Regular Expressions
Recap from Last Time
Regular Languages

- A language \( L \) is a *regular language* if there is a DFA \( D \) such that \( \mathcal{L}(D) = L \).

*Theorem*: The following are equivalent:

- \( L \) is a regular language.
- There is a DFA for \( L \).
- There is an NFA for \( L \).
Language Concatenation

• If $w \in \Sigma^*$ and $x \in \Sigma^*$, then $wx$ is the **concatenation** of $w$ and $x$.

• If $L_1$ and $L_2$ are languages over $\Sigma$, the **concatenation** of $L_1$ and $L_2$ is the language $L_1L_2$ defined as

$$L_1L_2 = \{ wx | w \in L_1 \text{ and } x \in L_2 \}$$

• Example: if $L_1 = \{ a, ba, bb \}$ and $L_2 = \{ aa, bb \}$, then

$$L_1L_2 = \{ aaa, abb, baaa, babb, bbaa, bbbb \}$$
Lots and Lots of Concatenation

- Consider the language $L = \{ \text{aa}, \text{b} \}$
- $LL$ is the set of strings formed by concatenating pairs of strings in $L$.
  $$\{ \text{aaaa}, \text{aab}, \text{baa}, \text{bb} \}$$
- $LLL$ is the set of strings formed by concatenating triples of strings in $L$.
  $$\{ \text{aaaaaa}, \text{aaaab}, \text{aabaa}, \text{aabb}, \text{baaaa}, \text{baab}, \text{bbaa}, \text{bbb} \}$$
- $LLLL$ is the set of strings formed by concatenating quadruples of strings in $L$.
  $$\{ \text{aaaaaaaa}, \text{aaaaaab}, \text{aaaaabaa}, \text{aaabbb}, \text{aabaaaaa}, \text{aabaaab}, \text{aabbaa}, \text{aabbb}, \text{baaaaaa}, \text{baaaaab}, \text{baabaa}, \text{baabb}, \text{baaaaaa}, \text{baaab}, \text{bbbaa}, \text{bbbaa}, \text{bbbb} \}$$
Language Exponentiation

- We can define what it means to “exponentiate” a language as follows:

- $L^0 = \{\varepsilon\}$
  - Intuition: The only string you can form by gluing no strings together is the empty string.
  - Notice that $\{\varepsilon\} \neq \emptyset$. Can you explain why?

- $L^{n+1} = LL^n$
  - Idea: Concatenating $(n+1)$ strings together works by concatenating $n$ strings, then concatenating one more.

- **Question to ponder**: Why define $L^0 = \{\varepsilon\}$?
- **Question to ponder**: What is $\emptyset^0$?
The Kleene Closure

• An important operation on languages is the **Kleene Closure**, which is defined as

\[ L^* = \{ w \in \Sigma^* \mid \exists n \in \mathbb{N}. w \in L^n \} \]

• Mathematically:

\[ w \in L^* \iff \exists n \in \mathbb{N}. w \in L^n \]

• Intuitively, all possible ways of concatenating zero or more strings in \( L \) together, possibly with repetition.

• **Question:** What is \( \emptyset^0 \)?
The Kleene Closure

If $L = \{ a, bb \}$, then $L^* = \{ \epsilon, a, bb, aa, abb, bba, bbb, aaaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb, \ldots \}$

Think of $L^*$ as the set of strings you can make if you have a collection of stamps – one for each string in $L$ – and you form every possible string that can be made from those stamps.
Closure Properties

• **Theorem:** If \( L_1 \) and \( L_2 \) are regular languages over an alphabet \( \Sigma \), then so are the following languages:
  
  • \( \overline{L_1} \)
  • \( L_1 \cup L_2 \)
  • \( L_1 \cap L_2 \)
  • \( L_1L_2 \)
  • \( L_1^* \)

• These properties are called **closure properties of the regular languages.**
New Stuff!
Another View of Regular Languages
Rethinking Regular Languages

• We currently have several tools for showing a language $L$ is regular:
  • Construct a DFA for $L$.
  • Construct an NFA for $L$.
  • Combine several simpler regular languages together via closure properties to form $L$.

• We have not spoken much of this last idea.
Constructing Regular Languages

• **Idea:** Build up all regular languages as follows:
  
  • Start with a small set of simple languages we already know to be regular.
  
  • Using closure properties, combine these simple languages together to form more elaborate languages.

• *This is a bottom-up approach to the regular languages.*
Constructing Regular Languages

- **Idea:** Build up all regular languages as follows:
  - Start with a small set of simple languages we already know to be regular.
  - Using closure properties, combine these simple languages to form more elaborate languages.
- This is a bottom-up approach to regular languages.
Regular Expressions

- **Regular expressions** are a way of describing a language via a string representation.
- They’re used just about everywhere:
  - They’re built into the JavaScript language and used for data validation.
  - They’re used in the UNIX `grep` and `flex` tools to search files and build compilers.
  - They’re employed to clean and scrape data for large-scale analysis projects.
- Conceptually, regular expressions are strings describing how to assemble a larger language out of smaller pieces.
Atomic Regular Expressions

- The regular expressions begin with three simple building blocks.
- The symbol $\emptyset$ is a regular expression that represents the empty language $\emptyset$.
- For any $a \in \Sigma$, the symbol $a$ is a regular expression for the language $\{a\}$.
- The symbol $\varepsilon$ is a regular expression that represents the language $\{\varepsilon\}$.
  - *Remember:* $\{\varepsilon\} \neq \emptyset!$
  - *Remember:* $\{\varepsilon\} \neq \varepsilon!$
Compound Regular Expressions

• If $R_1$ and $R_2$ are regular expressions, $R_1 R_2$ is a regular expression for the concatenation of the languages of $R_1$ and $R_2$.

• If $R_1$ and $R_2$ are regular expressions, $R_1 \cup R_2$ is a regular expression for the union of the languages of $R_1$ and $R_2$.

• If $R$ is a regular expression, $R^*$ is a regular expression for the Kleene closure of the language of $R$.

• If $R$ is a regular expression, $(R)$ is a regular expression with the same meaning as $R$. 
Operator Precedence

• Here’s the operator precedence for regular expressions:

\[
\begin{align*}
& (R) \\
& R^* \\
& R_1 R_2 \\
& R_1 \cup R_2 \\
\end{align*}
\]

• So \(ab^*c \cup d\) is parsed as \(((a(b^*))c) \cup d\)
Regular Expression Examples

• The regular expression $\text{trick} \cup \text{treat}$ represents the language

$$\{ \text{trick, treat} \}.$$

• The regular expression $\text{booo}^*$ represents the regular language

$$\{ \text{boo, booo, boooo, ...} \}.$$

• The regular expression $\text{candy!}(\text{candy!})^*$ represents the regular language

$$\{ \text{candy!, candy!candy!, candy!candy!candy!, ...} \}.$$
Regular Expressions, Formally

• The **language of a regular expression** is the language described by that regular expression.

• Formally:
  
  - $\mathcal{L}(\epsilon) = \{\epsilon\}$
  - $\mathcal{L}(\emptyset) = \emptyset$
  - $\mathcal{L}(a) = \{a\}$
  - $\mathcal{L}(R_1R_2) = \mathcal{L}(R_1) \mathcal{L}(R_2)$
  - $\mathcal{L}(R_1 \cup R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$
  - $\mathcal{L}(R^*) = \mathcal{L}(R)^*$
  - $\mathcal{L}(\langle R \rangle) = \mathcal{L}(R)$

**Worthwhile activity:** Apply this recursive definition to $a(b\cup c)((d))$ and see what you get.
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 $L = \{ w \in \Sigma^* \mid w$ contains $aa$ 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 $\}$.

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

bbabbbbaabab
aaaa
bbbbbbbbabbbabbbbabbbbbbb
Designing Regular Expressions

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

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

$bbabbbbaaabab$

$aaaa$

$bbbbbabbabbaabbabbb$
Designing Regular Expressions

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

$\Sigma^*aa\Sigma^*$

$bbabbbbaaabab$

$aaaa$

$bbbbbbabbbbaabbbbbb$
Designing Regular Expressions

- Let $\Sigma = \{a, b\}$.
- Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$. 
Designing Regular Expressions

Let $\Sigma = \{a, b\}$.
Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$. 

The length of a string $w$ is denoted $|w|$. 
Designing Regular Expressions

• Let $\Sigma = \{ a, b \}$.
• Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$. 
Designing Regular Expressions

• Let $\Sigma = \{a, b\}$.
• Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$. 

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

$\Sigma \Sigma \Sigma \Sigma$

a
a
a
a

b
b
b
b

b
b
b
b

b
a
a
a
Designing Regular Expressions

• Let $\Sigma = \{a, b\}$.
• Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$. 
Designing Regular Expressions

- Let $\Sigma = \{a, b\}$.
- Let $L = \{ w \in \Sigma^* \mid |w| = 4 \}$.

$\Sigma^4$

aaaa
baba
bbbb
baaa
Designing Regular Expressions

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

Here are some candidate regular expressions for the language $L$. Which of these are correct?

$\Sigma*a\Sigma*$
$b*ab* \cup b*$
$b*(a \cup \varepsilon)b*$
$b*a*b* \cup b*$
$b*(a* \cup \varepsilon)b*$
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 $L = \{ w \in \Sigma^* | w$ contains at most one $a \}$. 

\[ b^*(a \cup \epsilon)b^* \]
Designing Regular Expressions

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

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

- bbbbbabbb
- bbbbb
- abbb
- a
Designing Regular Expressions

• Let $\Sigma = \{a, b\}$.

• Let $L = \{ w \in \Sigma^* | w$ 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 $a \}$. 

$b^*a?b^*$

$bbbbaabbb$
$bbbbbbbb$
$abbb$
$a$
A More Elaborate Design

- Let $\Sigma = \{ a, ., @ \}$, where $a$ represents “some letter.”
- Let's make a regex for email addresses.
A More Elaborate Design

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

• Let's make a regex for email addresses.

  cs103@cs.stanford.edu
  first.middle.last@mail.site.org
  dot.at@dot.com
A More Elaborate Design

- Let $\Sigma = \{ a, ., @ \}$, where $a$ represents “some letter.”
- Let's make a regex for email addresses.

- `cs103@cs.stanford.edu`
- `first.middle.last@mail.site.org`
- `dot.at@dot.com`
A More Elaborate Design

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

• Let's make a regex for email addresses.

  \[
  aa^*
  \]

  cs103@cs.stanford.edu
  first.middle.last@mail.site.org
  dot.at@dot.com
A More Elaborate Design

- Let $\Sigma = \{a, ., @\}$, where $a$ represents “some letter.”
- Let's make a regex for email addresses.

$$aa^*$$

- cs103@cs.stanford.edu
- first.middle.last@mail.site.org
- dot.at@dot.com
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

\[
\text{cs103@cs.stanford.edu}
\]
\[
\text{first.middle.last@mail.site.org}
\]
\[
\text{dot.at@dot.com}
\]
A More Elaborate Design

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

• Let's make a regex for email addresses.

  \[ aa^* (.aa^*)^* \]

  \begin{align*}
  \text{cs103@cs.stanford.edu} \\
  \text{first.middle.last@mail.site.org} \\
  \text{dot.at@dot.com}
  \end{align*}
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

```plaintext
cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
```
A More Elaborate Design

• Let $\Sigma = \{a, ., @\}$, where $a$ represents "some letter."

• Let's make a regex for email addresses.

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

```
cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
```
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

- cs103@cs.stanford.edu
- first.middle.last@mail.site.org
- dot.at@dot.com
A More Elaborate Design

- Let $\Sigma = \{ a, ., @ \}$, where $a$ represents “some letter.”
- Let's make a regex for email addresses.

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

```plaintext
cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
```
A More Elaborate Design

- Let $\Sigma = \{ a, ., @ \}$, where $a$ represents “some letter.”
- Let's make a regex for email addresses.

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

cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
A More Elaborate Design

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

• Let's make a regex for email addresses.

```
aa* (.aa*)* @ aa* .aa* (.aa*)*
```
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

  cs103@cs.stanford.edu
  first.middle.last@mail.site.org
  dot.at@dot.com
A More Elaborate Design

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

• Let's make a regex for email addresses.

  \[ a^+ (.aa*)*@ aa*.aa* (.aa*)* \]

  \begin{align*}
  &\text{cs103@cs.stanford.edu} \\
  &\text{first.middle.last@mail.site.org} \\
  &\text{dot.at@dot.com}
  \end{align*}
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
A More Elaborate Design

- Let $\Sigma = \{a, ., @\}$, where $a$ represents "some letter."
- Let's make a regex for email addresses.

```
a+ (.a+)* @ a+.a+ (.a+)*
```

- $\text{cs103@cs.stanford.edu}$
- $\text{first.middle.last@mail.site.org}$
- $\text{dot.at@dot.com}$
A More Elaborate Design

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

• Let's make a regex for email addresses.

\[
\begin{align*}
a^+ \ (a^+)* \ @ \ a^+ \ .(a^+)^* \\
\end{align*}
\]

- $\text{cs103@cs.stanford.edu}$
- $\text{first.middle.last@mail.site.org}$
- $\text{dot.at@dot.com}$
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

  cs103@cs.stanford.edu
  first.middle.last@mail.site.org
  dot.at@dot.com
A More Elaborate Design

- Let $\Sigma = \{ \text{a, ., @} \}$, where $\text{a}$ represents “some letter.”
- Let's make a regex for email addresses.

\[
a^+ \ (a^+)* \ @ \ a^+ \ (a^+)^+
\]

cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

cs103@cs.stanford.edu
first.middle.last@mail.site.org
dot.at@dot.com
A More Elaborate Design

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

• Let's make a regex for email addresses.

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

- cs103@cs.stanford.edu
- first.middle.last@mail.site.org
- dot.at@dot.com
For Comparison

\(a^+ (\cdot a^+)^* \cdot a^+ (\cdot a^+)\)
Shorthand Summary

- $R^n$ is shorthand for $RR \ldots R$ ($n$ times).
  - Edge case: define $R^0 = \varepsilon$.
- $\Sigma$ is shorthand for “any character in $\Sigma$.”
- $R?$ is shorthand for $(R \cup \varepsilon)$, meaning “zero or one copies of $R$.”
- $R^+$ is shorthand for $RR^*$, meaning “one or more copies of $R$.”
Time-Out for Announcements!
Problem Set Four Graded

- Your diligent and hardworking TAs have finished grading PS4. Grades and feedback are now available on Gradescope.

- As always, please review your feedback! Knowing where to improve is more important than just seeing a raw score.

- Did we make a mistake? Regrades on Gradescope will open tomorrow and are due in one week.
Problem Set Six

- Problem Set Five was due at 2:30PM today.
- Problem Set Six goes out today. It’s due next Friday at 2:30PM.
  - Design DFAs and NFAs for a range of problems!
  - Explore formal language theory!
  - See some clever applications!
Back to CS103!
The Lay of the Land
Regular Languages

Languages you can build a DFA for.

Languages you can build an NFA for.
Regular Languages

Languages you can build a DFA for.

Languages you can build an NFA for.

Languages You Can Write a Regex For
Languages you can build a DFA for.

Languages you can build an NFA for.

Regular Languages

Languages You Can Write a Regex For
Languages you can build a DFA for.

Languages you can build an NFA for.

Languages you can write a regex for.
Regular Languages

Languages you can build a DFA for.

Languages you can build an NFA for.

Languages you can write a Regex for.
Languages you can build a DFA for.

Languages you can build an NFA for.

Regular Languages

Languages You Can Write a Regex For
The Power of Regular Expressions

*Theorem:* If $R$ is a regular expression, then $\mathcal{L}(R)$ is regular.

*Proof idea:* Use induction!

- The atomic regular expressions all represent regular languages.
- The combination steps represent closure properties.
- So anything you can make from them must be regular!
Thompson’s Algorithm

• In practice, many regex matchers use an algorithm called *Thompson's algorithm* to convert regular expressions into NFAs (and, from there, to DFAs).
  • Read Sipser if you’re curious!

• **Fun fact:** the “Thompson” here is Ken Thompson, one of the co-inventors of Unix!
Languages you can build a DFA for.  

Languages you can build an NFA for.
Languages you can build a DFA for.

Languages you can build an NFA for.

Regular Languages

Languages You Can Write a Regex For
Languages you can build a DFA for.

Languages you can build an NFA for.

**Regular Languages**

Languages You Can Write a Regex For
The Power of Regular Expressions

**Theorem:** If $L$ is a regular language, then there is a regular expression for $L$.

*This is not obvious!*

**Proof idea:** Show how to convert an arbitrary NFA into a regular expression.
Generalizing NFAs

Diagram:
- Start state: $q_0$
- Transitions:
  - $q_0 \xrightarrow{a} q_1$
  - $q_0 \xrightarrow{\varepsilon} q_2$
  - $q_2 \xrightarrow{\Sigma} q_3$
  - $q_3 \xrightarrow{\Sigma} q_4$
  - $q_4 \xrightarrow{b} q_2$
  - $q_1 \xrightarrow{b} q_4$

States:
- $q_0$
- $q_1$
- $q_2$
- $q_3$
- $q_4$
Generalizing NFAs
Generalizing NFAs

These are all regular expressions!
Generalizing NFAs

\[ q_0 \xrightarrow{ab \cup b} q_1 \xrightarrow{ab^*} q_2 \xrightarrow{a*b?a^*} q_3 \]

-start-
Generalizing NFAs

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

-q₀
  ↓
  a
  ↓
q₂
  ↓
a*?b?a*
  ↓
q₃
  ↓
ab*
  ↓
q₁
  ↓
ab  u  b
  ↓
start

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

Diagram:
- Start state: $q_0$
- Transitions:
  - $q_0$ to $q_1$: $ab \cup b$
  - $q_0$ to $q_2$: $ab^*$
  - $q_1$ to $q_2$: $a$
  - $q_2$ to $q_3$: $a^*b?a^*$
  - $q_2$ to $q_3$: $a$
  - $q_3$: Final state

Input: a | a | a | b | a | a | b | b | b
Generalizing NFAs
Generalizing NFAs

\[ q_0 \xrightarrow{ab \cup b} q_1 \]
\[ q_0 \xrightarrow{ab^*} q_2 \]
\[ q_2 \xrightarrow{a*b?a^*} q_3 \]
Generalizing NFAs
Generalizing NFAs

Diagram:

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

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

**Diagram:**

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

**Word:**

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

\[ q_0 \rightarrow \text{start} \rightarrow q_2 \rightarrow a \rightarrow q_2 \rightarrow a*b?a* \rightarrow q_3 \rightarrow ab* \rightarrow q_1 \rightarrow ab \cup b \rightarrow q_0 \]

\[ \text{a a a a b a a b b b b} \]
Generalizing NFAs

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

\[ q_2 \xrightarrow{a} q_1 \xrightarrow{a*b?a*} q_3 \]

Input: a a a b a a b b b
**Key Idea 1:** Imagine that we can label transitions in an NFA with arbitrary regular expressions.
Generalizing NFAs
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

$q_0$ \rightarrow \begin{array}{c} a^+ (a^+)^* @ a^+ (a^+)^+ \\ q_1 \end{array}
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...

...then we can easily read off a regular expression for the original NFA.
From NFAs to Regular Expressions
From NFAs to Regular Expressions

Here, $R_{11}$, $R_{12}$, $R_{21}$, and $R_{22}$ are arbitrary regular expressions.
From NFAs to Regular Expressions

Question: Can we get a clean regular expression from this NFA?
From NFAs to Regular Expressions

Key Idea 3: Somehow transform this NFA so that it looks like

\[ \text{some-regex} \]
The first step is going to be a bit weird...
From NFAs to Regular Expressions

\[ q_s \xrightarrow{\text{start}} q_1 \xrightarrow{R_{12}} q_2 \xrightarrow{R_{21}} q_2 \xrightarrow{R_{22}} q_f \]
From NFAs to Regular Expressions

[Diagram of an NFA with states labeled $q_s$, $q_1$, $q_2$, and $q_f$, transitions marked with $R_{11}$, $R_{12}$, $R_{21}$, and $R_{22}$, and an initial state labeled 'start'.]
From NFAs to Regular Expressions
From NFAs to Regular Expressions
From NFAs to Regular Expressions

Could we eliminate this state from the NFA?
From NFAs to Regular Expressions
From NFAs to Regular Expressions
Note: We’re using concatenation and Kleene closure in order to skip this state.
From NFAs to Regular Expressions

\[ \varepsilon R_{11} * R_{12} \]
From NFAs to Regular Expressions

\[ \varepsilon R_{11} \ast R_{12} \]

```
\begin{align*}
\text{start} & \rightarrow q_s \\
q_s & \rightarrow q_1 \quad \varepsilon \\
q_1 & \rightarrow q_2 \quad R_{11} \\
q_2 & \rightarrow q_f \quad R_{12} \\
q_f & \rightarrow q_f \quad \varepsilon 
\end{align*}
```
From NFAs to Regular Expressions

\[ \varepsilon R_{11} \ast R_{12} \]
From NFAs to Regular Expressions

\[ \varepsilon R_{11} \ast R_{12} \]

\[ R_{21} R_{11} \ast R_{12} \]
From NFAs to Regular Expressions

\[
\begin{align*}
& q_s & \xrightarrow{\varepsilon} & q_1 & \xrightarrow{R_{11}} & q_1 & \xrightarrow{R_{12}^*} & q_2 & \xrightarrow{R_{21}} & q_2 & \xrightarrow{R_{22}^*} & q_f \\
& q_s & \xrightarrow{\varepsilon} & q_1 & \xrightarrow{R_{11}} & q_1 & \xrightarrow{R_{12}} & q_2 & \xrightarrow{R_{21}} & q_2 & \xrightarrow{R_{22}^*} & q_f \\
\end{align*}
\]
From NFAs to Regular Expressions
From NFAs to Regular Expressions

\[ R_{11} * R_{12} \]

\[ R_{21} R_{11} * R_{12} \]
From NFAs to Regular Expressions

Note: We’re using union to combine these transitions together.
From NFAs to Regular Expressions

\[ R_{22} \cup R_{21} R_{11}^* R_{12} \]
From NFAs to Regular Expressions

\[ R_{22} \cup R_{21} R_{11}^* R_{12} \]
From NFAs to Regular Expressions

\[
\begin{align*}
q_s & \xrightarrow{R_{11} \ast R_{12}} q_2 \\
q_2 & \xrightarrow{\varepsilon} q_f \\
R_{22} & \cup R_{21} R_{11} \ast R_{12}
\end{align*}
\]
From NFAs to Regular Expressions

\[
R_{22} \cup R_{21} \quad R_{11}^* \quad R_{12}
\]
From NFAs to Regular Expressions

What should we put on this transition?

\[ q_s \xrightarrow{R_{11} \ast R_{12}} q_2 \xrightarrow{\varepsilon} q_f \]

\[ R_{22} \cup R_{21} R_{11} \ast R_{12} \]
From NFAs to Regular Expressions

\[ R_{11} \ast R_{12} (R_{22} \cup R_{21} R_{11} \ast R_{12})^* \varepsilon \]

\[ R_{22} \cup R_{21} R_{11} \ast R_{12} \]
From NFAs to Regular Expressions

\[ R_{11} \ast R_{12} (R_{22} \cup R_{21} R_{11} \ast R_{12})^* \varepsilon \]
From NFAs to Regular Expressions

\[ R_{11} \ast R_{12} \ (R_{22} \cup R_{21} \ R_{11} \ast R_{12})^\ast \ \varepsilon \]

\[ R_{22} \cup R_{21} \ R_{11} \ast R_{12} \]
From NFAs to Regular Expressions

\[ R_{11} \ast R_{12} \ (R_{22} \cup R_{21} \ R_{11} \ast R_{12}) \ast \varepsilon \]
From NFAs to Regular Expressions

$R_{11} \ast R_{12} (R_{22} \cup R_{21} R_{11} \ast R_{12}) \ast \varepsilon$
From NFAs to Regular Expressions

\[ R_{11}^* R_{12} (R_{22} \cup R_{21} R_{11}^* R_{12})^* \]
From NFAs to Regular Expressions

\[
q_s \xrightarrow{\text{start}} R_{11}^* R_{12} (R_{22} \cup R_{21} R_{11}^* R_{12})^* q_f
\]
From NFAs to Regular Expressions

\[ R_{11} \ast R_{12} (R_{22} \cup R_{21} R_{11} \ast R_{12})^* \]
The State-Elimination Algorithm

- Start with an NFA $N$ for the language $L$.
- Add a new start state $q_s$ and accept state $q_f$ to the NFA.
  - Add an $\varepsilon$-transition from $q_s$ to the old start state of $N$.
  - Add $\varepsilon$-transitions from each accepting state of $N$ to $q_f$, then mark them as not accepting.
- Repeatedly remove states other than $q_s$ and $q_f$ from the NFA by “shortcutting” them until only two states remain: $q_s$ and $q_f$.
- The transition from $q_s$ to $q_f$ is then a regular expression for the NFA.
The State-Elimination Algorithm

• To eliminate a state $q$ from the automaton, do the following for each pair of states $q_0$ and $q_1$, where there's a transition from $q_0$ into $q$ and a transition from $q$ into $q_1$:
  
  • Let $R_{in}$ be the regex on the transition from $q_0$ to $q$.
  
  • Let $R_{out}$ be the regex on the transition from $q$ to $q_1$.
  
  • If there is a regular expression $R_{stay}$ on a transition from $q$ to itself, add a new transition from $q_0$ to $q_1$ labeled $((R_{in})(R_{stay})^*(R_{out}))$.
    
  • If there isn't, add a new transition from $q_0$ to $q_1$ 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 $R_1 \cup R_2 \cup ... \cup R_k$. 
Our Transformations

- DFA to NFA: direct conversion
- NFA to Regexp: state elimination
- NFA to DFA: subset construction
- Regexp to NFA: Thompson's algorithm
**Theorem:** The following are all equivalent:

- \( L \) is a regular language.
- There is a DFA \( D \) such that \( \mathcal{L}(D) = L \).
- There is an NFA \( N \) such that \( \mathcal{L}(N) = L \).
- There is a regular expression \( R \) such that \( \mathcal{L}(R) = L \).
Why This Matters

• The equivalence of regular expressions and finite automata has practical relevance.
  • Regular expression matchers have all the power available to them of DFAs and NFAs.
  • This also is hugely theoretically significant: the regular languages can be assembled “from scratch” using a small number of operations!
Next Time

• **Applications of Regular Languages**
  • Answering “so what?”
• **Intuiting Regular Languages**
  • What makes a language regular?
• **The Myhill-Nerode Theorem**
  • The limits of regular languages.