 \) transitions to
- \( q_1 \):
  - \( ab \cup b \)
- \( q_2 \):
  - \( a \)
  - \( a*b?a* \)
- \( q_3 \):
  - \( ab* \)

Input sequence:
- \( a \ a \ a \ b \ a \ a \ b \ b \ b \ b \)
Generalizing NFAs

\[ \begin{align*}
q_0 & \xrightarrow{ab \cup b} q_1 \\
q_2 & \xrightarrow{ab^*} q_3 \\
q_0 & \xrightarrow{a} q_2 \\
q_2 & \xrightarrow{a^*b?a^*} q_3 \\
start & \rightarrow q_0
\end{align*} \]
Generalizing NFAs

\[ q_0 \xrightarrow{ab, a} q_2 \xrightarrow{ab^*} q_3 \]

\[ q_0 \xrightarrow{b} q_1 \]

\[ \text{start} \]

Input sequence: a a a b a a b b b b
Generalizing NFAs
Generalizing NFAs

Diagram:

- Start state $q_0$ transitions to $q_1$ on $ab$ and $b$.
- $q_0$ transitions to $q_2$ on $a$.
- $q_2$ transitions to $q_3$ on $a^*b?a^*$.
- $q_3$ transitions back to $q_2$ on $ab^*$.

Input sequence: $a$ $a$ $a$ $b$ $a$ $a$ $b$ $b$ $b$ $b$
Generalizing NFAs

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

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

\[ q_2 \xrightarrow{ab^*} q_3 \]

Input sequence: \[ a a a b a a b b b b \]
Generalizing NFAs

Diagram:

- Start state: $q_0$
- Transitions:
  - $q_0 \xrightarrow{ab \cup b} q_1$
  - $q_0 \xrightarrow{a} q_2$
  - $q_2 \xrightarrow{a^*b?a^*} q_3$
  - $q_2 \xrightarrow{ab^*} q_3$

Input: $a^4 a b a b b b$
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?
Is there a simple regular expression for the language of this generalized NFA?
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,

![Diagram](image)

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

\[ R_{00}, R_{01}, R_{11}, \text{ and } R_{10} \text{ are arbitrary regular expressions.} \]
From GNFA s 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 GNFA to regular expressions

First add new start and accept states

![Diagram showing state transitions](image)
From GNFAs to regular expressions

First add new start and accept states
From GNFAs to regular expressions

First add new start and accept states
From GNFAs to regular expressions

Then, for each of the original states, ask

could we eliminate this state from the GNFA?
From GNFAs to regular expressions
We can use concatenation and Kleene closure to skip this state
From GNFA to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions

\[ R_{00}^* R_{12} \]

\[ R_{11} \]

\[ R_{10} R_{00}^* R_{01} \]

\[ \varepsilon \]
From GNFAs to regular expressions
We can use union to combine these transitions
From GNFA to regular expressions

could we eliminate this state from the GNFA?

\[ R_{00}^* R_{12} \]

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

could we eliminate this state from the GNFA?

\[
\begin{align*}
R_{00} & \cdot R_{12} \\
\varepsilon \\
R_{11} & \cup R_{10} R_{00} \cdot R_{01}
\end{align*}
\]
From GNFAs to regular expressions

\[ R_{00}^*R_{12} \]

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

What should we put on this transition?
From GNFAs to regular expressions

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

\[ R_{00}^*R_{12} \ (R_{11} \cup R_{10}R_{00}^*R_{01})^* \epsilon \]
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

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

After

\[ q_i \xrightarrow{R_{\text{in}}} q_{\text{rip}} \xrightarrow{R_{\text{stay}}} q_j \]

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

\[ q_i \xrightarrow{\left( R_{\text{in}} \right) \left( R_{\text{stay}} \right)^* \left( R_{\text{out}} \right) \cup \left( R_{\text{other}} \right)} q_j \]
Eliminating a state

To eliminate a state $q_{rip}$ 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 $q_{rip}$ and a transition from $q_{rip}$ into $q_j$:

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

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

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

Otherwise,

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

If a pair of states has multiple transitions between them labeled $R_1$, $R_2$, ..., $R_k$, replace them with a single transition labeled $R_1 \cup R_2 \cup \cdots \cup R_k$. 

See diagram on previous slide
Example 1
Example 1

Add new start and end state
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**

- **Initial State:** 1
  - Transition on 'a': 1 to 2
  - Transition on 'b': 1 to 2

- **Final State:** 2
  - Transition on 'a': 1 to 2
  - Transition on 'b': 2 to 1

- **Add new start and end state**
  - New state: S
  - Transition on 'a': S to A
  - Transition on 'b': S to A

- **Remove state 2**
  - Transition on 'ε': S to A
  - Transition on 'b(au b)*': A to S

---

- **Modified Diagram:**
  - Initial state: S
  - Final state: A
  - Transition on 'a': S to A
  - Transition on 'b': S to A
  - Transition on 'ε': S to A
  - Transition on 'b(au b)*': A to S
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
Example 1

Add new start and end state

Remove state 1

Remove state 2
Example 1

1. Add new start and end state.
2. Remove state 1.
3. Remove state 2.
Example 2

Original FA

\[
\begin{array}{c}
q_1 \\
\downarrow b \\
q_3 \\
\downarrow b \\
q_2 \\
\downarrow a \\
\end{array}
\]

\[
\begin{array}{c}
a \\
a \\
b \\
b \\
b \\
\end{array}
\]
Example 2

Original FA

Add new start and end state
Example 2

Original FA

Add new start and end state

$q_1 \xrightarrow{a} q_3 \xrightarrow{b} q_2 \xrightarrow{b} q_1\quad q_3 \xrightarrow{a} q_4 \xrightarrow{\varepsilon} q_1 \xrightarrow{b} q_2 $
Example 2

Original FA

Add new start and end state
Example 2

Original FA

Add new start and end state

Eliminate q₁

q₁, q₂, q₃, q₄, q₅

q₁ → q₂ (a, b)
q₂ → q₁ (b)
q₁ → q₃ (a)
q₃ → q₁ (a, b)
q₃ → q₅ (ε)
q₅ → q₄ (ε)
q₄ → q₃ (ε)
q₃ → q₂ (b, a)
q₂ → q₃ (b, a)
q₅ → q₂ (b, a)

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 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 q₂

Eliminate q₁
Example 2

Original FA

Add new start and end state

Eliminate q2

Eliminate q1

Eliminate q3
Example 2

Original FA

Add new start and end state

Eliminate q₂

Eliminate q₁

Eliminate q₃

\[(a^*b \cup a^*ba^*b)(a \cup ba^*b)^*\]
Acknowledgments

This lecture incorporates material from:

- Nancy Ide
- Keith Schwarz
- Michael Sipser
- Jeffrey Ullman