Regular Expressions

Recap from Last Time

Automata Progression

- **DFAs** (Deterministic Finite Automata)
  = **NFAs** (DFA + superpowers of guessing)
- **PDAs** (NFA + a stack to use as memory)
  We'll actually explore an equivalent mechanism called CFGs, you'll see soon...
- **Turing Machines** (DFA or NFA + an infinite array to use as memory)

New

At each step, we'll try to find a problem (language) that's too hard for the existing machine type, and motivate moving up a level in power.

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 Σ, the ***concatenation*** of L_1 and L_2 is the language L_1L_2 defined as

$$\boldsymbol{L_1L_2 = \{\, wx \mid w \in L_1 \text{ and } x \in L_2 \,\}}$$

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

$$L_1L_2 = \{ \textbf{aaa}, \textbf{abb}, \textbf{baaa}, \textbf{babb}, \textbf{bbaa}, \textbf{bbbb} \}$$

Lots and Lots of Concatenation

- Consider the language L = { **aa**, **b** }
- $L^0 = \{\varepsilon\}$
- $LL = L^2$ is the set of strings formed by concatenating pairs of strings in L.

  { **aaaa**, **aab**, **baa**, **bb** }
- $LLL = L^3$ is the set of strings formed by concatenating triples of strings in L.

  { **aaaaaa**, **aaaab**, **aabaa**, **aabb**, **baaaa**, **baab**, **bbaa**, **bbb**}
- $LLLL = L^4$ is the set of strings formed by concatenating quadruples of strings in L.

  { **aaaaaaaa**, **aaaaaab**, **aaaabaa**, **aaaabb**, **aabaaaa**,
  aabaab, **aabbaa**, **aabbb**, **baaaaaa**, **baaaab**, **baabaa**,
  baabb, **bbaaaa**, **bbaab**, **bbbaa**, **bbbb**}

The Kleene Closure

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

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

Closure Properties

- ***Theorem:*** If L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:
  - $\overline{L}_1$
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
- These properties are called ***closure properties of the regular languages***.

Quick check 1:
Let Σ = {1, 2, 3, a, b, c}.
Let L_1 = {aa, b}, L_2 = {33, 2}
be languages over Σ.

Name two strings string **in** $\overline{L}_1$.

Closure Properties

- ***Theorem:*** If L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:
  - $\overline{L_1}$
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
- These properties are called ***closure properties of the regular languages***.

Quick check 2:
Let Σ = {1, 2, 3, a, b, c}.
Let L_1 = {aa, b}, L_2 = {33, 2}
be languages over Σ.

Name two strings **in** $L_1 \cup L_2$.

Closure Properties

- ***Theorem:*** If L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:
  - $\overline{L_1}$
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
- These properties are called ***closure properties of the regular languages***.

Quick check 3:
Let Σ = {1, 2, 3, a, b, c}.
Let L_1 = {aa, b}, L_2 = {33, 2}
be languages over Σ.

Name two strings **in** $L_1 \cap L_2$.

Closure Properties

- ***Theorem:*** If L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:
  - $\overline{L_1}$
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
- These properties are called ***closure properties of the regular languages***.

Quick check 4:
Let Σ = {1, 2, 3, a, b, c}.
Let L_1 = {aa, b}, L_2 = {33, 2}
be languages over Σ.

Name two strings **in** $L_1 L_2$.

Closure Properties

- ***Theorem:*** If L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:
  - $\overline{L_1}$
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
- These properties are called ***closure properties of the regular languages***.

Quick check 5:
Let Σ = {1, 2, 3, a, b, c}.
Let L_1 = {aa, b}, L_2 = {33, 2}
be languages over Σ.

Name two strings **in** L_1^*.

Defining Regular Languages

- We currently have several tools for showing a language L is regular:
  - Construct a **DFA** for L.
    - *We used this for a variety of languages, as well as proving closure for complement.*
  - Construct an **NFA** for L.
    - *We used this for a variety of languages, as well as proving closure for union, concatenation, and Kleene *.*
  - Combine several simpler regular languages together via **closure properties** to form L.
    - *We used this for proving closure for intersection.*
    - *(Today we'll greatly expand on this last idea.)*

New Stuff!

Another View of Regular Languages:

Regular Expressions (regex)

xkcd

80 minutes from now, you will be able to say this

Regular Expressions

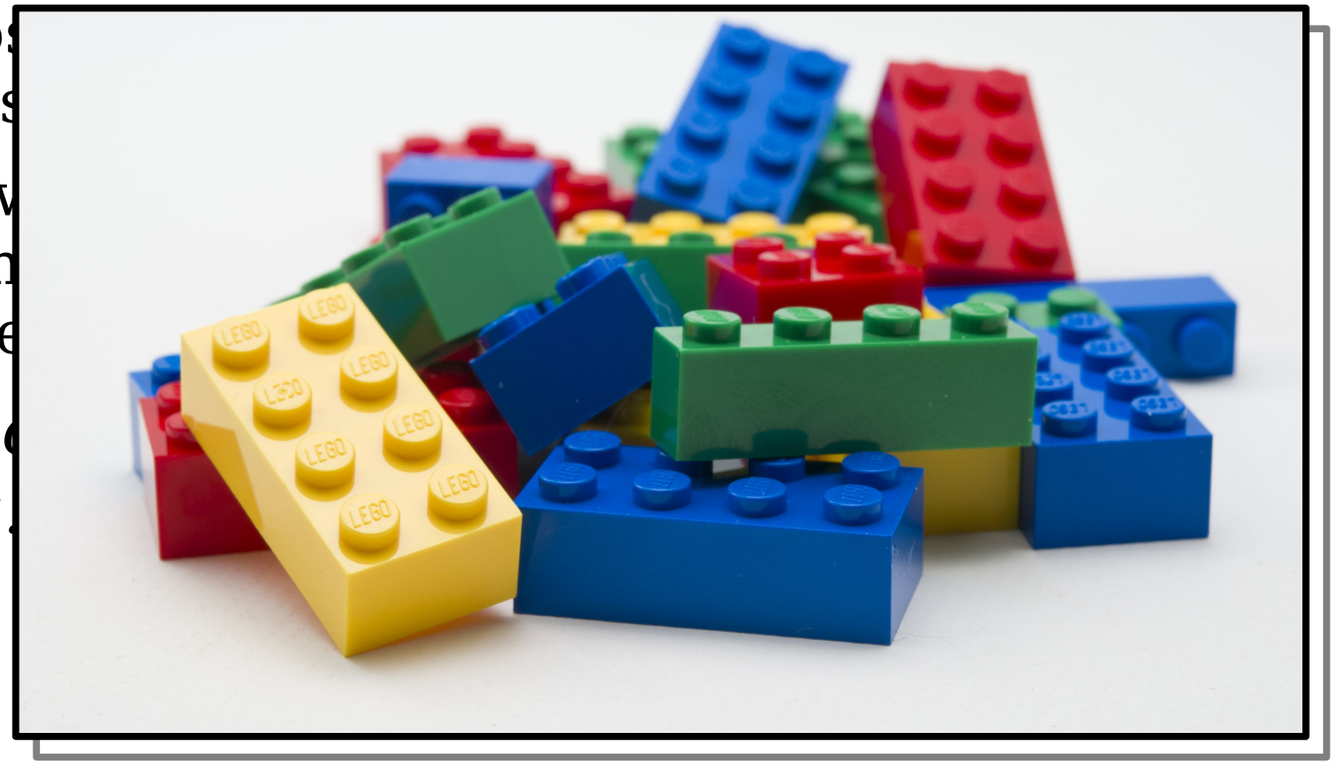
- ***Regular expressions*** are a way of describing a language via a string pattern-match representation.
  - They’re used all over in practical day-to-day programming:
    - Data validation tools in just about every programming language (python, Javascript, C++, etc)
    - UNIX `grep` and `flex` tools to search files
    - Compilers

Constructing Regular Languages with RegEx

- ***Idea:*** Build up all regular languages as follows:
  - Start with a small set of super-simple languages we already know to be regular.
  - Using closure properties, combine these simple languages together to form more languages.
  - Because we only use closure properties, the results of even the most elaborate combinations are guaranteed to be regular.
- *This is a bottom-up approach to the regular languages.*

Constructing Regular Languages with RegEx

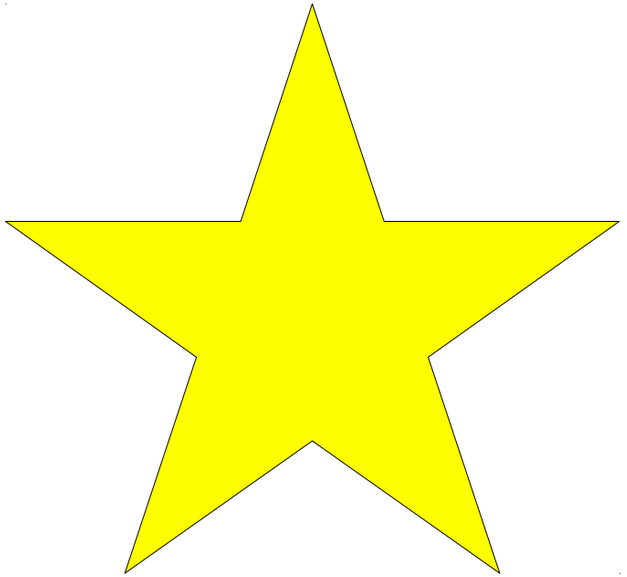
- ***Idea:*** Build up all regular languages as follows:
  - Start with a small set of super-simple languages we already know to be regular.
  - Using clos
    languages
  - Because w
    of even th
    guarantee
- *This is a be*
  languages.



RegEx “Starter Languages”

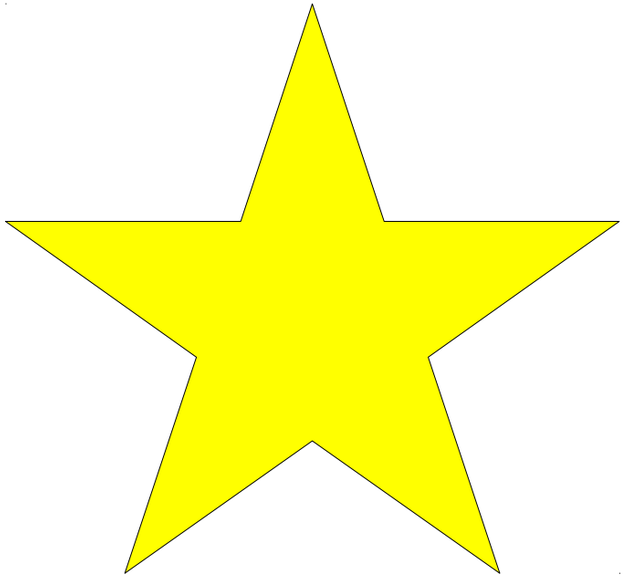
- The regular expressions begin with **three starter languages** *(a bit like base cases in induction!)*

  1. The symbol **Ø** is a regular expression that represents the empty language Ø.
  2. For any **a** ∈ Σ, the symbol **a** is a regular expression for the language {**a**}. (*So this “starter language” is actually many languages, one for each character in Σ.)*
  3. The symbol **ε** is a regular expression that represents the language {ε}.



Combining RegEx

- Let R_1 and R_2 be regular expressions (either one of the starter languages, or a more complex regex). We can combine them in these ways:
- ***Concatenation:*** $\boldsymbol{R_1R_2}$
- ***Union:*** $\boldsymbol{R_1 \cup R_2}$
- ***Kleene closure:*** $\boldsymbol{R_1*}$
- ***Parentheses:*** $\boldsymbol{(R_1)}$ may be used to control order of operations, and for reading clarity.



Operator Precedence

- Here's the operator precedence for regular expressions:

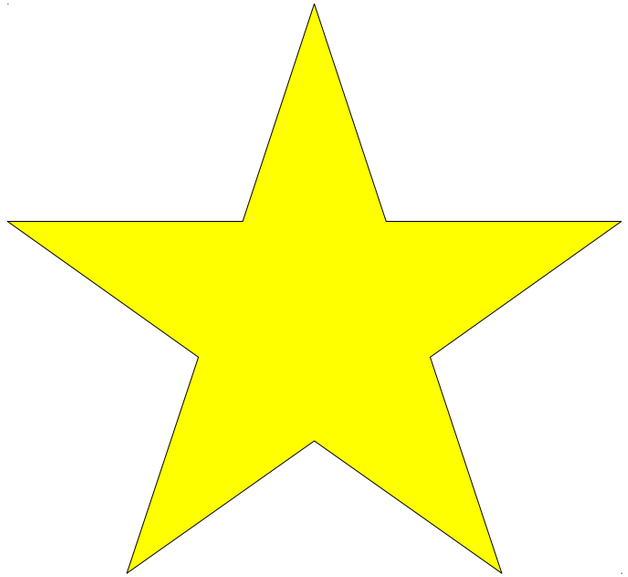
$$(R_1)$$

$$R_1*$$

$$R_1R_2$$

$$R_1 \cup R_2$$

- So **ab*c∪d** is parsed as **((a(b*))c)∪d**



Regular Expression Examples

- The regular expression **`trick∪treat`** represents the language

  { **`trick`**, **`treat`** }.

- The regular expression **`booo*`** represents the regular language

  { **`boo`**, **`booo`**, **`boooo`**, … }.

- The regular expression **`candy!(candy!)*`** represents the regular language

  { **`candy!`**, **`candy!candy!`**, **`candy!candy!candy!`**, … }.

Regular Expressions, Formally

- The ***language of a regular expression*** is the language described by that regular expression.
- Formally:
  - $\mathscr{L}(\mathbf{\varepsilon}) = \{\varepsilon\}$
  - $\mathscr{L}(\mathbf{\emptyset}) = \emptyset$
  - $\mathscr{L}(\mathbf{a}) = \{\mathbf{a}\}$
  - $\mathscr{L}(R_1 R_2) = \mathscr{L}(R_1)\,\mathscr{L}(R_2)$
  - $\mathscr{L}(R_1 \cup R_2) = \mathscr{L}(R_1) \cup \mathscr{L}(R_2)$
  - $\mathscr{L}(R^*) = \mathscr{L}(R)^*$
  - $\mathscr{L}(\mathbf{(}R\mathbf{)}) = \mathscr{L}(R)$

Regex quick check:

Let Σ = {a, b, c, d}.

Let $L_1 = \mathcal{L}(\mathbf{a(b \cup c)((d))})$ be a language over Σ.

Name two strings **in** L_1.

Name one string **not in** L_1.

Designing Regular Expressions

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

Designing Regular Expressions

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

$$(\mathbf{a} \cup \mathbf{b})^*\mathbf{aa}(\mathbf{a} \cup \mathbf{b})^*$$

Designing Regular Expressions

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

$$(\mathtt{a} \cup \mathtt{b})^*\mathtt{aa}(\mathtt{a} \cup \mathtt{b})^*$$

Designing Regular Expressions

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

$$(\mathbf{a} \cup \mathbf{b})^*\mathbf{aa}(\mathbf{a} \cup \mathbf{b})^*$$

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

Designing Regular Expressions

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

(a ∪ b)*aa(a ∪ b)*

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

Designing Regular Expressions

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

Σ*aaΣ*

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

Designing Regular Expressions

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

Designing Regular Expressions

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

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

ΣΣΣΣ

Designing Regular Expressions

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

ΣΣΣΣ

Designing Regular Expressions

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

ΣΣΣΣ

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

ΣΣΣΣ

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

$\mathbf{\Sigma^4}$

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

$$\mathbf{\Sigma^4}$$

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

Here are some candidate regular expressions for the language L. **How many** of these are correct? (Discuss specifically which with your neighbors.)

$$\mathbf{\Sigma^*a\Sigma^*}$$

$$\mathbf{b^*ab^* \cup b^*}$$

$$\mathbf{b^*(a \cup \varepsilon)b^*}$$

$$\mathbf{b^*a^*b^* \cup b^*}$$

$$\mathbf{b^*(a^* \cup \varepsilon)b^*}$$

Designing Regular Expressions

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

b*(a $\cup$ **ε)b***

Designing Regular Expressions

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

$$\mathtt{b}^*(\mathtt{a} \cup \varepsilon)\mathtt{b}^*$$

Designing Regular Expressions

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

$$\mathtt{b}^*(\mathtt{a} \cup \varepsilon)\mathtt{b}^*$$

bbbbabbb

bbbbbb

abbb

a

Designing Regular Expressions

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

$$\mathtt{b}^\star(\mathtt{a} \cup \varepsilon)\mathtt{b}^\star$$

bbbbabbb

bbbbbb

abbb

a

Designing Regular Expressions

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

b*a?b*

bbbbabbb

bbbbbb

abbb

a

A More Elaborate Design

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

A More Elaborate Design

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

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

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

A More Elaborate Design

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

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

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

A More Elaborate Design

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

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

$$a^+ \, (.a^+)^* \, @ \, a^+ \qquad (.a^+)^+$$

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

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

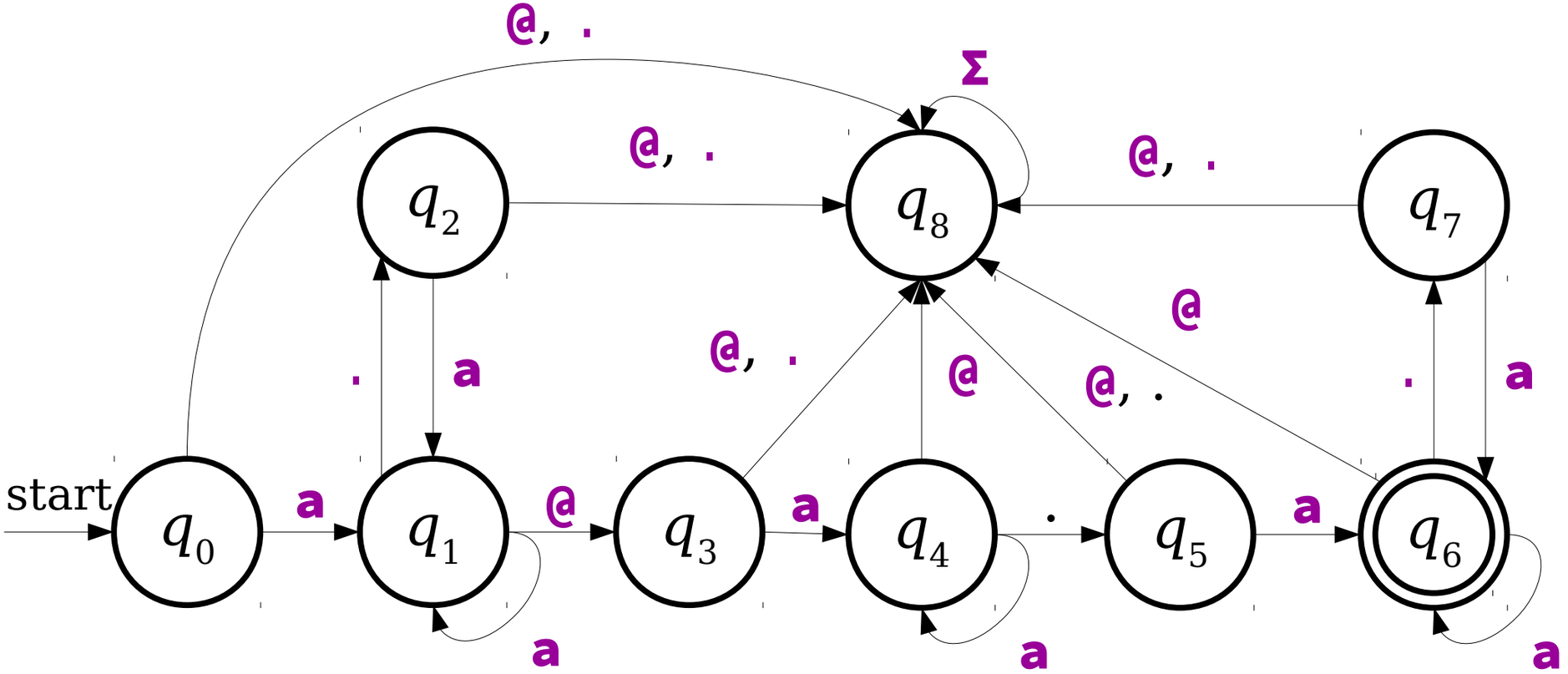
cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

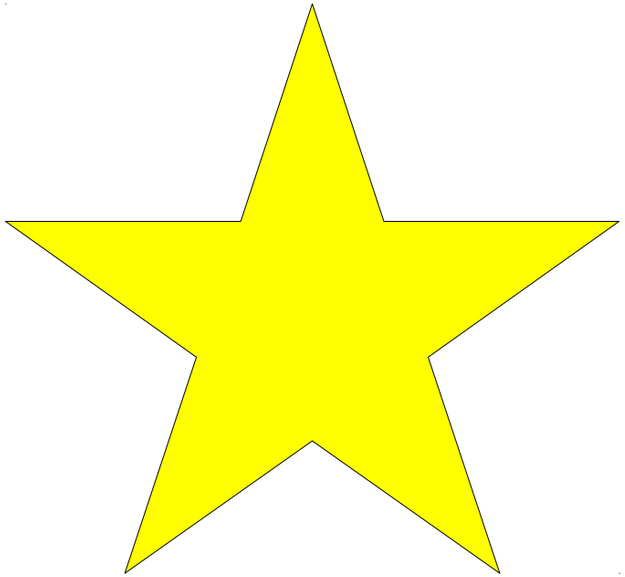
For Comparison

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

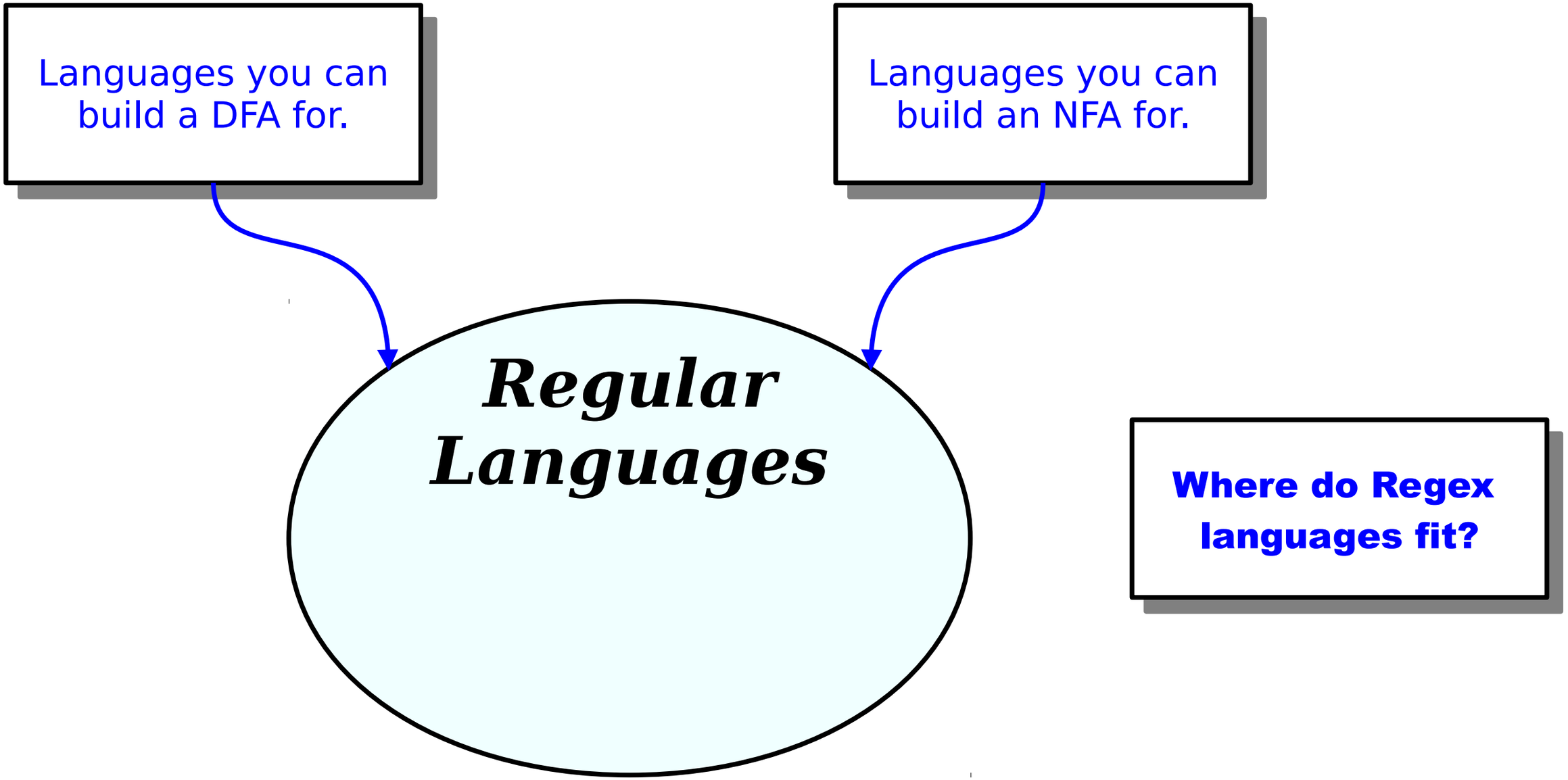


RegEx Shorthand Summary

- **R**^{n} is shorthand for **RR** … **R** (n times).
  - Edge case: define **R**^{0} = ε.
- **Σ** is shorthand for "any character in Σ."
- **R?** is shorthand for **(R ∪ ε)**, meaning "zero or one copies of R."
- **R**$^{+}$ is shorthand for **RR***, meaning "one or more copies of R."



The Lay of the Land


Languages you can build a DFA for.
Languages you can build an NFA for.
Regular Languages
Where do Regex languages fit?

The Power of Regular Expressions

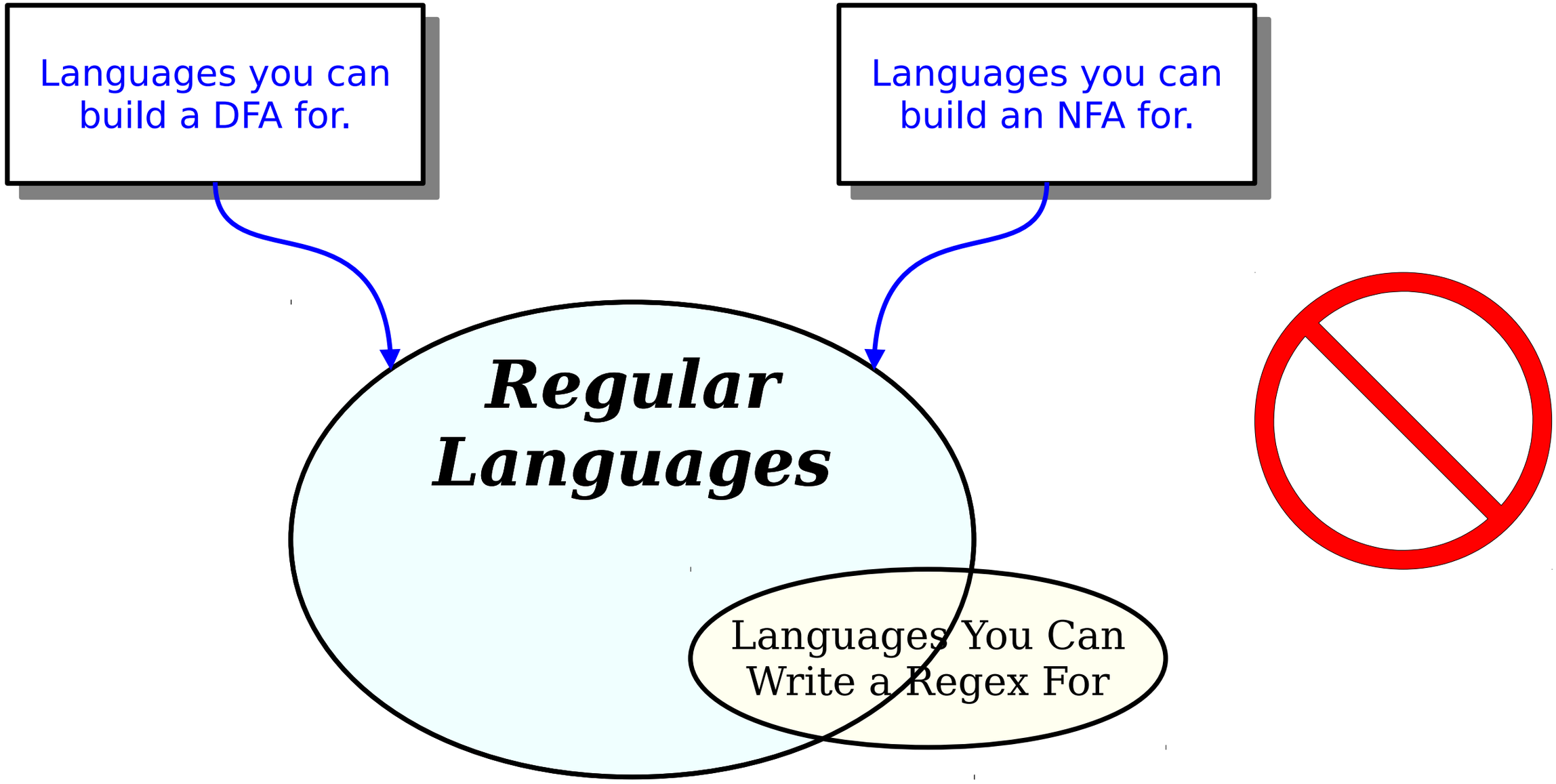
Theorem: If R is a regular expression, then $\mathscr{L}(R)$ is regular.

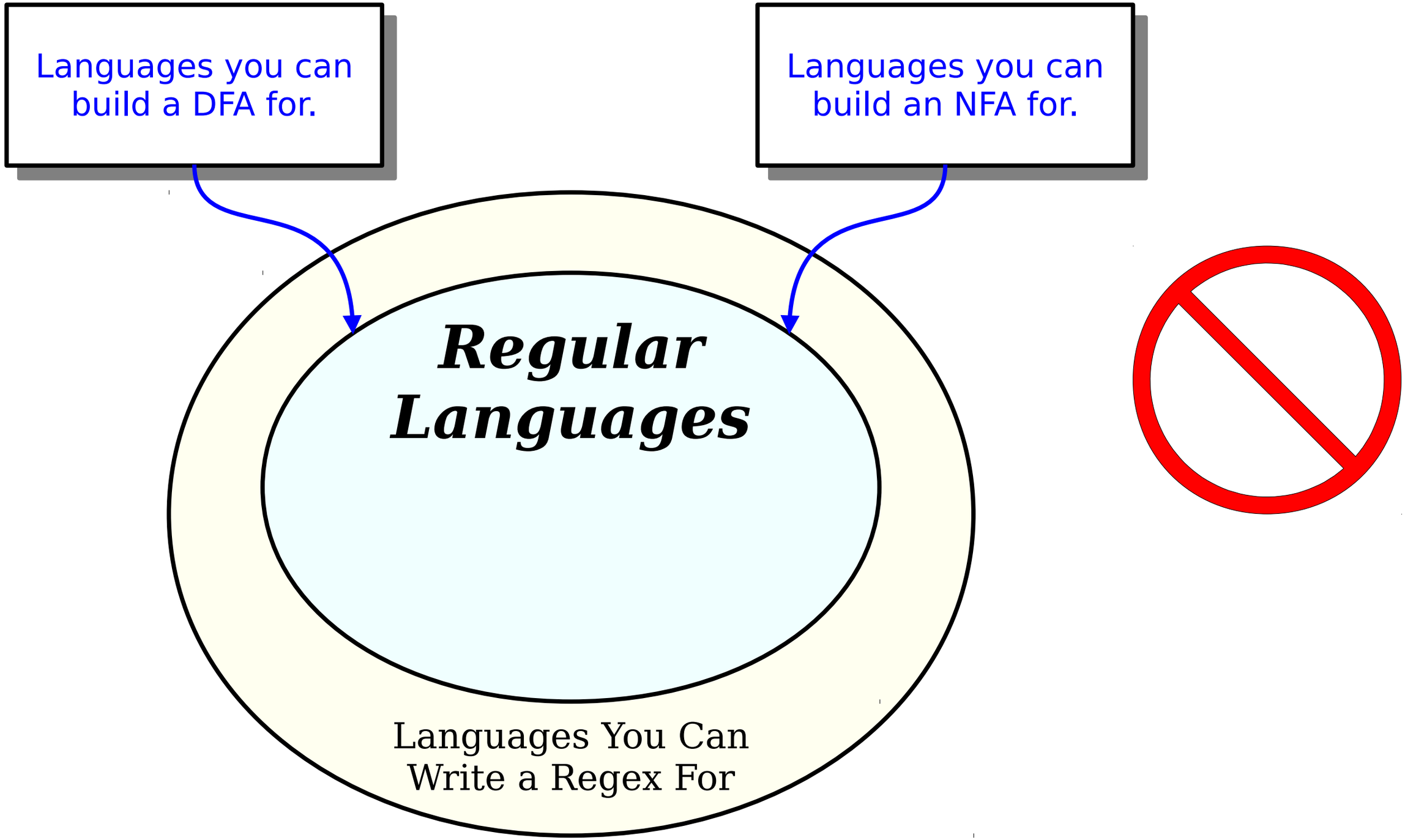
Proof idea: Use induction!

- **Base cases:** The atomic regular expressions all represent regular languages. *(Remember I said they were like induction base cases? Well they literally are!)*
- 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.
Regular Languages
Languages You Can Write a Regex For

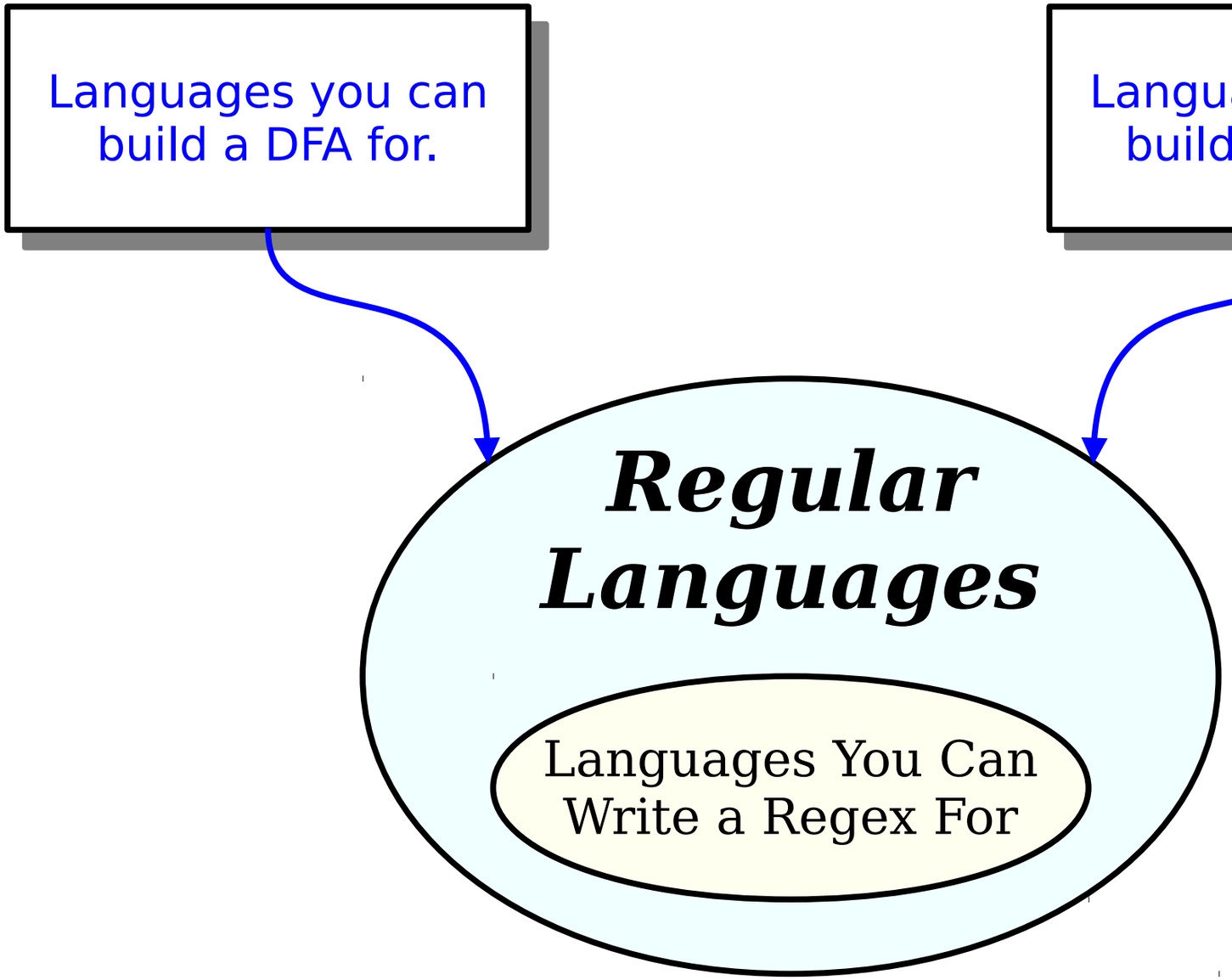

Languages you can build a DFA for.
Languages you can build an NFA for.
Regular Languages
Languages You Can Write a Regex For

Languages you can build a DFA for.

Languages you can build an NFA for.

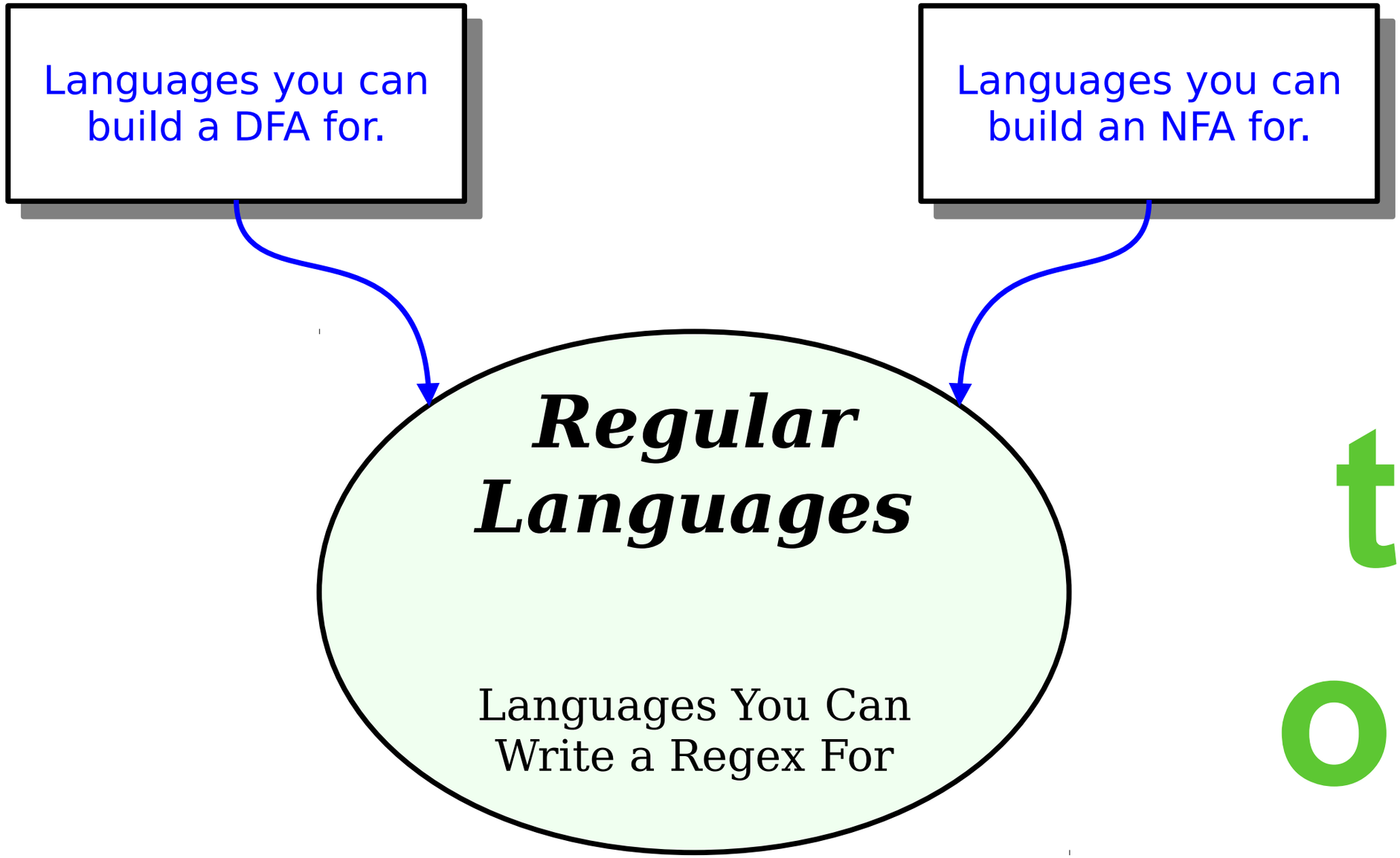
Regular Languages

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

This one?

or


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

this one?

The Power of Regular Expressions

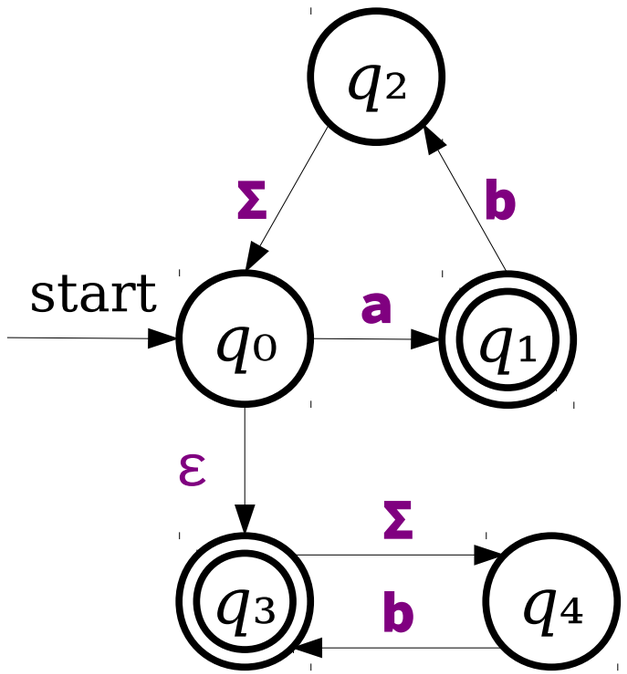
Theorem: For all languages L, if L is a regular language, then there is a regular expression for L.

This is not obvious!

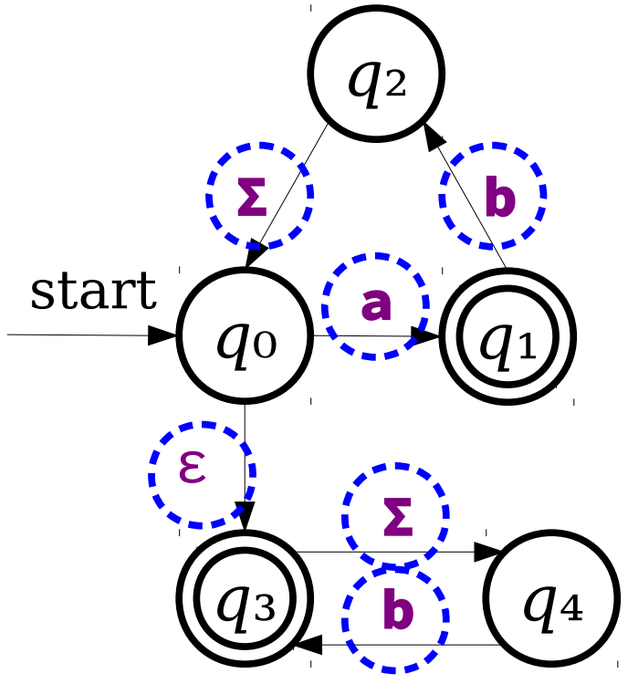
Proof plan:

- ***Assume:*** Pick an arbitrary regular language L.
- ***Want to show:*** We want to show there exists a regex for L.
- Since L is regular, there exists an NFA N for L.
- Describe how to transform N into a Regex, in a generalizable way. *For the next few slides, we'll explore how to do that.*

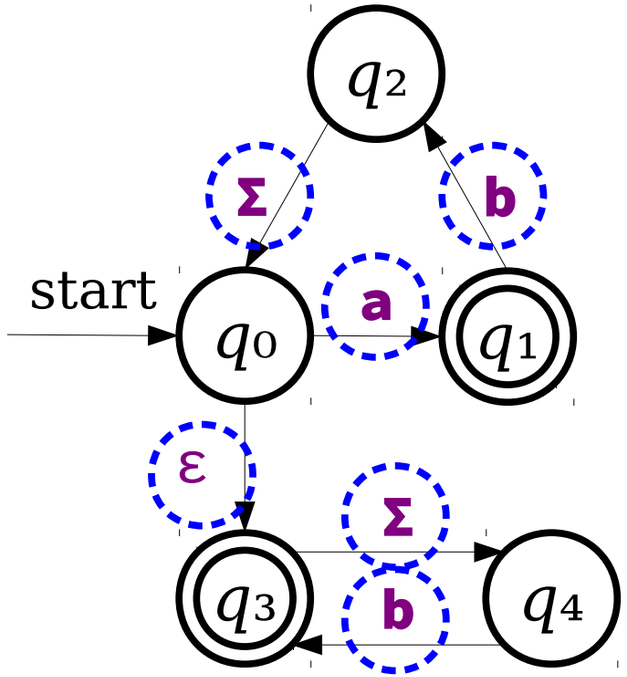
Generalizing NFAs



Generalizing NFAs

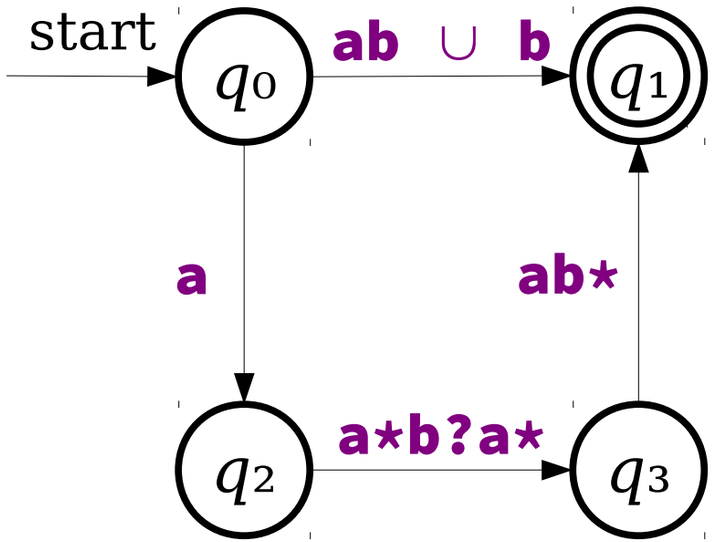


Generalizing NFAs

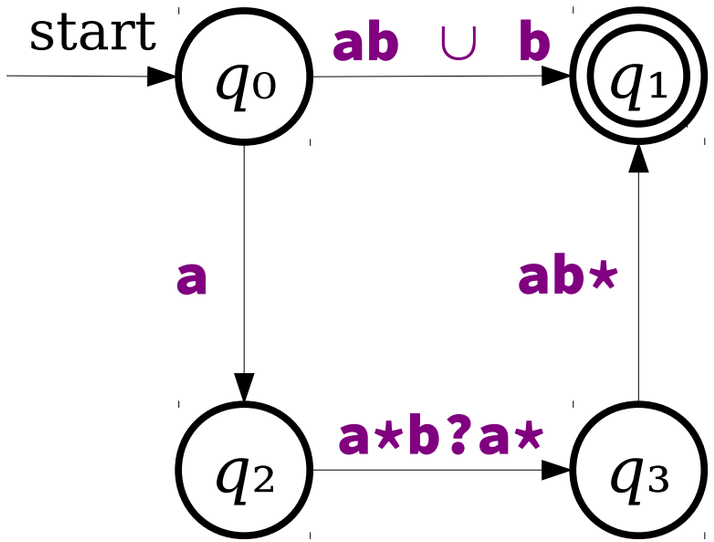


These are all regular expressions!

Generalizing NFAs

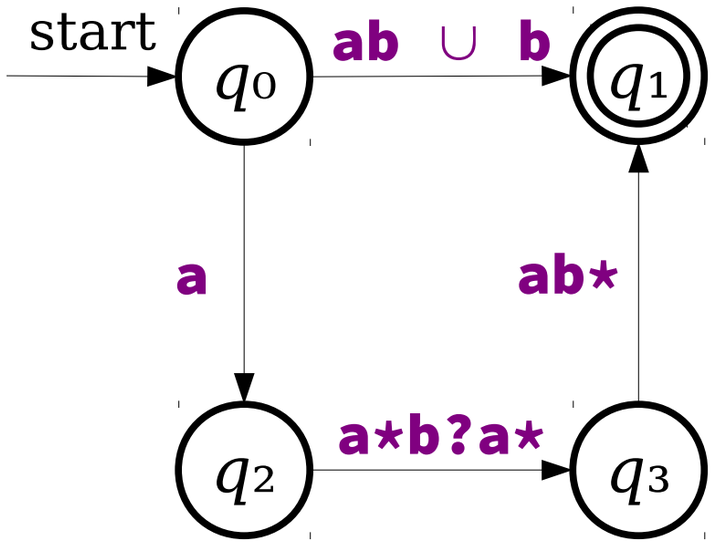


Generalizing NFAs



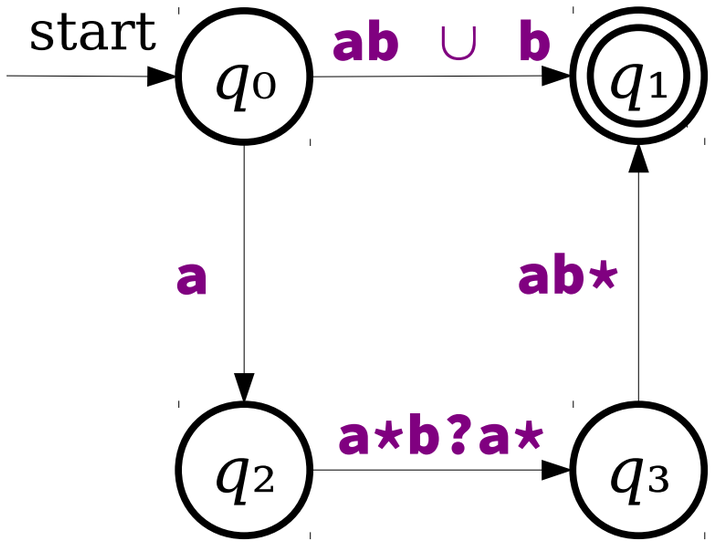
Note: Actual NFAs aren't allowed to have full regex transitions like these. This is just a thought experiment.

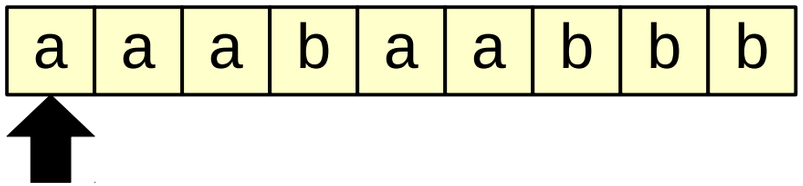
Generalizing NFAs



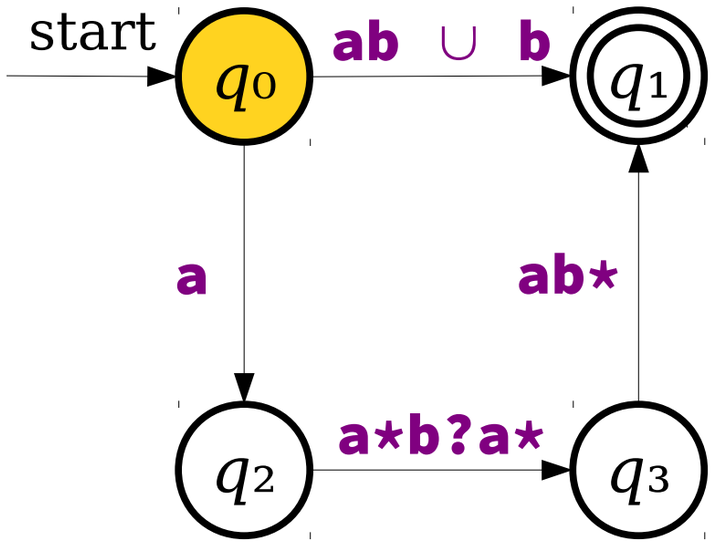
a	a	a	b	a	a	b	b	b

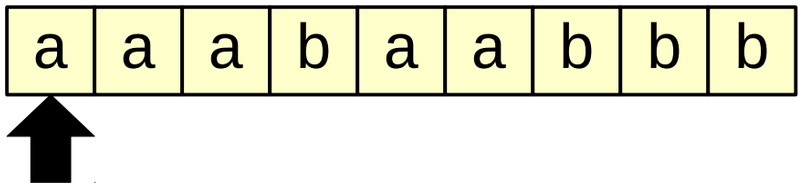
Generalizing NFAs



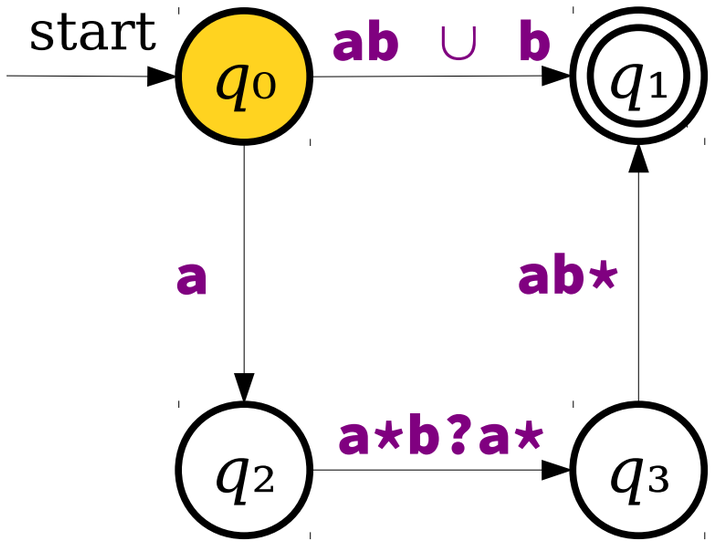


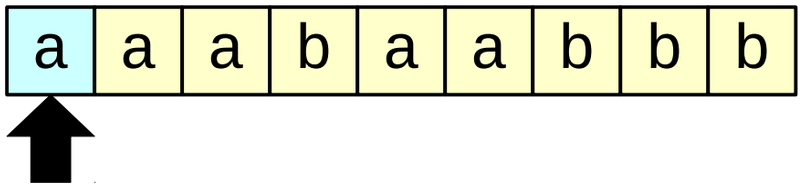
Generalizing NFAs



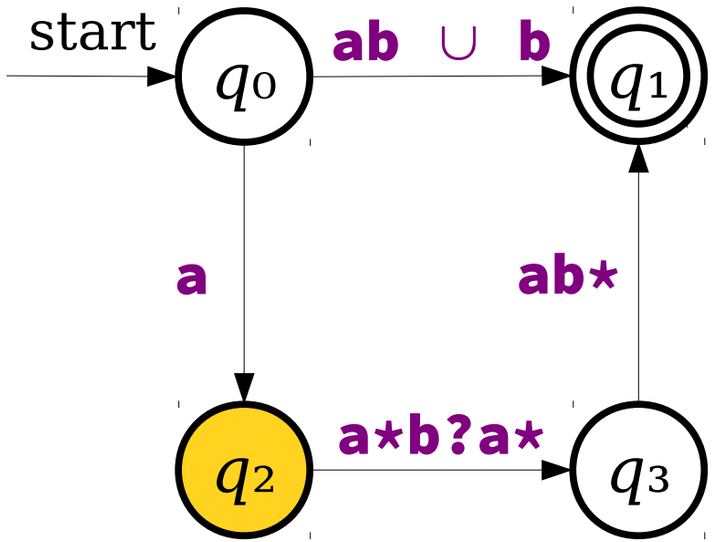


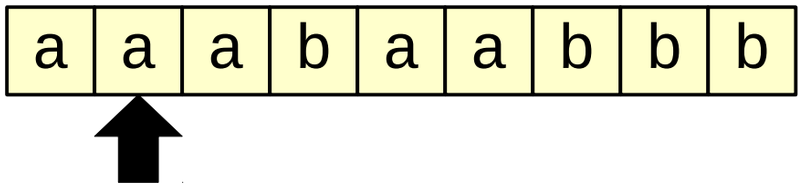
Generalizing NFAs



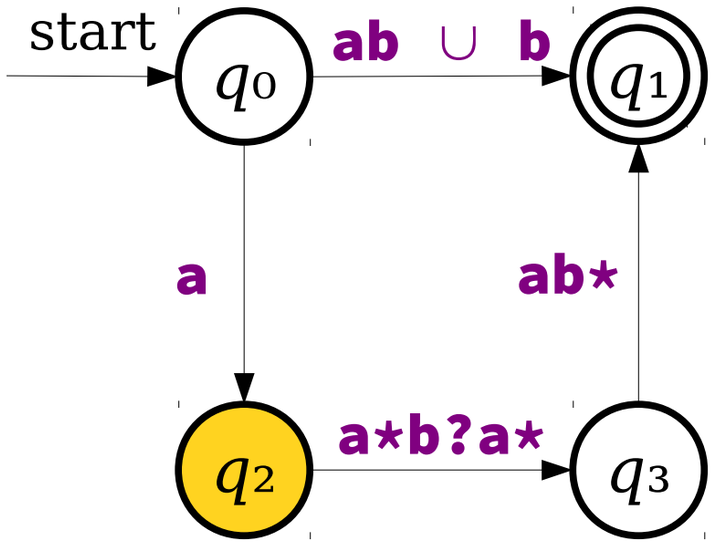


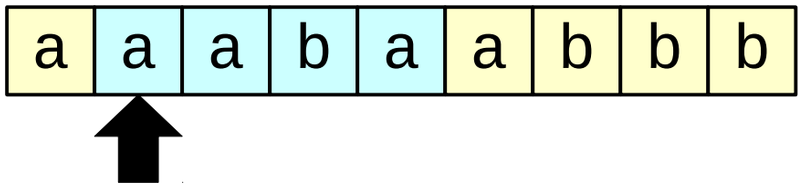
Generalizing NFAs



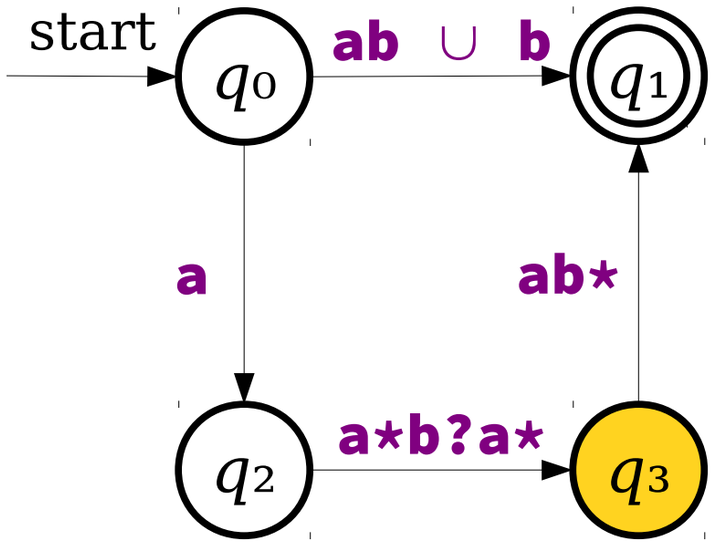


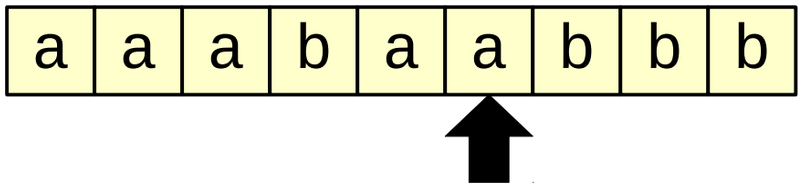
Generalizing NFAs



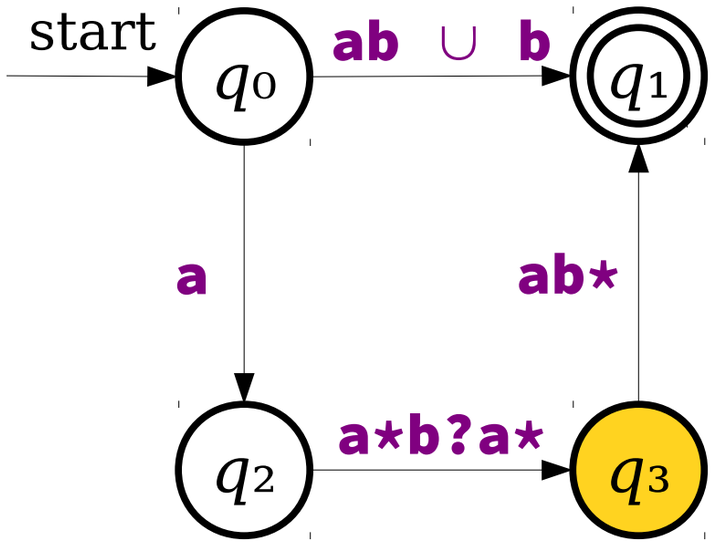


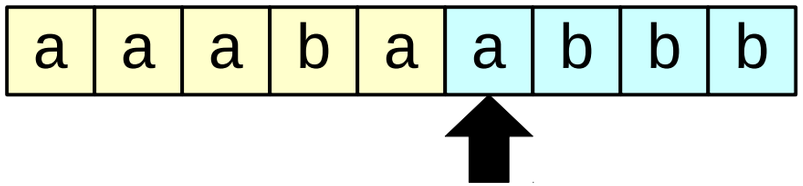
Generalizing NFAs



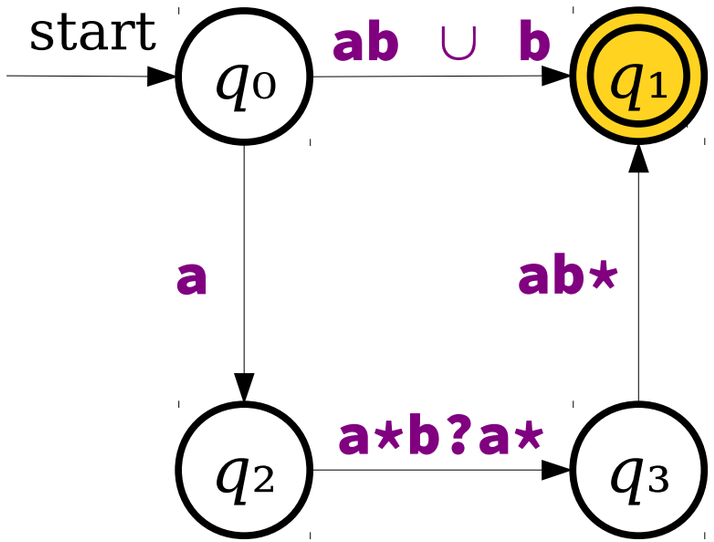


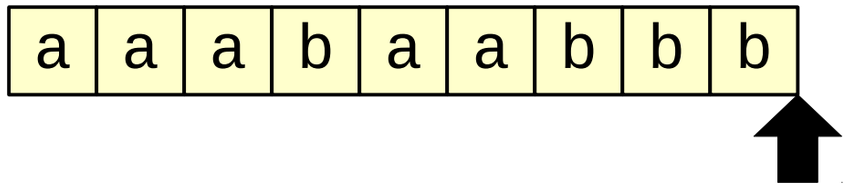
Generalizing NFAs





Generalizing NFAs





Key Idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions.

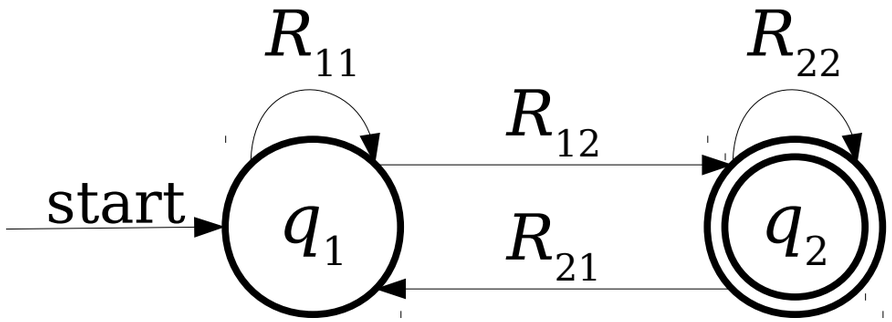
Generalizing NFAs

Key Idea 2: If we can convert an NFA into a generalized NFA that looks like this...

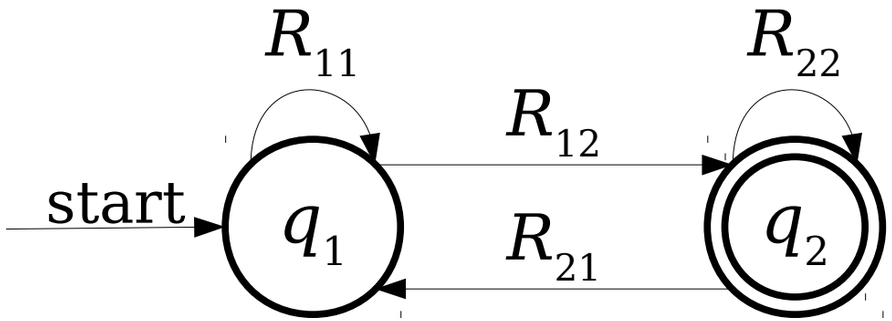
...then we can easily read off a regular expression for the original NFA.

...but not all NFAs look like this :-(
but maybe we can fix that?

From NFAs to Regular Expressions

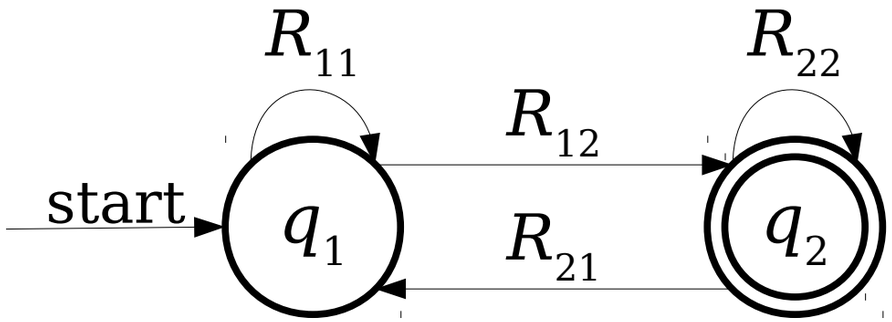


From NFAs to Regular Expressions



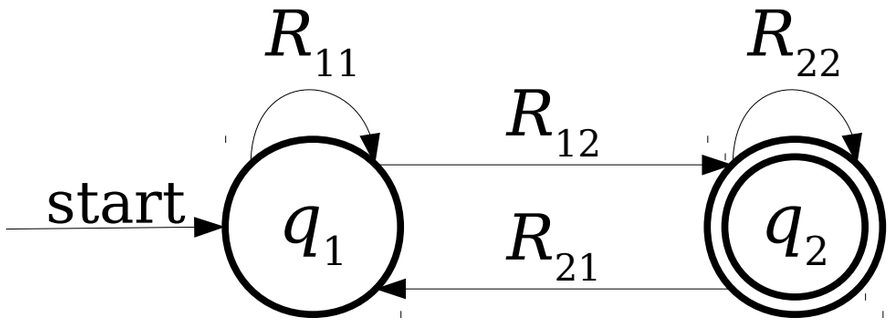
Here, R_{11}, R_{12}, R_{21}, and R_{22} are arbitrary regular expressions.

From NFAs to Regular Expressions



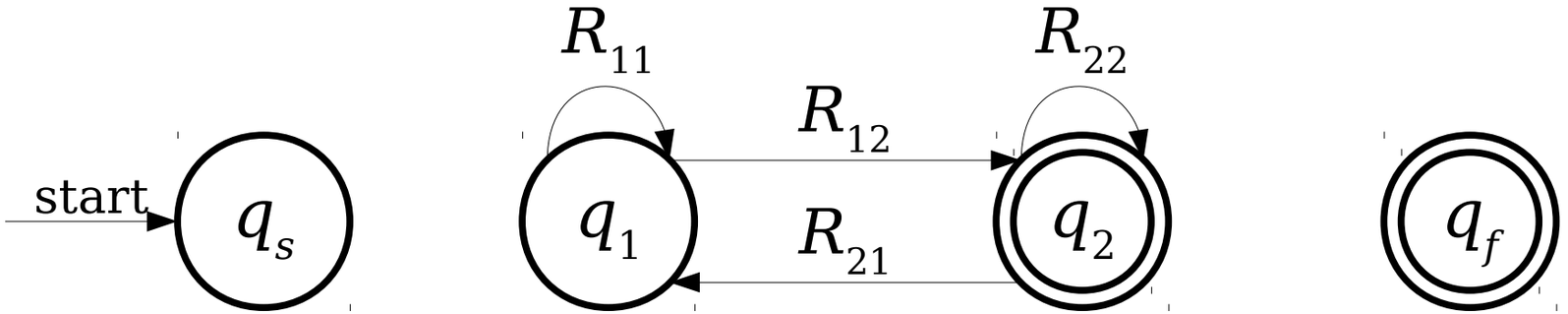
Key Idea 2 Goal: Somehow transform this NFA so that it looks like this:

From NFAs to Regular Expressions

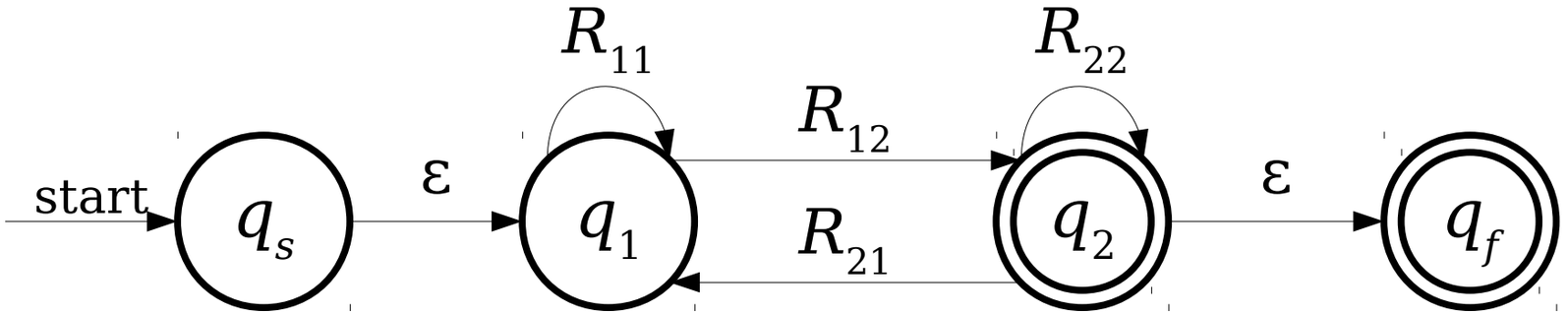


The first step is going to be a simple one...

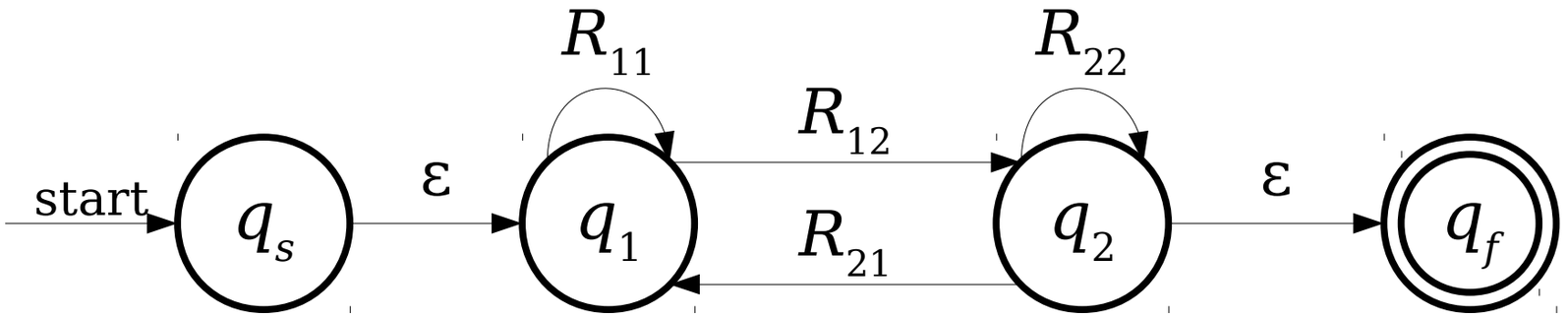
From NFAs to Regular Expressions



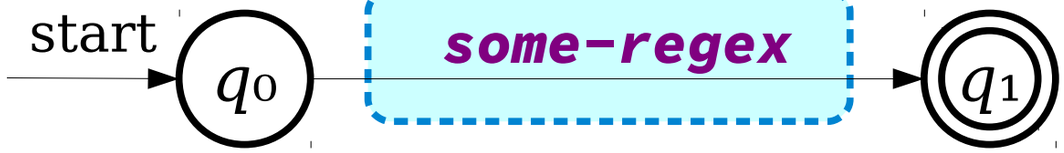
From NFAs to Regular Expressions



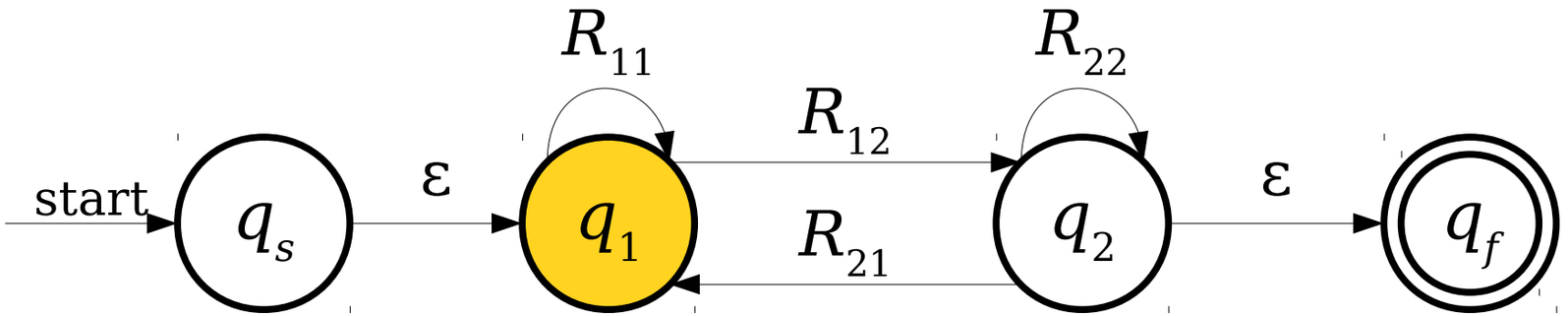
From NFAs to Regular Expressions



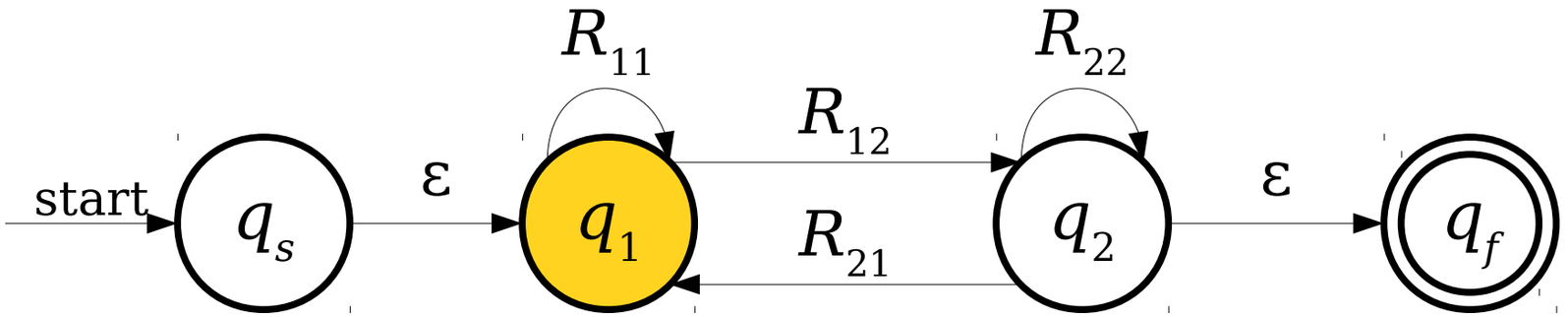
Key Idea 2 Goal Progress: Now it has clean start and end, but middle is messy. Need to get rid of those states.



From NFAs to Regular Expressions

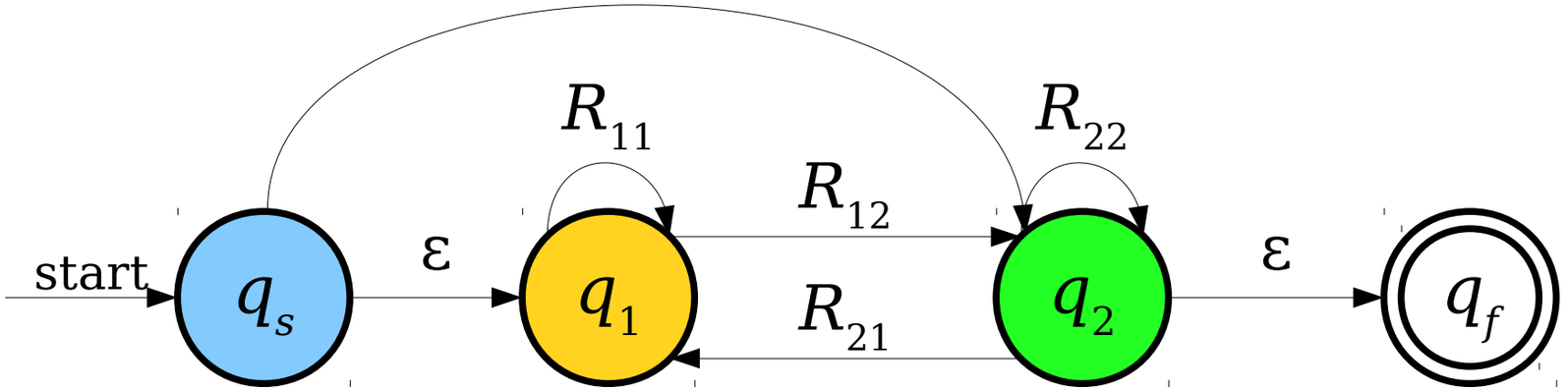


From NFAs to Regular Expressions

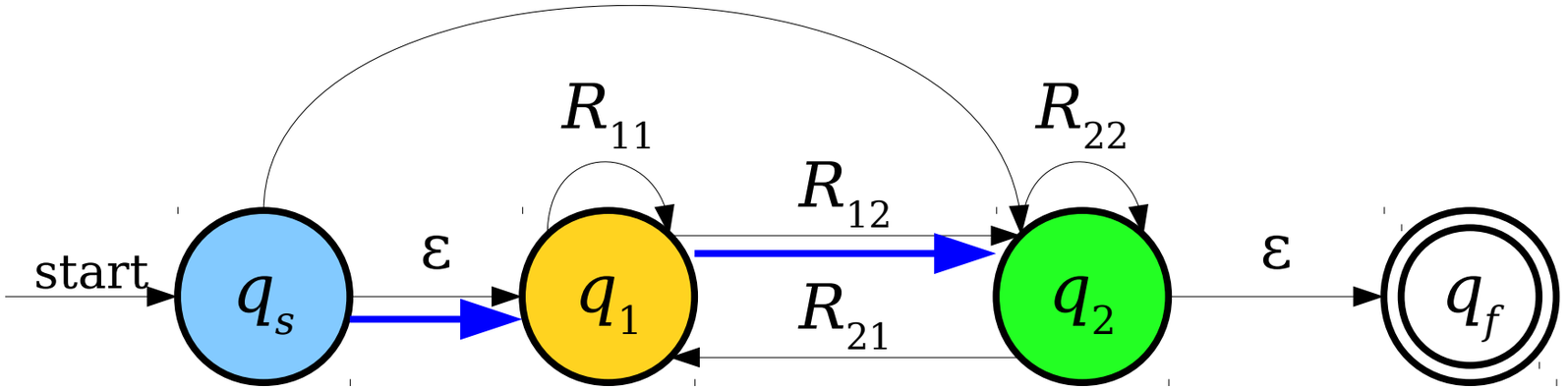


Could we eliminate this state from the NFA?

From NFAs to Regular Expressions

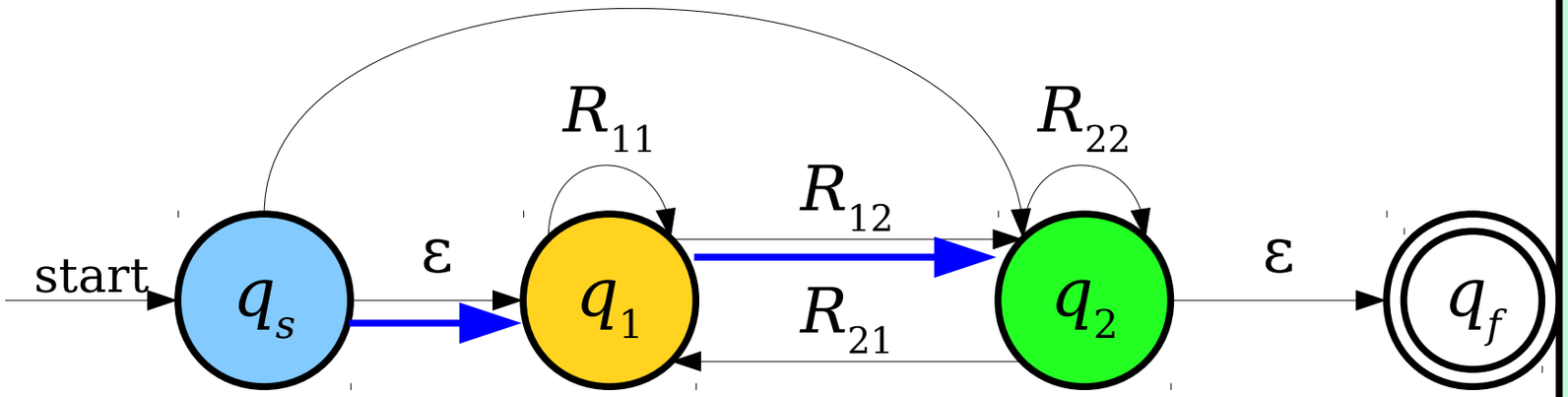


From NFAs to Regular Expressions



Here is a pattern that we might process when going from q_s to q_2: εR_{12}

From NFAs to Regular Expressions



Here is a pattern that we might process when going from q_s to q_2: εR_{12}

State elimination question:

How many of the following are also patterns might we process when going from q_s to q_2?

$\varepsilon R_{11} R_{12}$

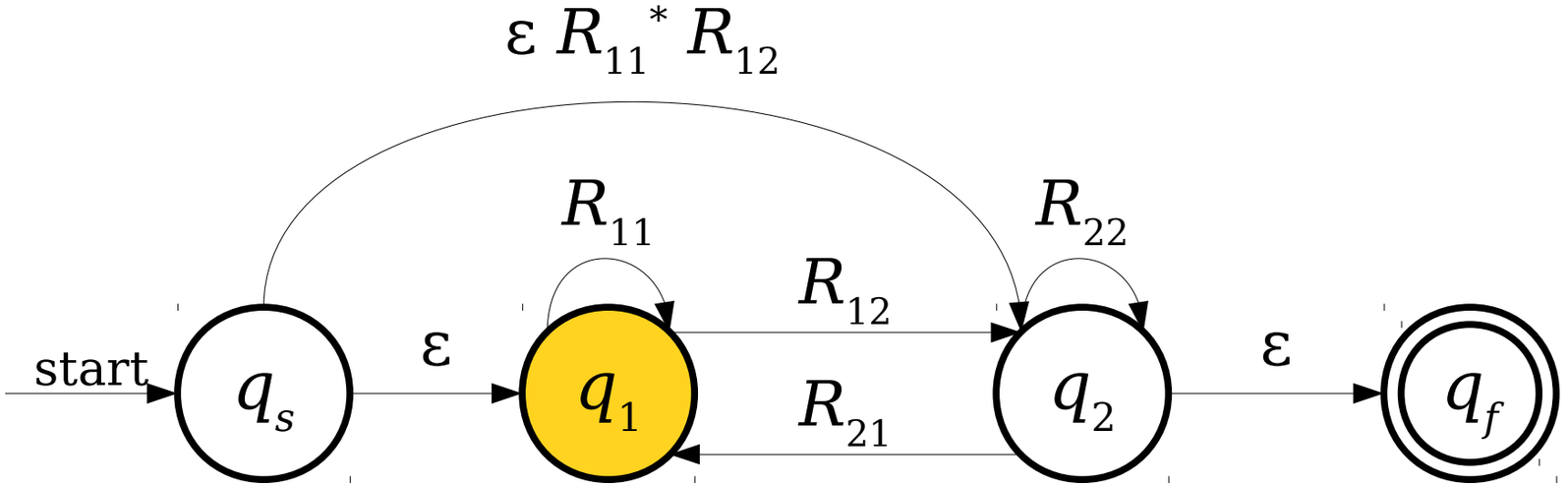
$\varepsilon R_{11} R_{11} R_{12}$

$\varepsilon R_{11} R_{12} R_{11}$

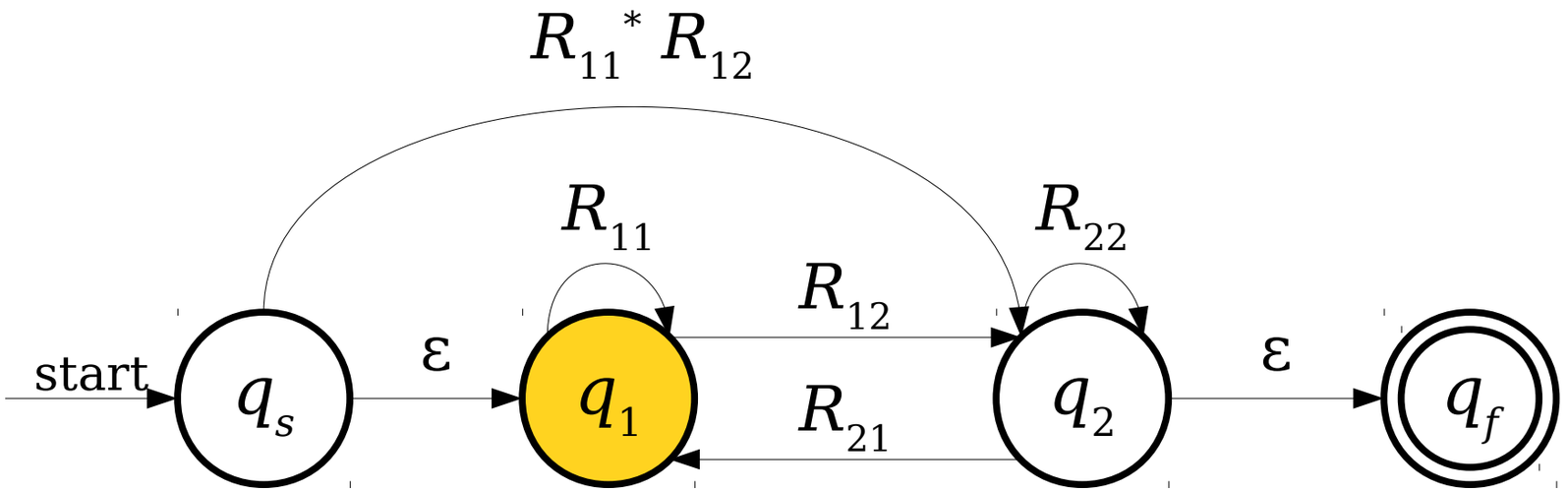
$\varepsilon R_{11} R_{11} R_{12}$

$\varepsilon R_{11} R_{11} R_{11} R_{11} R_{11} R_{12}$

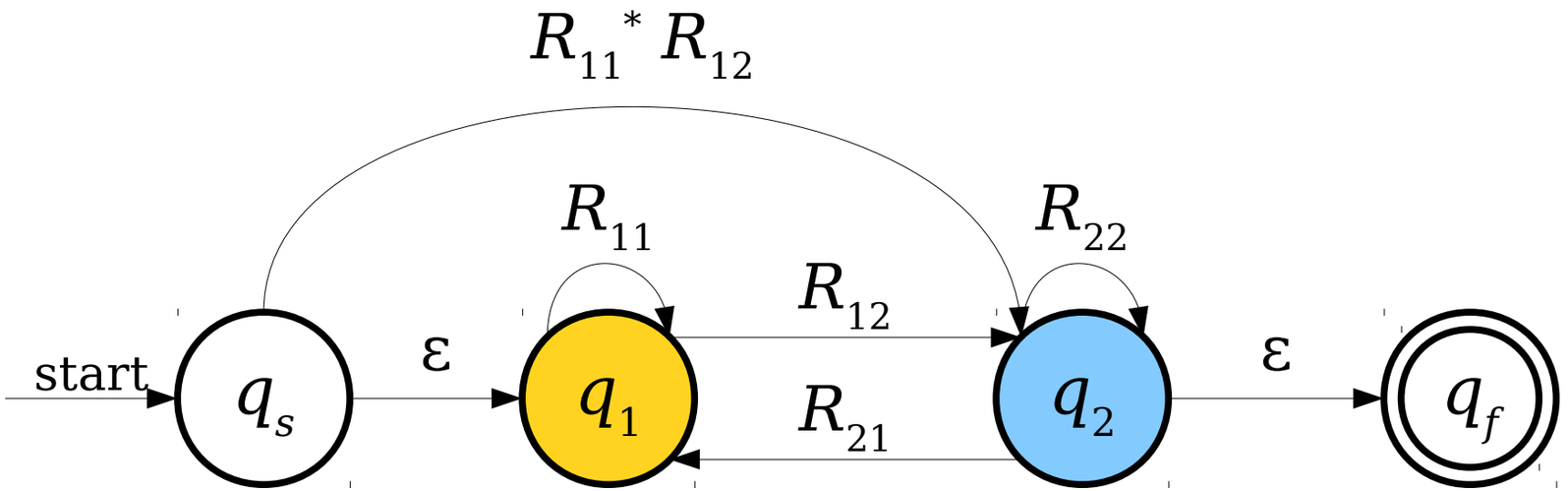
From NFAs to Regular Expressions



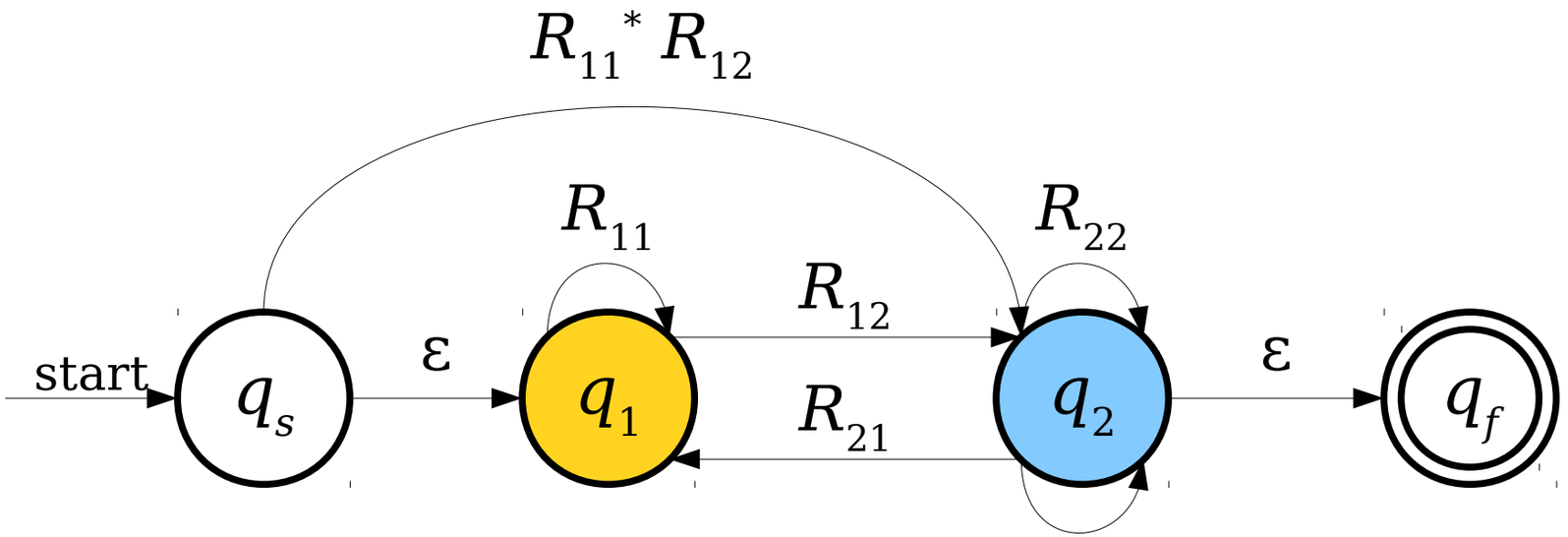
From NFAs to Regular Expressions



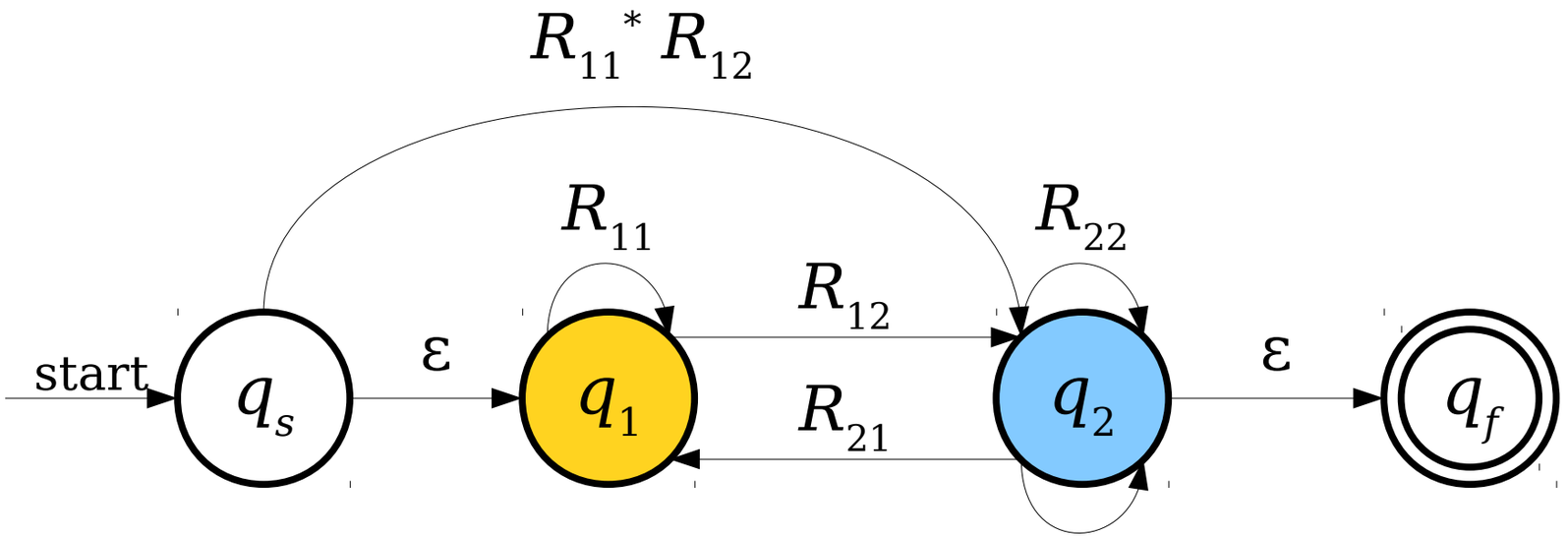
From NFAs to Regular Expressions



From NFAs to Regular Expressions

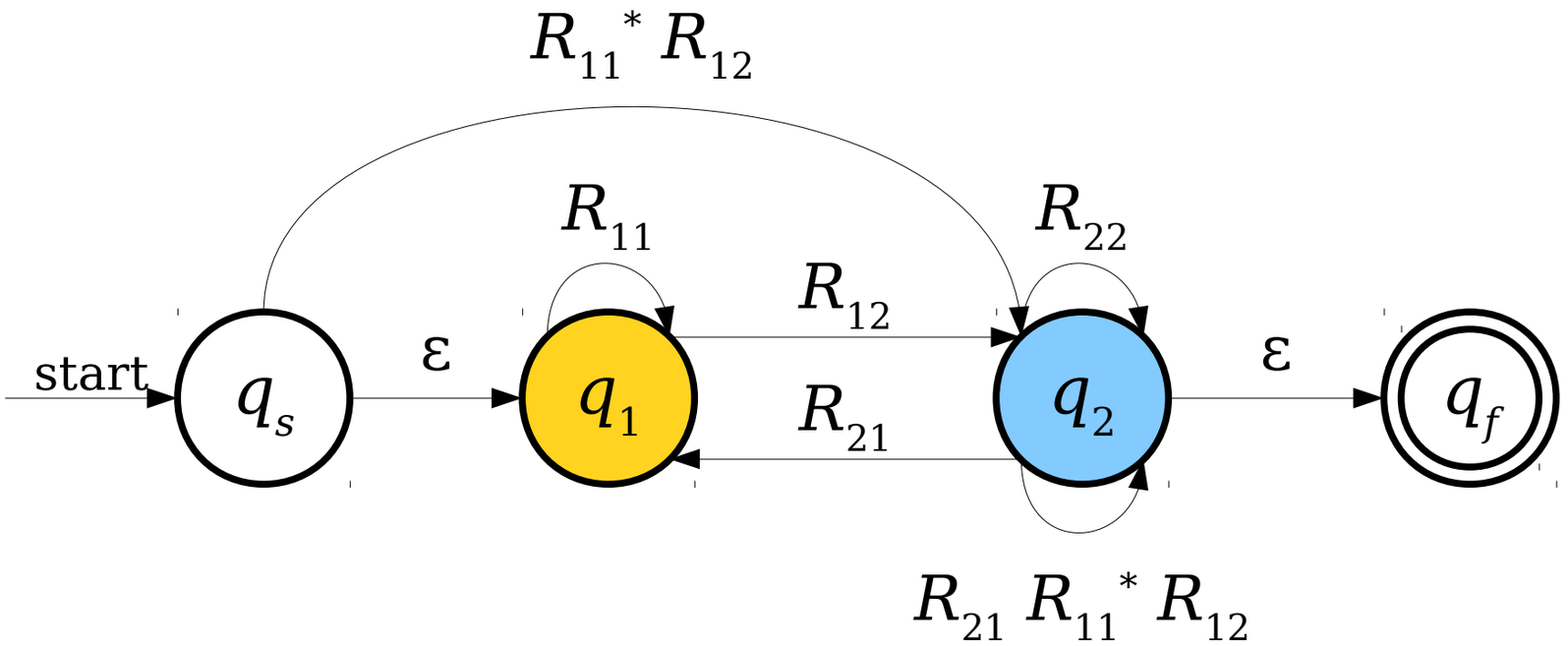


From NFAs to Regular Expressions



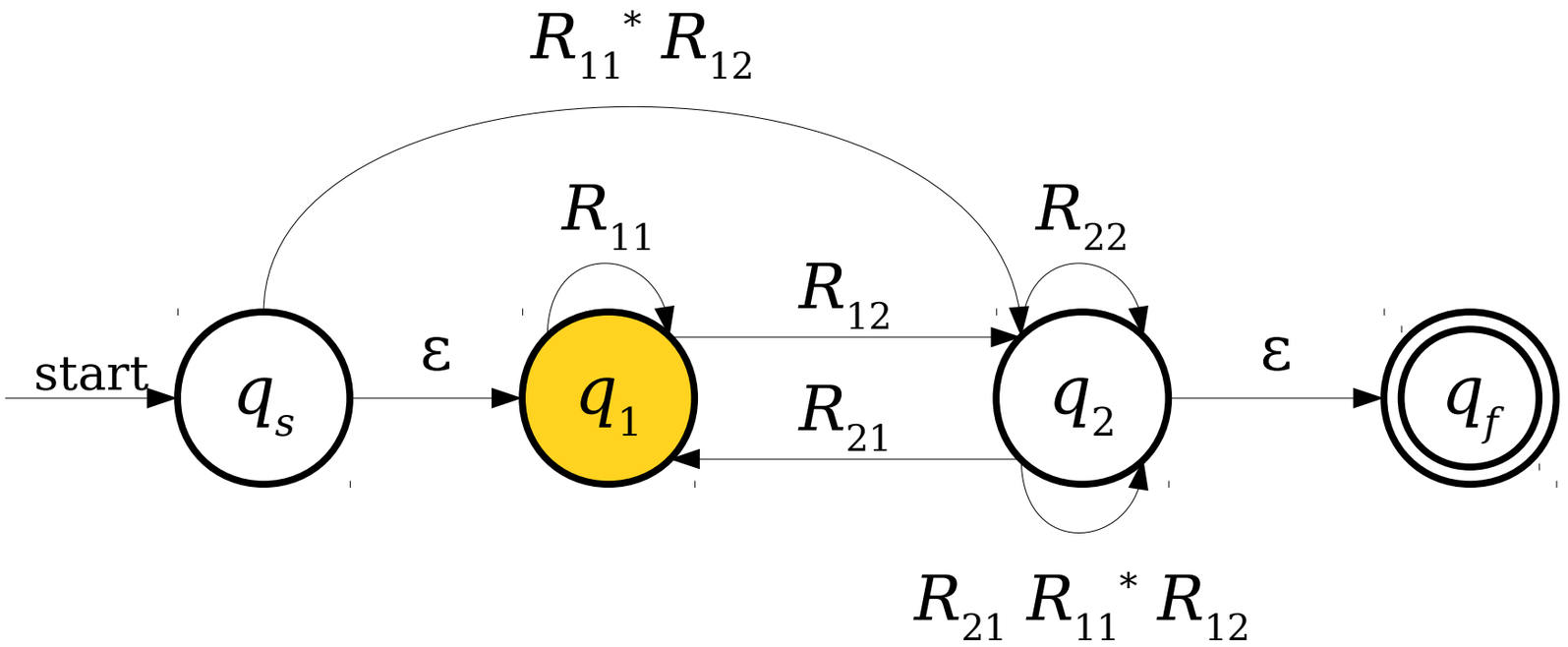
Here is a pattern that we might process when going from q_2 to q_2: $R_{21}R_{11}R_{12}$

From NFAs to Regular Expressions

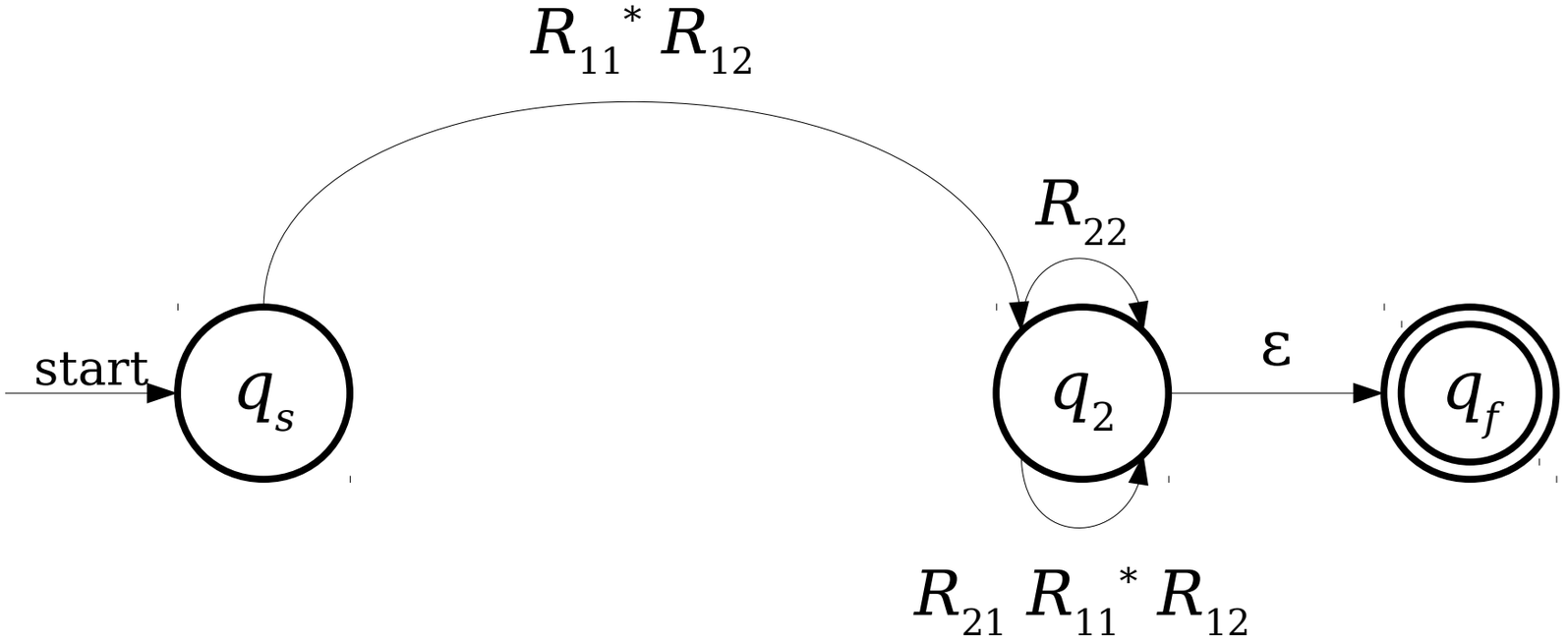


Here is a pattern that we might process when going from q_2 to q_2: $R_{21}R_{11}R_{12}$

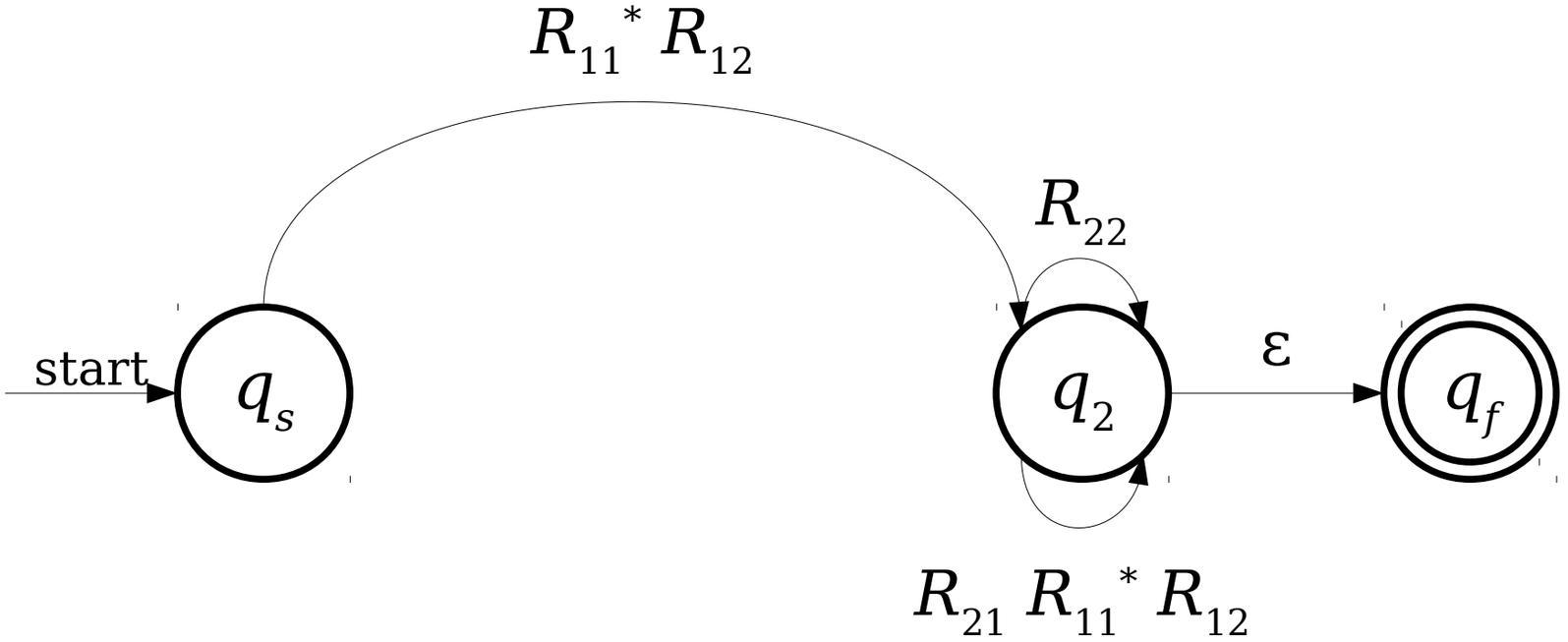
From NFAs to Regular Expressions



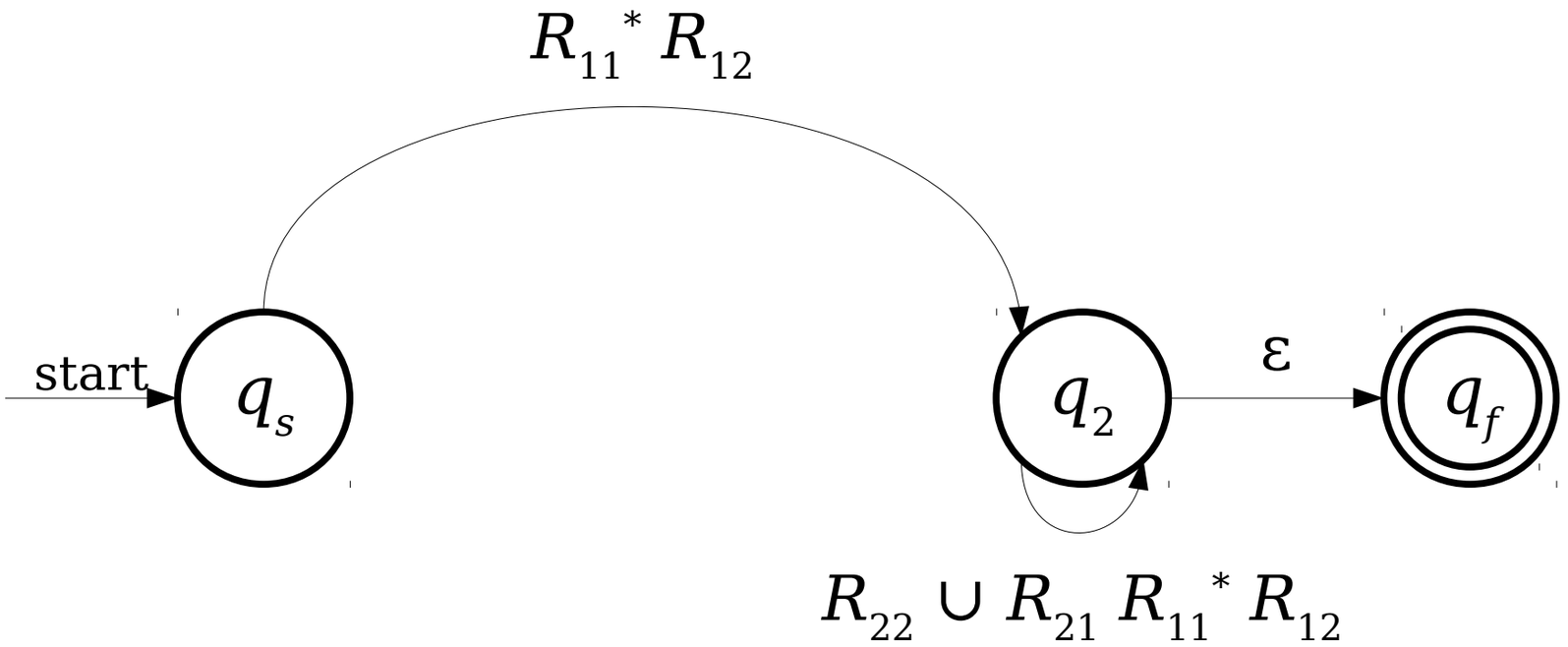
From NFAs to Regular Expressions



From NFAs to Regular Expressions

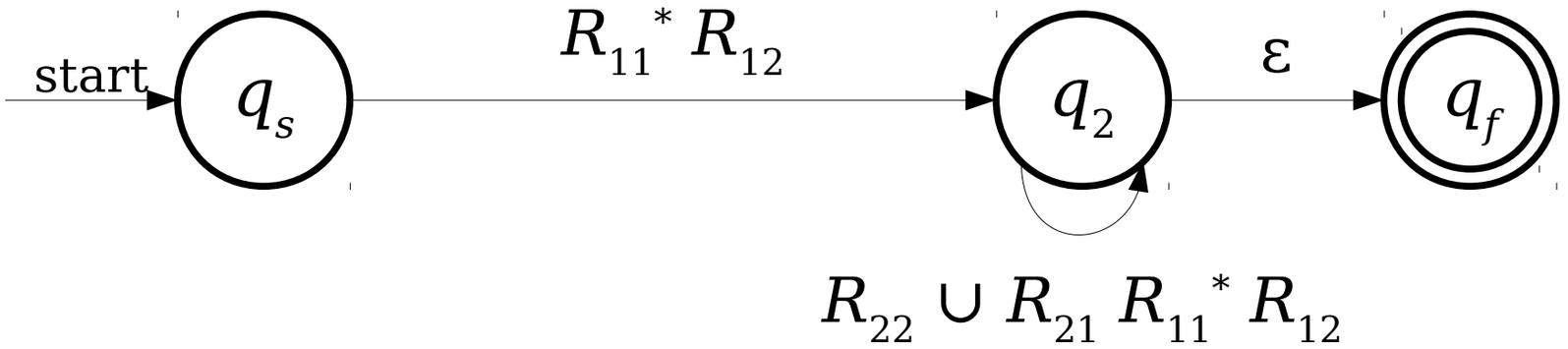


From NFAs to Regular Expressions

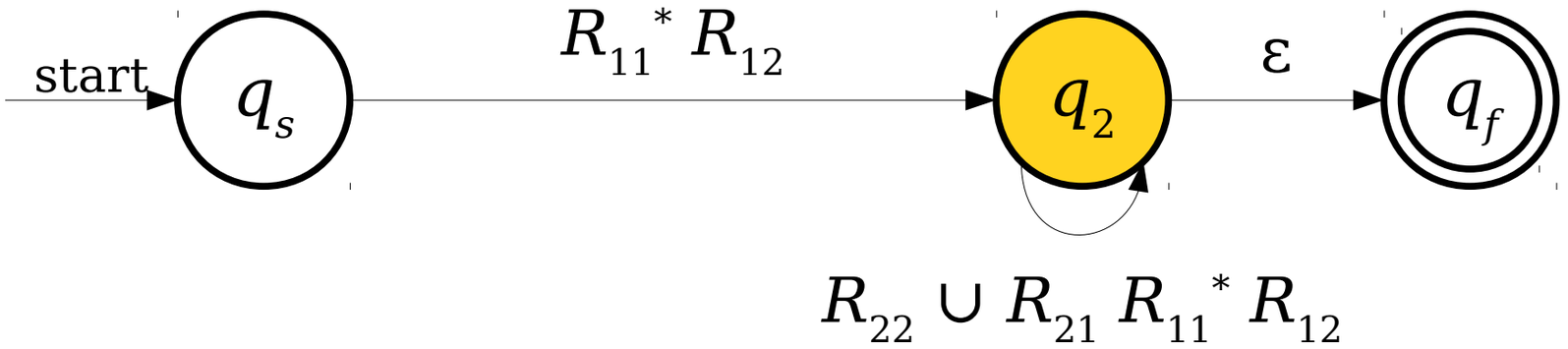


Note: We're using **union** to combine these transitions together.

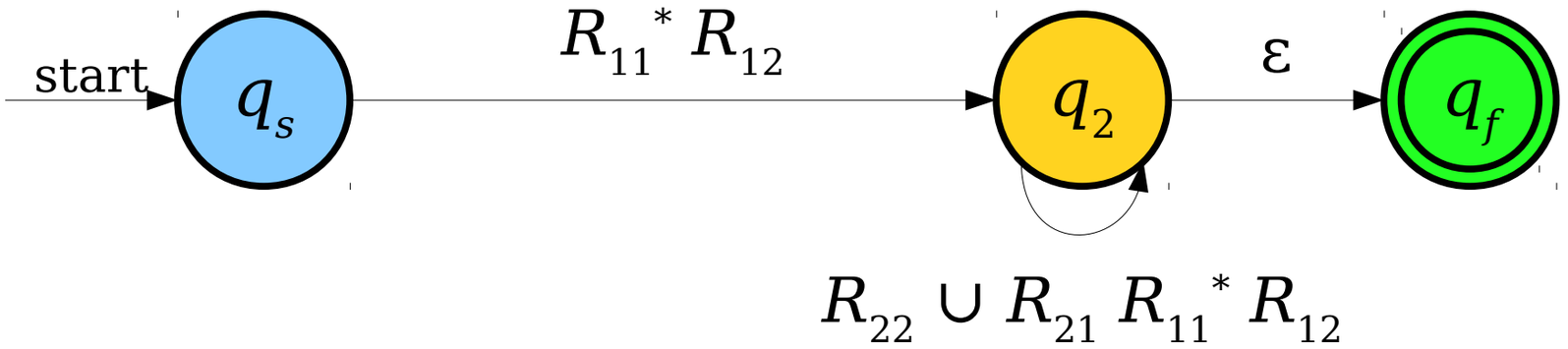
From NFAs to Regular Expressions



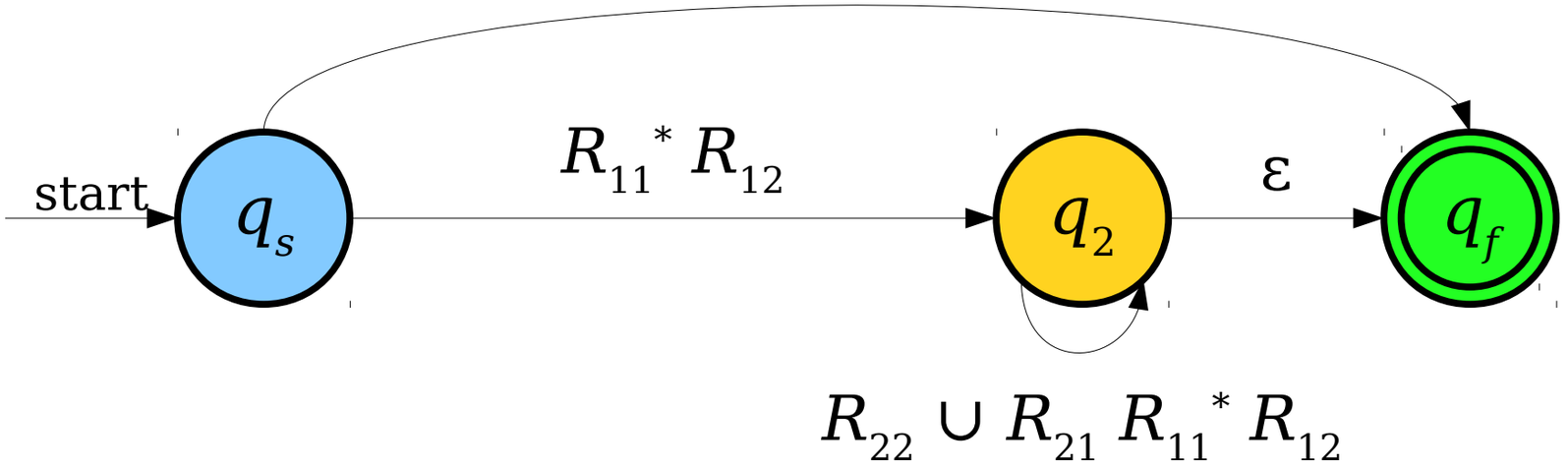
From NFAs to Regular Expressions



From NFAs to Regular Expressions

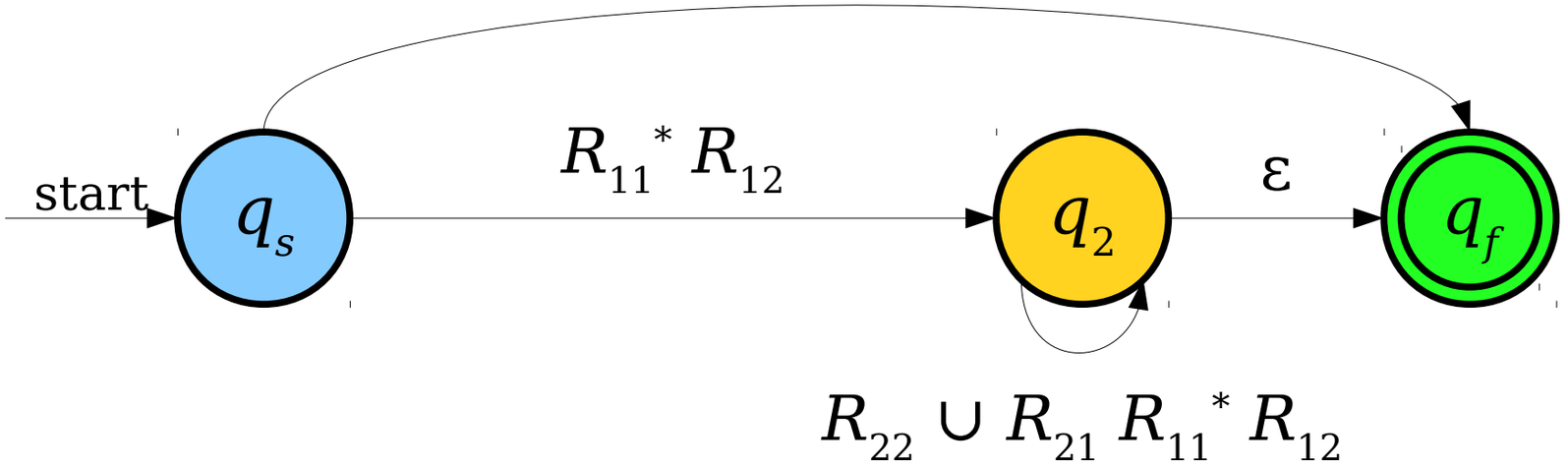


From NFAs to Regular Expressions

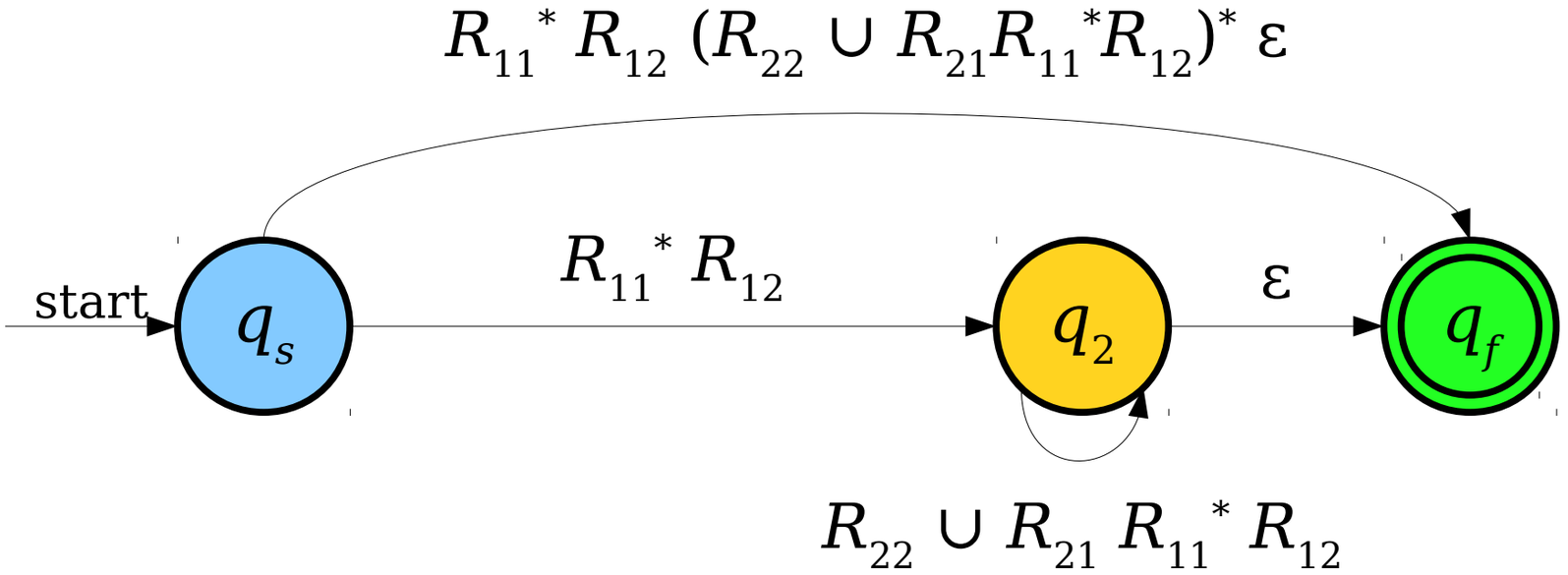


From NFAs to Regular Expressions

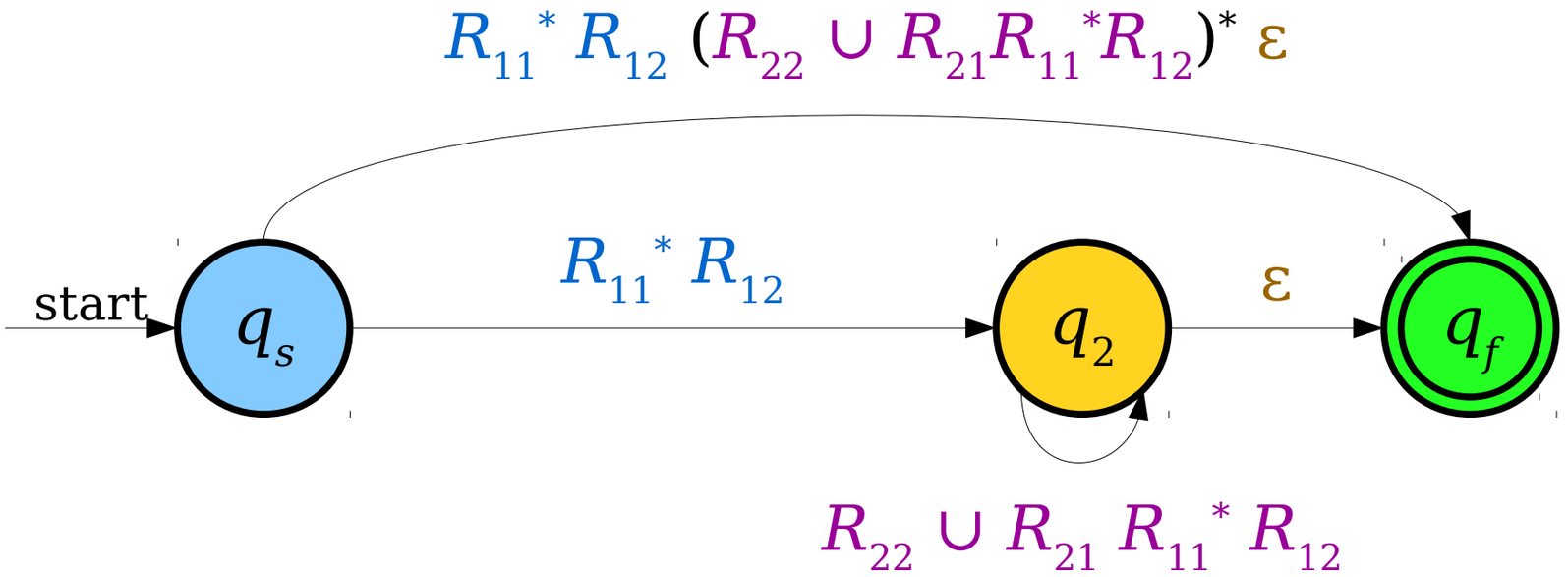
Quick check: what goes on this transition?



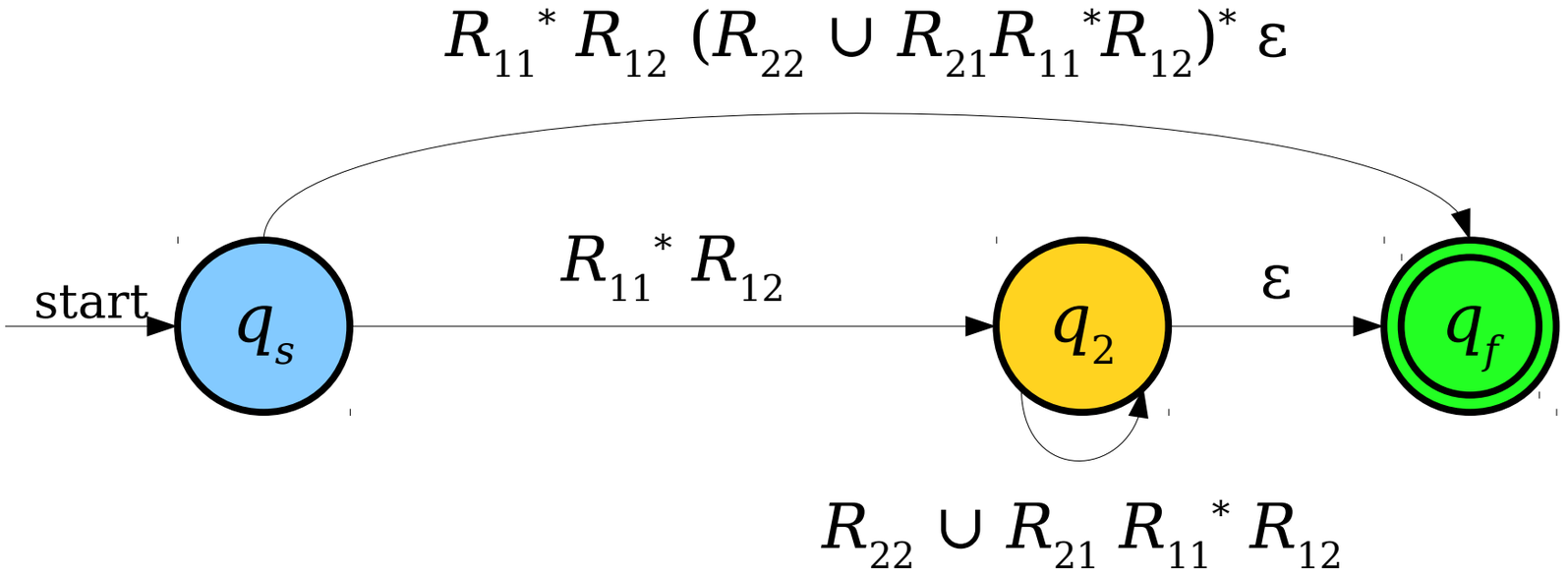
From NFAs to Regular Expressions



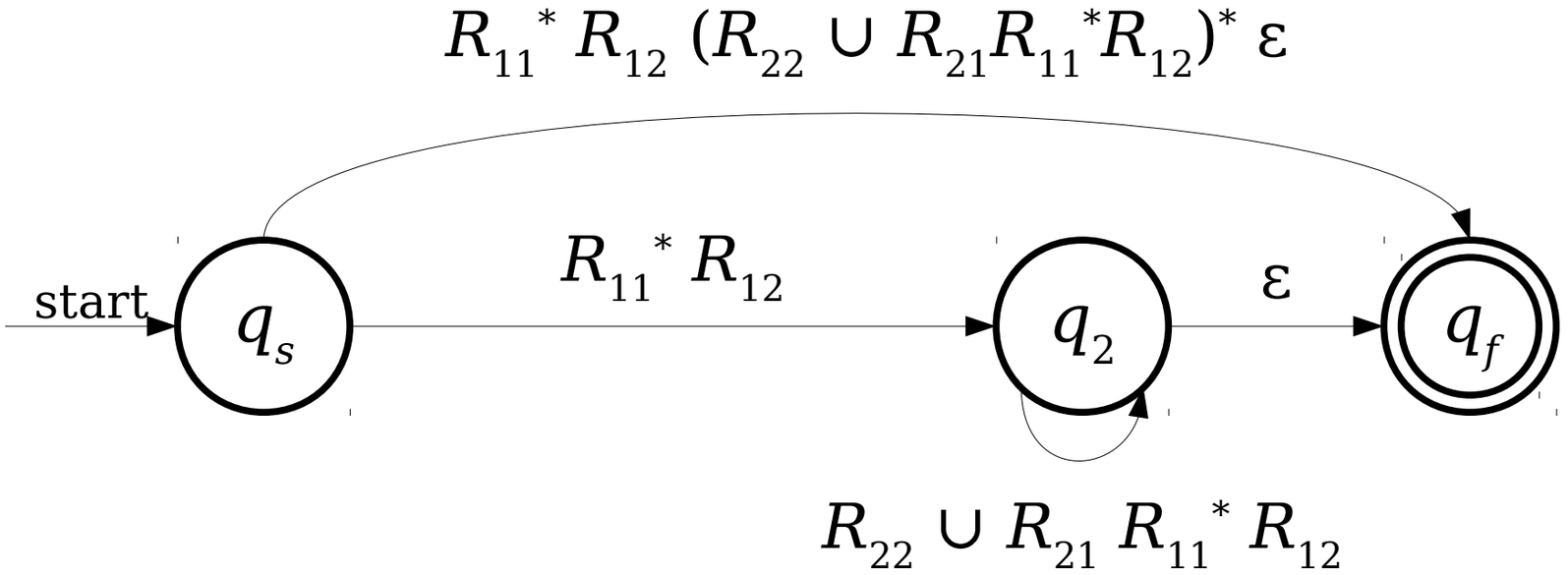
From NFAs to Regular Expressions



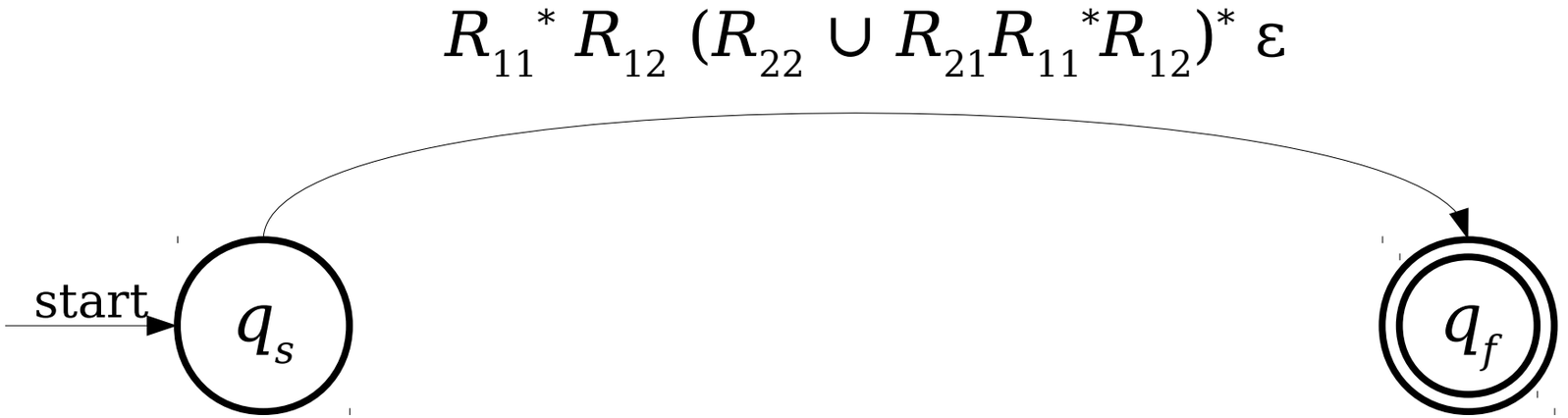
From NFAs to Regular Expressions



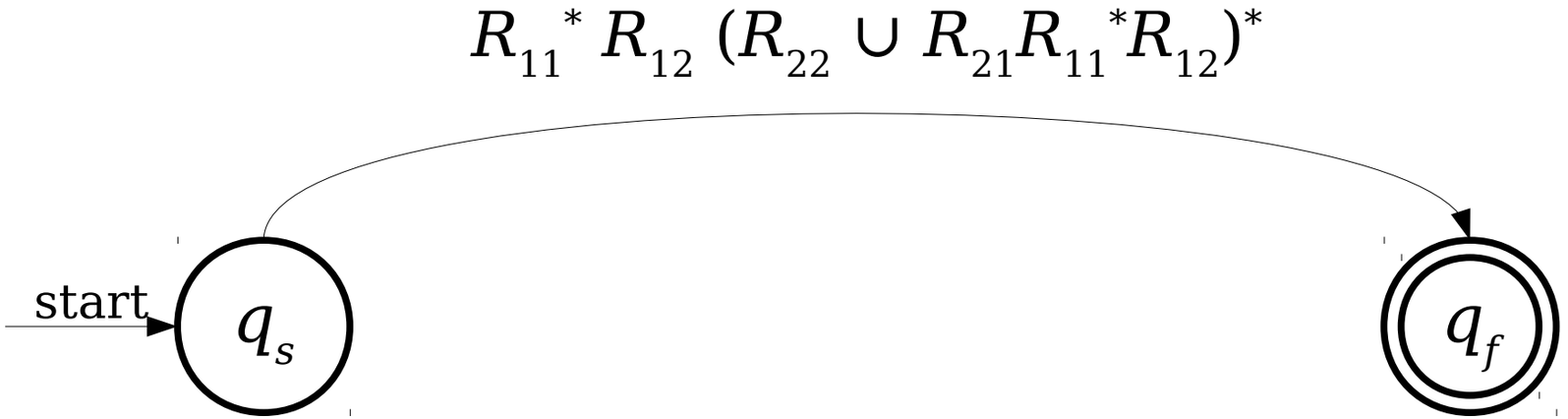
From NFAs to Regular Expressions



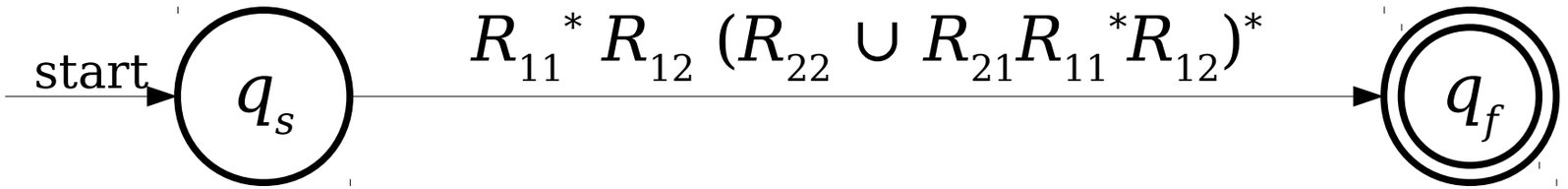
From NFAs to Regular Expressions



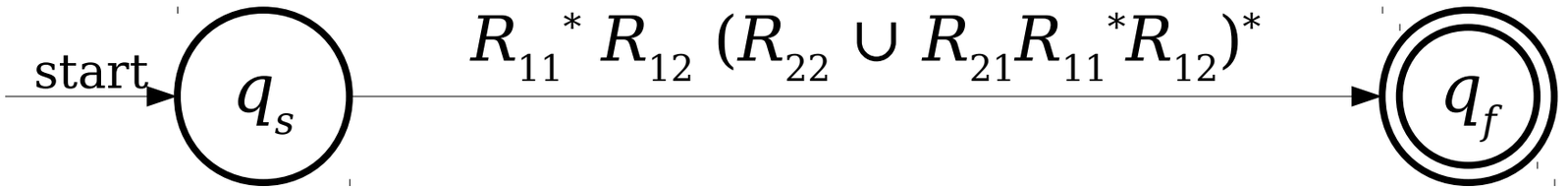
From NFAs to Regular Expressions

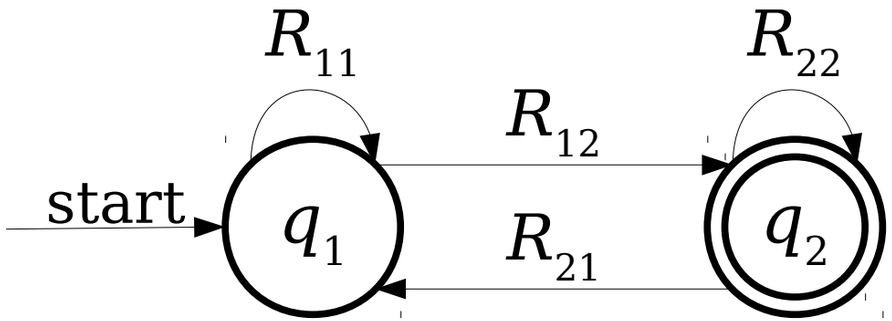


From NFAs to Regular Expressions



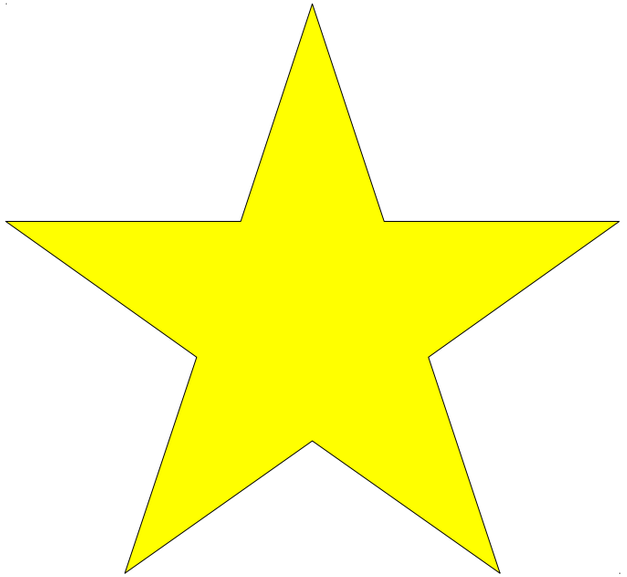
From NFAs to Regular Expressions



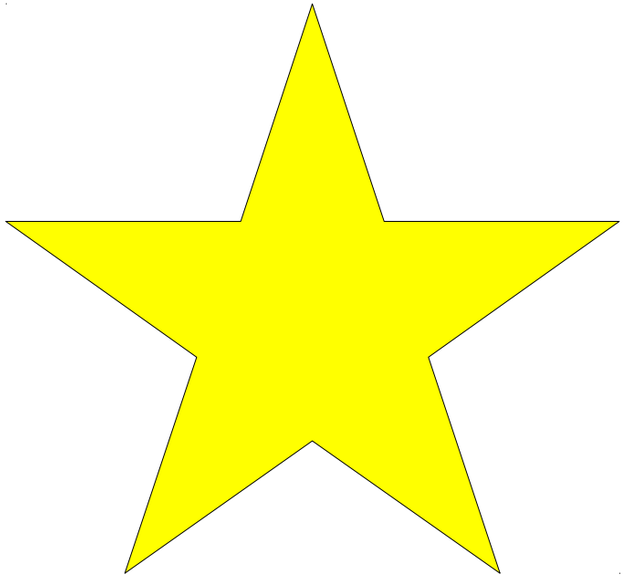


The State-Elimination Algorithm

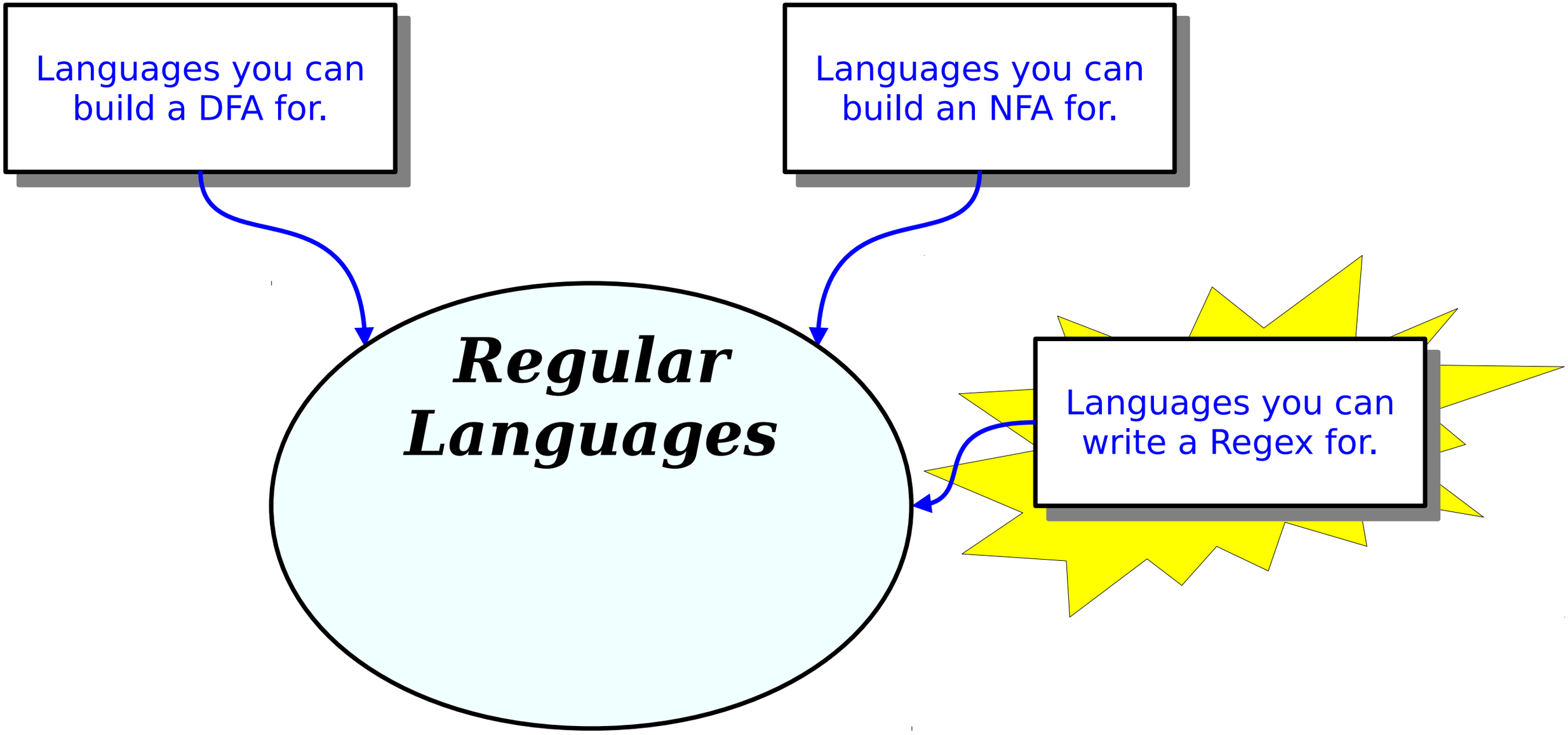
- Start with an NFA N for the language L.
- Add a new start state q_s and accept state q_f to the NFA.
  - Add an ε-transition from q_s to the old start state of N.
  - Add ε-transitions from each accepting state of N to q_f, then mark them as not accepting.
- Repeatedly remove states other than q_s 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.



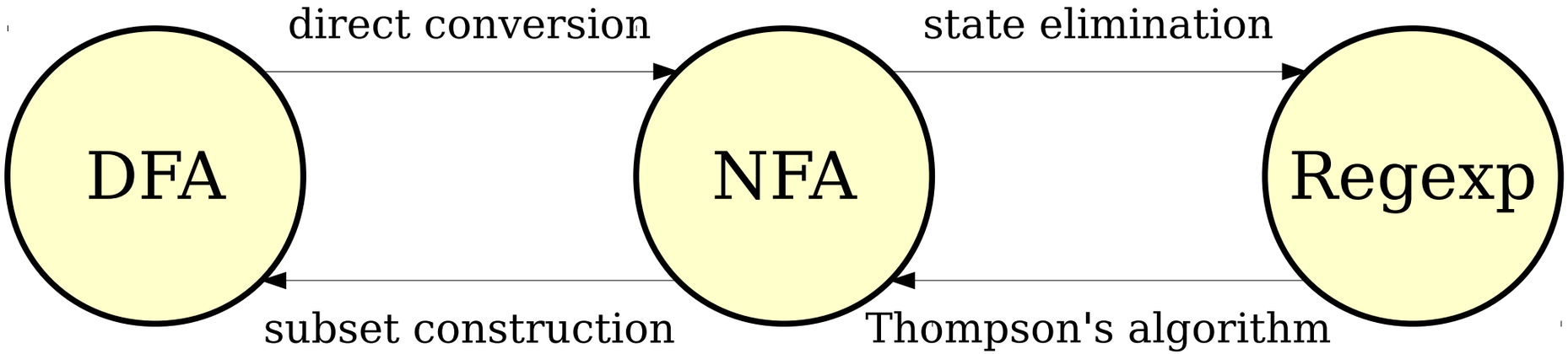
The State-Elimination Algorithm



- 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 $R_1 \cup R_2 \cup \ldots \cup R_k$.


Languages you can build a DFA for.
Languages you can build an NFA for.
Regular Languages
Languages you can write a Regex for.

Our Transformations



Theorem: The following are all equivalent:

- L is a regular language.
- There is a DFA D such that $\mathscr{L}(D) = L$.
- There is an NFA N such that $\mathscr{L}(N) = L$.
- There is a regular expression R such that $\mathscr{L}(R) = L$.

Next Time

- ***On Beyond Regular!***
  - What kinds of languages *aren't* regular?
- ***The Myhill-Nerode Theorem***
  - Proof technique to show non-regular