

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

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

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

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

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 .

$\{ \text{aaaaaaa}, \text{aaaab}, \text{aabaa}, \text{aabb}, \text{baaaa}, \text{baab}, \text{bbaa}, \text{bbb} \}$

- $LLLL$ is the set of strings formed by concatenating quadruples of strings in L .

$\{ \text{aaaaaaaa}, \text{aaaaaab}, \text{aaaabaa}, \text{aaaabb}, \text{aabaaaa}, \text{aabbaab}, \text{aabbaa}, \text{aabbb}, \text{baaaaaa}, \text{baaaab}, \text{baabaa}, \text{baabb}, \text{bbaaaa}, \text{bbaab}, \text{bbbaa}, \text{bbbb} \}$

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:

$$w \in L^* \quad \text{iff} \quad \exists n \in \mathbb{N}. w \in L^n$$

- 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 $L = \{ \text{a}, \text{bb} \}$, then $L^* = \{$

$\epsilon,$

$\text{a}, \text{bb},$

$\text{aa}, \text{abb}, \text{bba}, \text{bbbb},$

$\text{aaa}, \text{aabb}, \text{abba}, \text{abbbb}, \text{bbaa}, \text{bbabb}, \text{bbbba}, \text{bbbbbb},$

\dots

$\}$

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 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.***

New Stuff!

Another View of Regular Languages

Rethinking Regular Languages

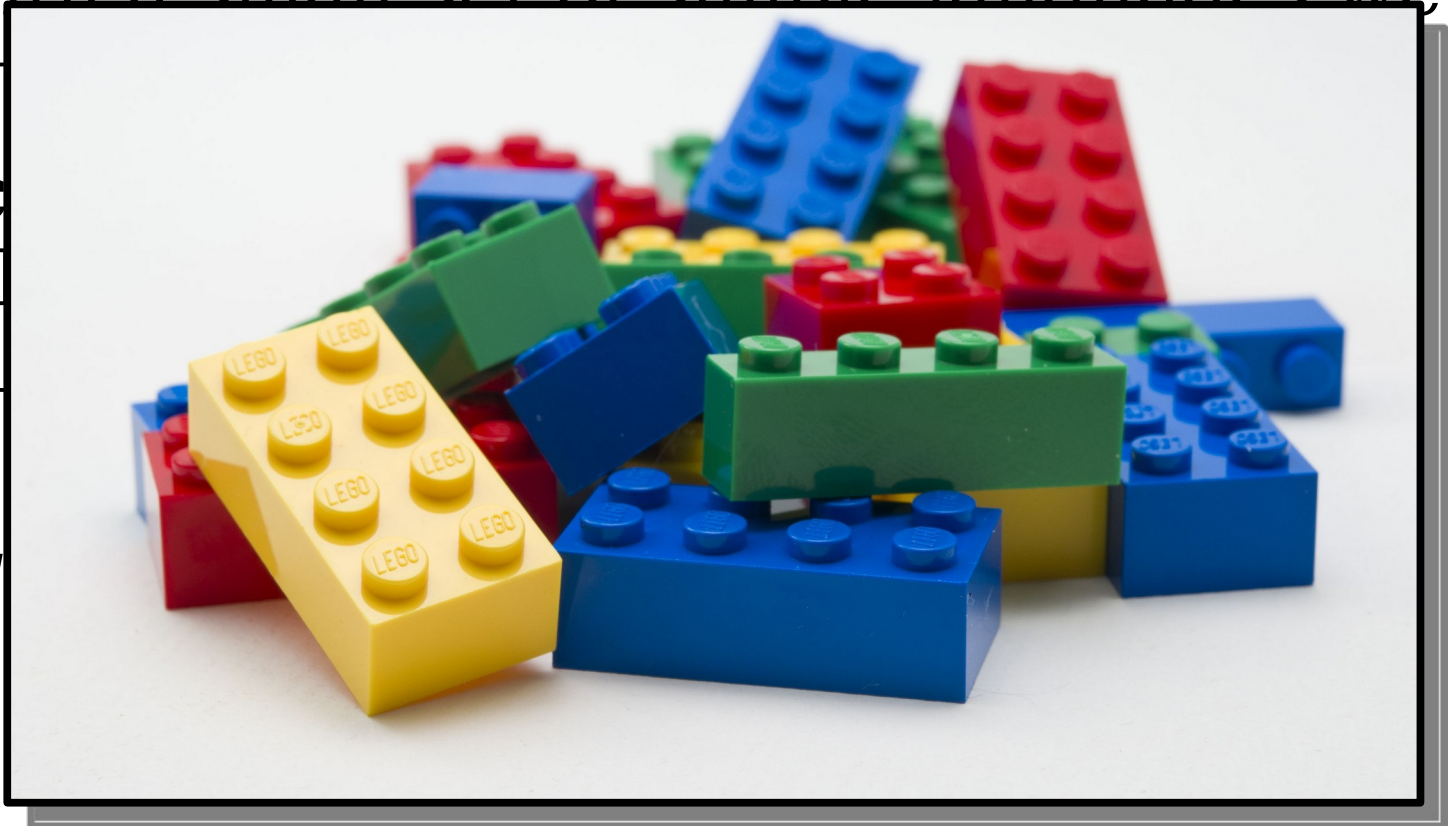
- We currently have several tools for showing a language L is regular:
 - Construct a DFA for L .
 - Construct an NFA for L .
 - Combine several simpler regular languages together via closure properties to form L .
- We have not spoken much of this last idea.

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
 - Using closure properties of simple languages to elaborate
- *This is a regular language*



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 ϵ is a regular expression that represents the language $\{\epsilon\}$.
 - **Remember:** $\{\epsilon\} \neq \emptyset!$
 - **Remember:** $\{\epsilon\} \neq \epsilon!$

Compound Regular Expressions

- If R_1 and R_2 are regular expressions, $\mathbf{R_1R_2}$ 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, $\mathbf{R_1 \cup R_2}$ is a regular expression for the *union* of the languages of R_1 and R_2 .
- If R is a regular expression, $\mathbf{R^*}$ is a regular expression for the *Kleene closure* of the language of R .
- If R is a regular expression, $\mathbf{(R)}$ is a regular expression with the same meaning as R .

Operator Precedence

- Here's the operator precedence for regular expressions:

(R)

R^*

R_1R_2

$R_1 \cup R_2$

- So **ab*cUd** is parsed as **((a(b*))c)Ud**

Regular Expression Examples

- The regular expression **trickUtrear** represents the language

{ **trick**, **trear** }.

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

Worthwhile activity: Apply this recursive definition to

$a(b \cup c)((d))$

and see what you get.

Designing Regular Expressions

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

Designing Regular Expressions

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

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

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

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

bbabbbaabab
aaaa
bbbbbabbbbaabbbbb

Designing Regular Expressions

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

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

bbabbb**aa**bab

aaaa

bbbbbbabbbb**aa**bbbbbb

Designing Regular Expressions

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

$\Sigma^*aa\Sigma^*$

bbabbb**aa**bab

aaaa

bbbbbbabbbb**aa**bbbbbb

Designing Regular Expressions

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

Designing Regular Expressions

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

$\Sigma\Sigma\Sigma\Sigma$

Designing Regular Expressions

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

$\Sigma\Sigma\Sigma\Sigma$

Designing Regular Expressions

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

$\Sigma\Sigma\Sigma\Sigma$

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

$\Sigma\Sigma\Sigma\Sigma$

$\mathbf{a}\mathbf{a}\mathbf{a}\mathbf{a}$

$\mathbf{b}\mathbf{a}\mathbf{b}\mathbf{a}$

$\mathbf{b}\mathbf{b}\mathbf{b}\mathbf{b}$

$\mathbf{b}\mathbf{a}\mathbf{a}\mathbf{a}$

Designing Regular Expressions

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

Σ^4

aaaa
baba
bbbb
baaa

Designing Regular Expressions

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

Σ^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?

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

Designing Regular Expressions

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

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

Designing Regular Expressions

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

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

Designing Regular Expressions

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

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

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

bbbbbb

abbb

a

Designing Regular Expressions

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

$b^*a?b^*$

bbbbabbb

bbbbbb

abbb

a

A More Elaborate Design

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

A More Elaborate Design

- Let $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 **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 **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 **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 **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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 $\Sigma = \{ \text{a}, ., @ \}$, 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 **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 **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 **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 **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

A More Elaborate Design

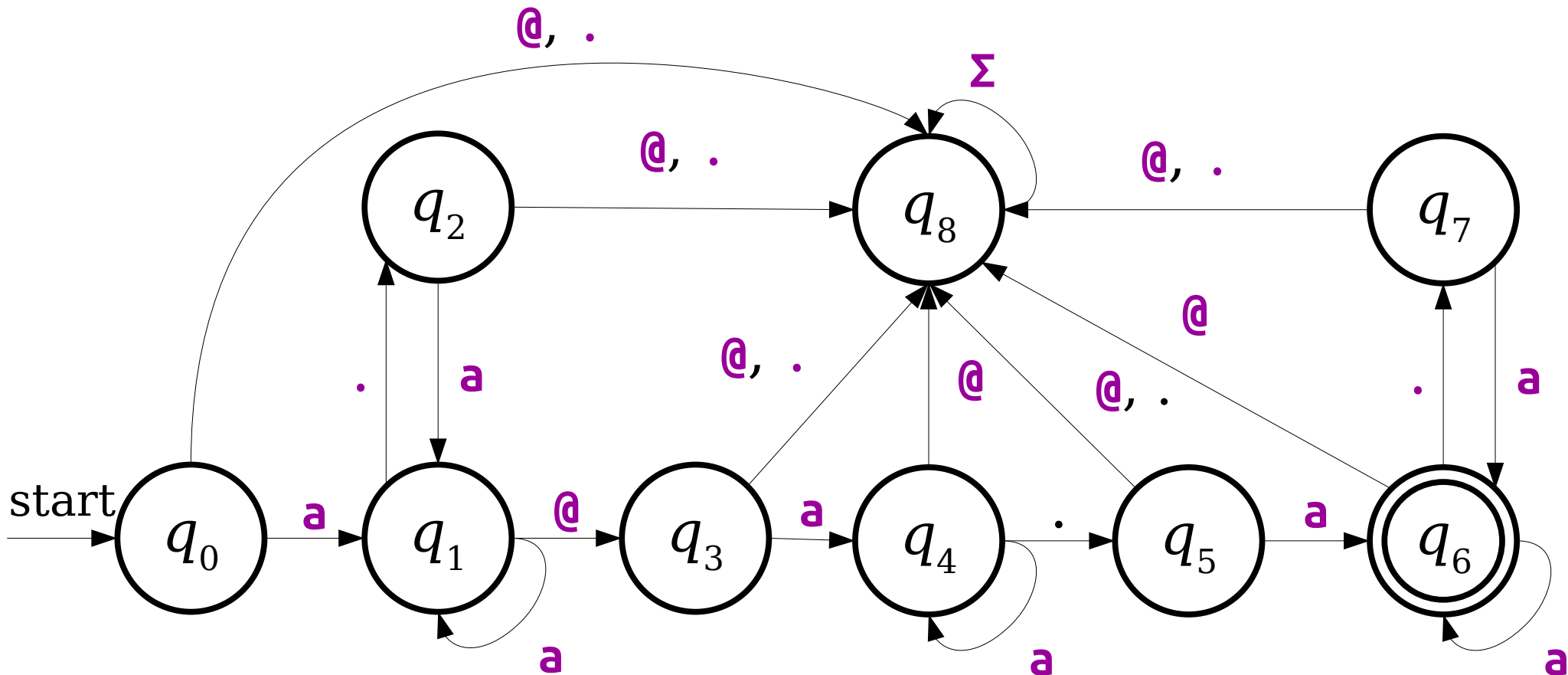
- Let $\Sigma = \{ \text{a}, ., @ \}$, where **a** represents “some letter.”
- Let's make a regex for email addresses.

a⁺ (**.****a**⁺)^{*} **@** **a**⁺ (**.****a**⁺)⁺

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

For Comparison

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



Shorthand Summary

- R^n is shorthand for $RR \dots R$ (n times).
 - Edge case: define $R^0 = \varepsilon$.
- Σ is shorthand for “any character in Σ .”
- $R?$ 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!

Midterm Exam Logistics

- Our next Midterm runs this Friday, November 5th at 2:30PM through this Sunday, November 7th at 2:30PM, Pacific time.
 - That's 49 hours rather than the normal 48. Huzzah!
- Topic coverage is primarily lectures 06 – 13 (functions through induction) and PS3 – PS5. Finite automata and onward won't be tested here.
 - Because the material is cumulative, topics from PS1 – PS2 and Lectures 00 – 05 are also fair game.
- Extra Practice Problems 2 is available on the course website if you want to get more practice with these topics.
- ***We want you to do well on this exam.*** Keep in touch and let us know what we can do to help make that happen!

Your Questions

“I know that one assignment, one test, or one class doesn't define my ability to succeed in CS, but is there a point that I really should consider another career path if I still can't do well? If so, where?”

If you're really struggling with something, the first question to ask is “what am I doing, and why isn't it working?” That can be hard to ask because it requires you to introspect on what your approach is, what parts are working, and what aren't. And often that requires chatting with someone more experienced to get input on what you're doing.

My recommendation would be to start with this and to size up where you are and what energy would be required to change things. That will give you a roadmap of what you need to do. From there, you can evaluate whether what you're doing is worth the time investment. If it is, great! Put in the time, be circumspect, evaluate as you go, and you'll be in good shape. If not, there's your answer. But I would avoid “jumping ship” until you have a clear answer to this question, since it would be a real shame if you needed to only tune or tweak a few things to get on a great track and you ended up walking away.

Back to CS103!

The Lay of the Land

Languages you can
build a DFA for.

Languages you can
build an NFA for.

***Regular
Languages***

```
graph TD; A[Languages you can build a DFA for.] --> C((Regular Languages)); B[Languages you can build an NFA for.] --> C;
```

The diagram illustrates the relationship between different types of languages and Regular Languages. At the top left, a box contains the text "Languages you can build a DFA for.". At the top right, a box contains the text "Languages you can build an NFA for.". Both boxes have blue arrows pointing from their bottom centers to a large, light-blue oval in the center. The oval contains the text "Regular Languages" in a bold, italicized black font. This visualizes that both DFA-recognizable languages and NFA-recognizable languages are subsets of Regular Languages.

Languages you can
build a DFA for.

Languages you can
build an NFA for.

***Regular
Languages***

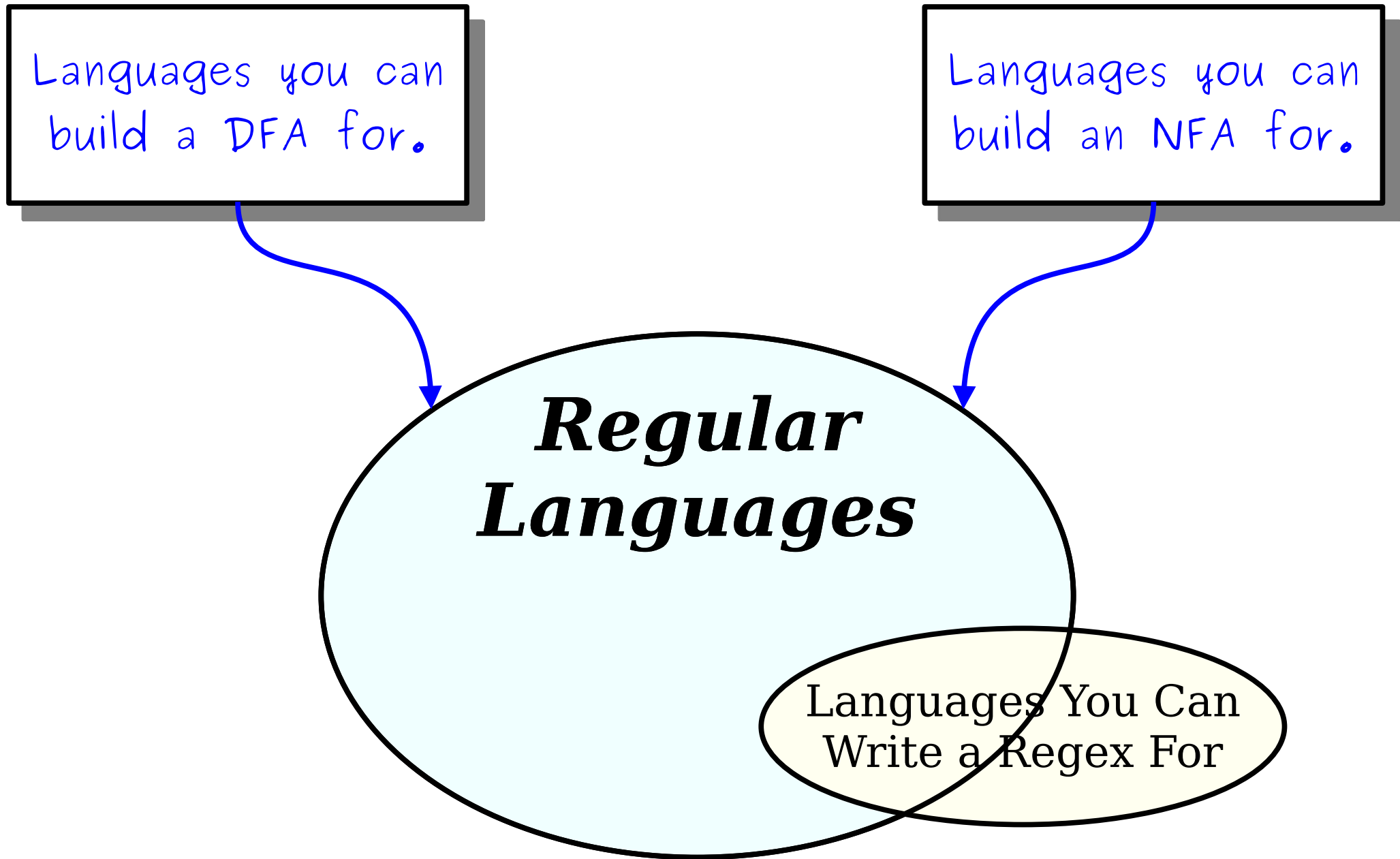
Languages You Can
Write a Regex For

Languages you can
build a DFA for.

Languages you can
build an NFA for.

***Regular
Languages***

Languages You Can
Write a Regex For

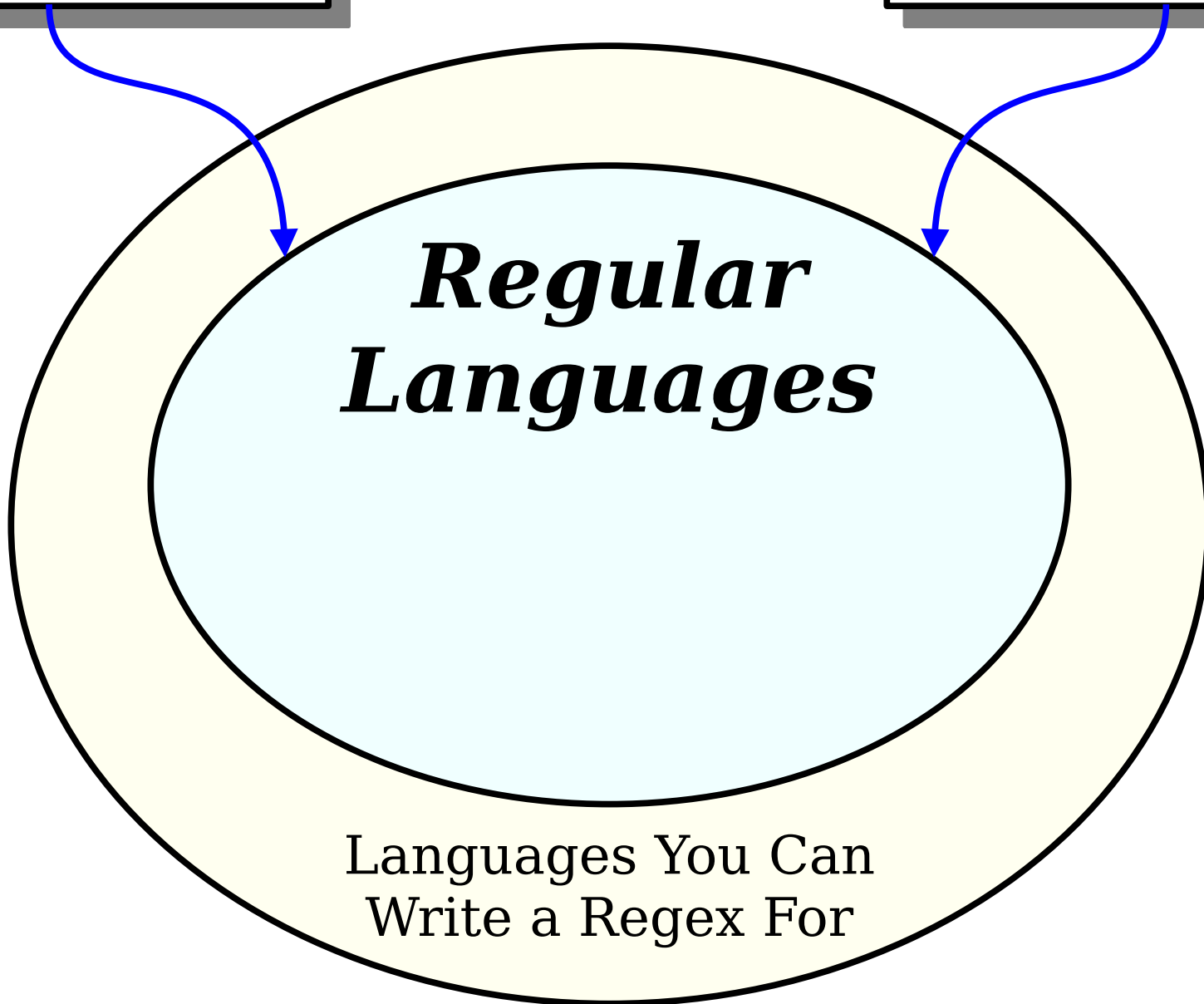


Languages you can
build a DFA for.

Languages you can
build an NFA for.

***Regular
Languages***

Languages You Can
Write a Regex For

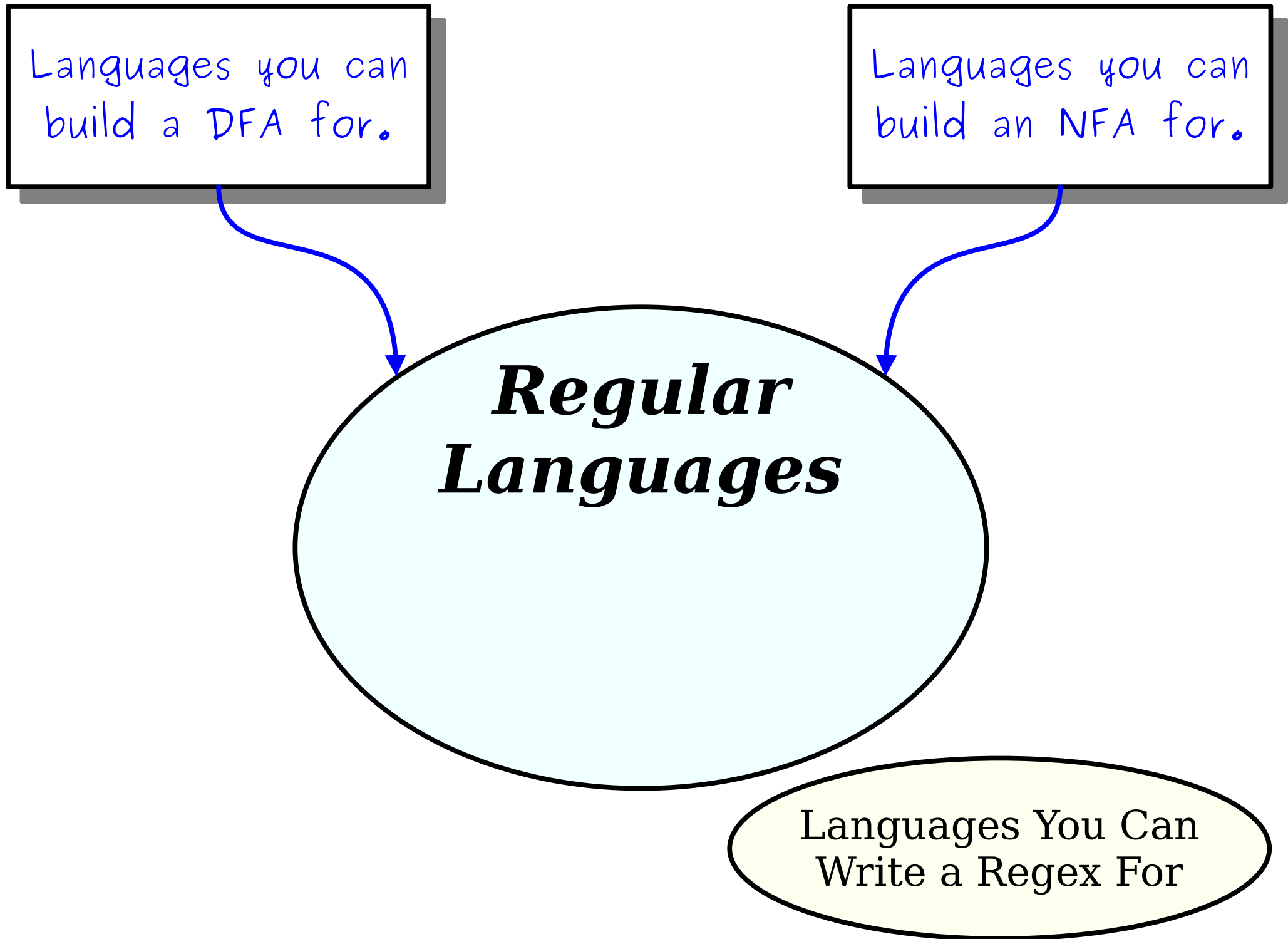


Languages you can
build a DFA for.

Languages you can
build an NFA for.

Regular Languages

Languages You Can
Write a Regex For

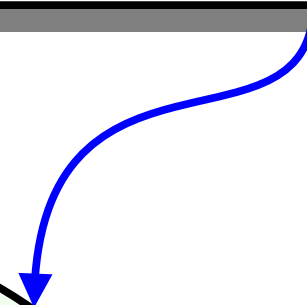
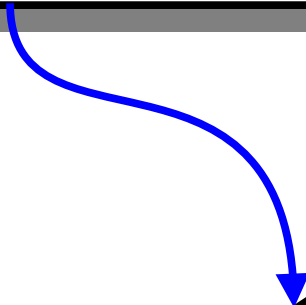


Languages you can
build a DFA for.

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.

***Regular
Languages***

```
graph TD; A[Languages you can build a DFA for.] --> C((Regular Languages)); B[Languages you can build an NFA for.] --> C;
```

The diagram consists of a central light blue oval with a black border containing the text ***Regular Languages***. Two rectangular boxes with black borders are positioned above the oval. The left box contains the text "Languages you can build a DFA for." and the right box contains "Languages you can build an NFA for.". Both boxes have a grey drop shadow. A blue curved arrow points from the bottom of the left box to the top-left edge of the oval. Another blue curved arrow points from the bottom of the right box to the top-right edge of the oval.

Languages you can
build a DFA for.

Languages you can
build an NFA for.

***Regular
Languages***

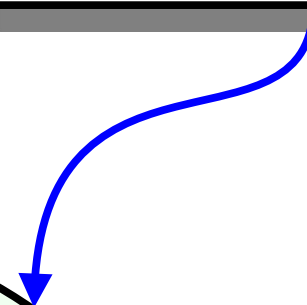
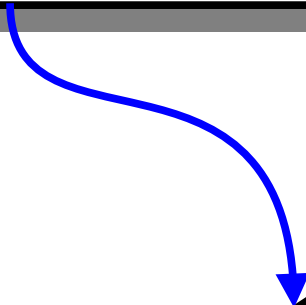
Languages You Can
Write a Regex For

Languages you can
build a DFA for.

Languages you can
build an NFA for.

Regular Languages

Languages You Can
Write a Regex For



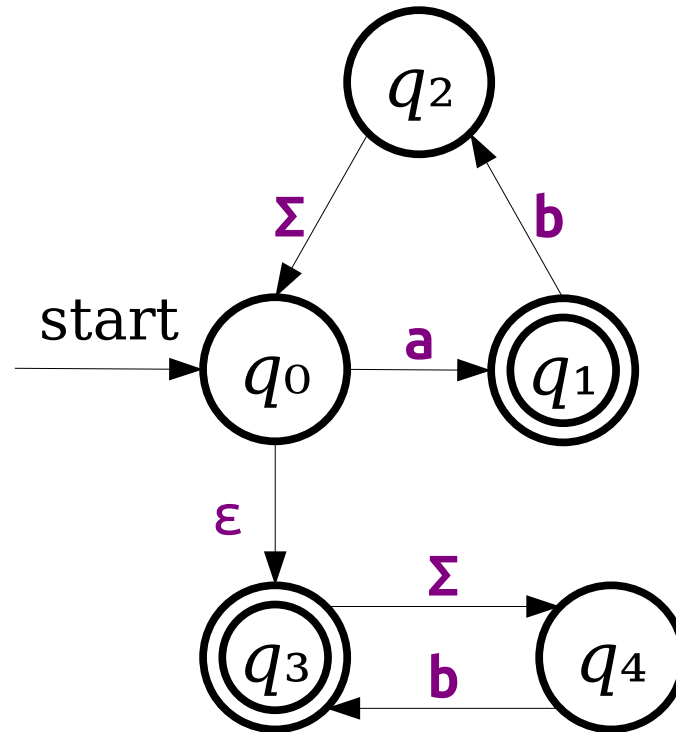
The Power of Regular Expressions

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

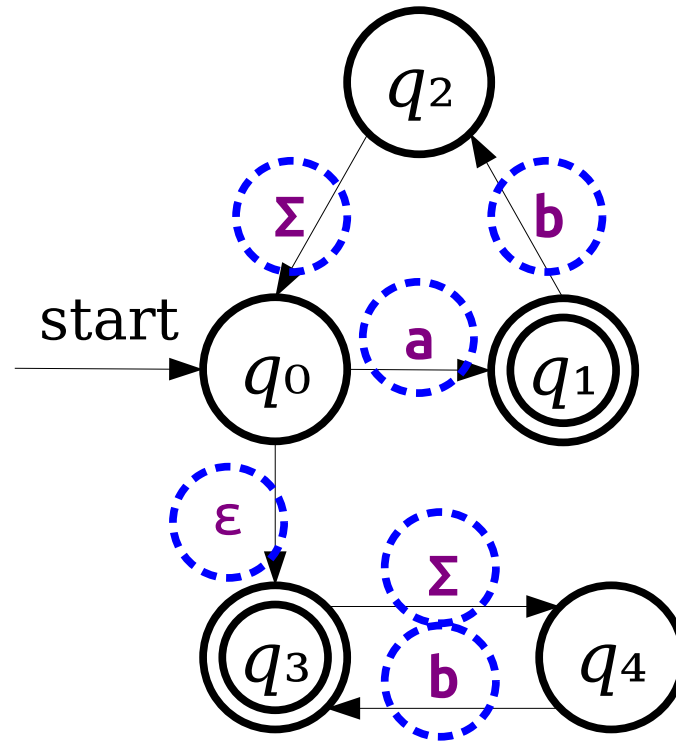
This is not obvious!

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

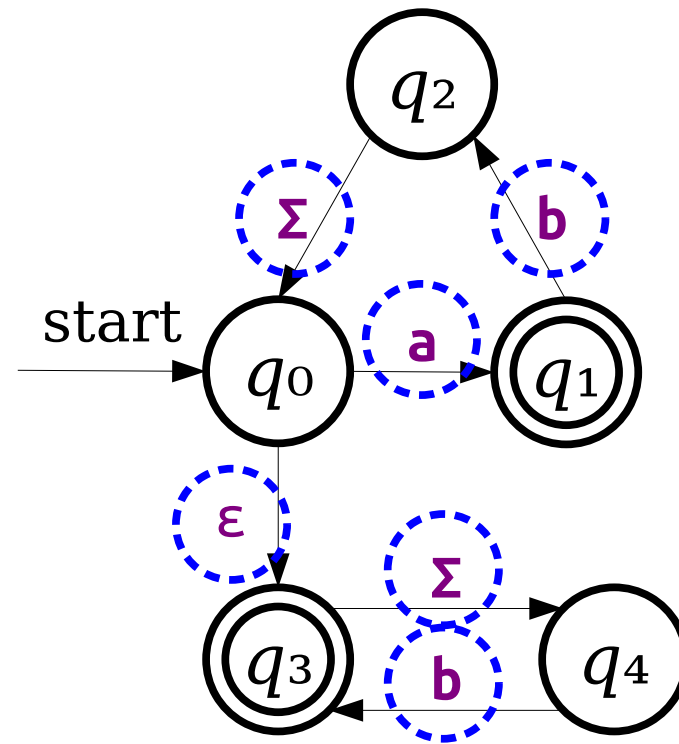
Generalizing NFAs



Generalizing NFAs

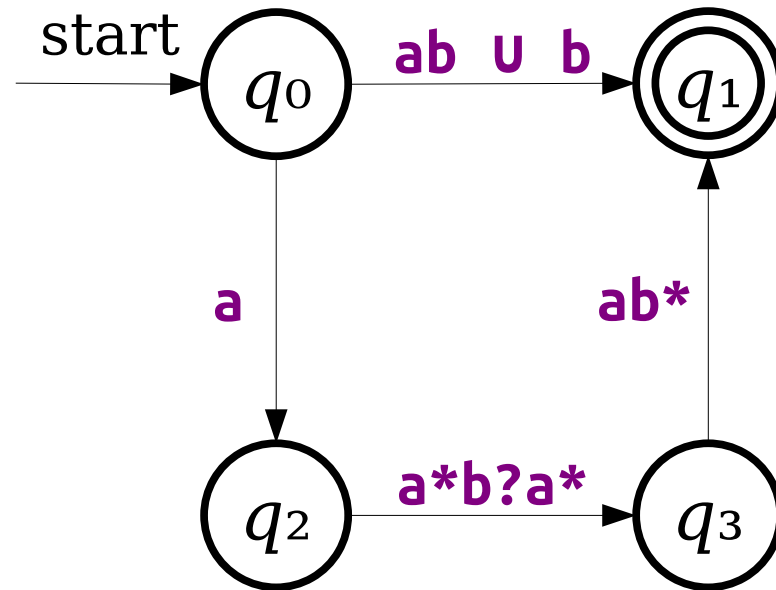


Generalizing NFAs

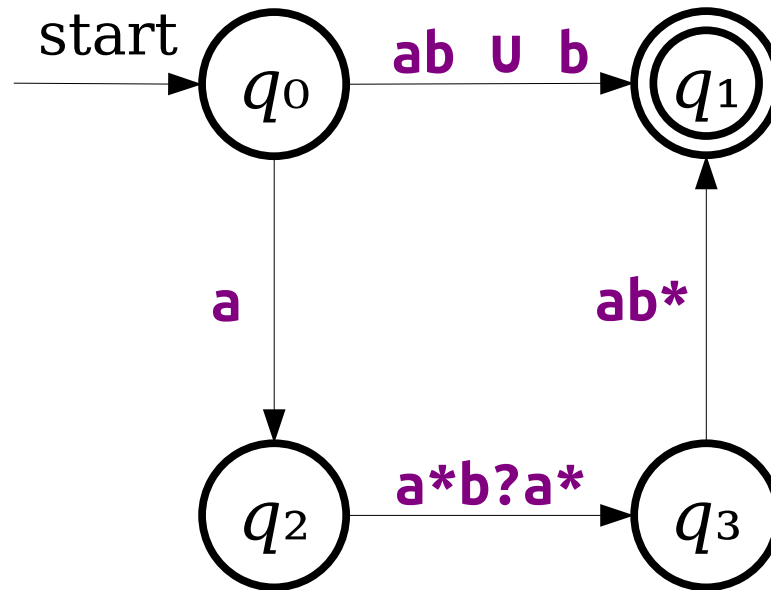


These are all regular expressions!

Generalizing NFAs

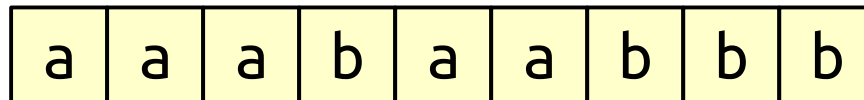
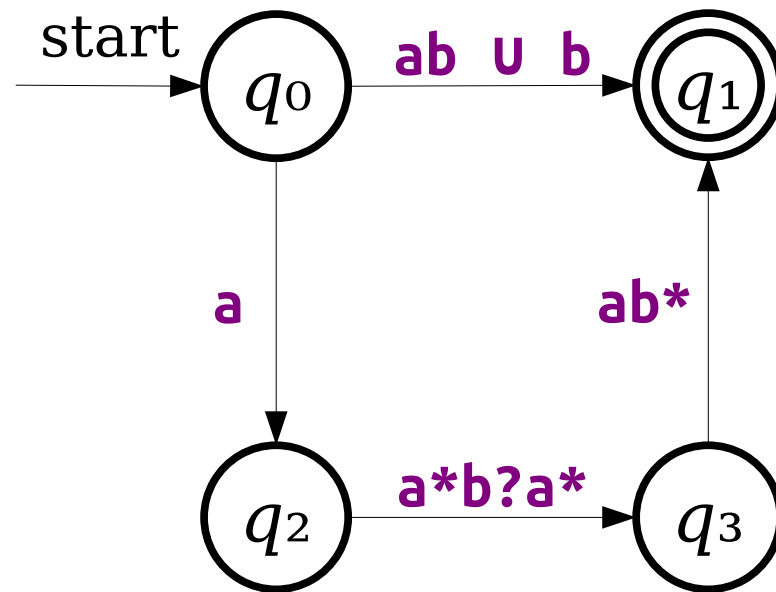


Generalizing NFAs

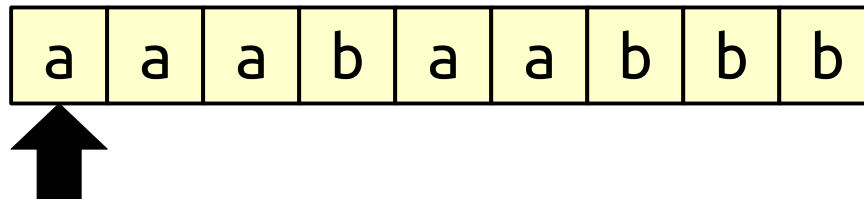
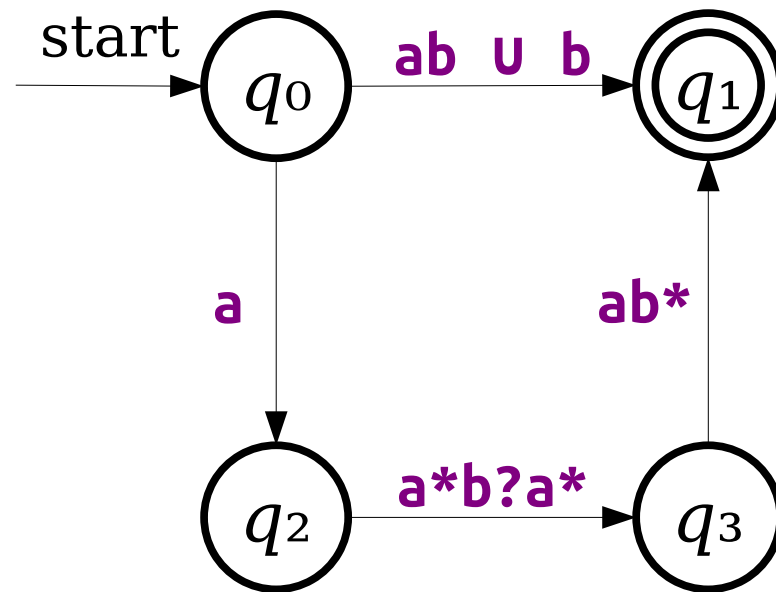


Note: Actual NFAs aren't allowed to have transitions like these. This is just a thought experiment.

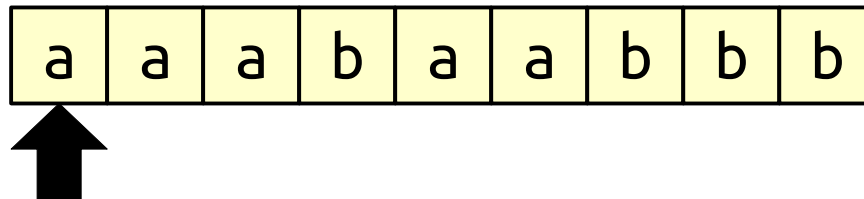
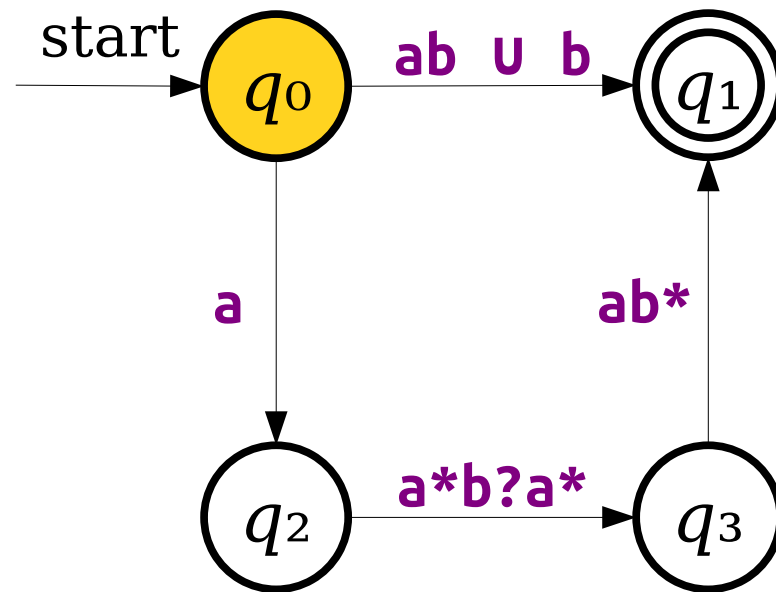
Generalizing NFAs



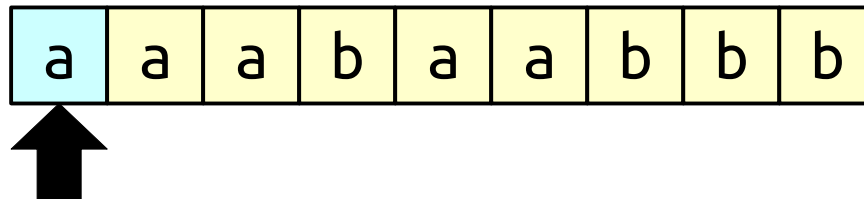
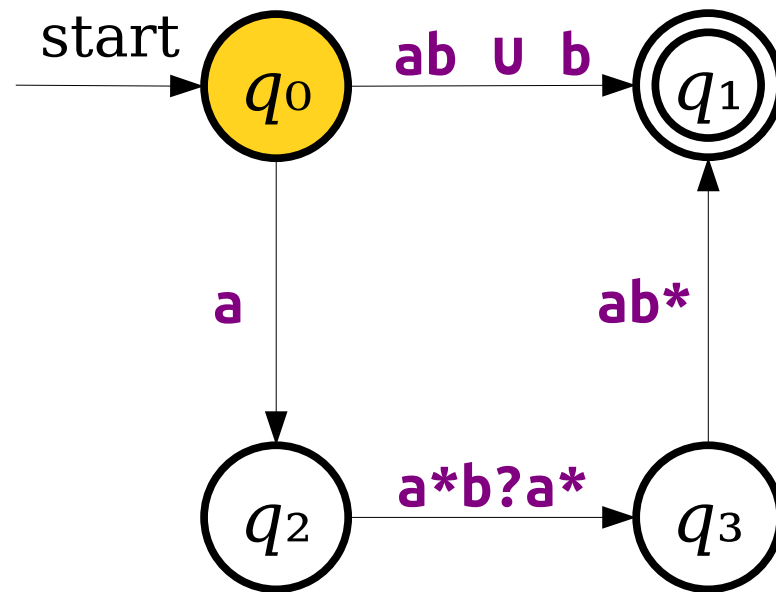
Generalizing NFAs



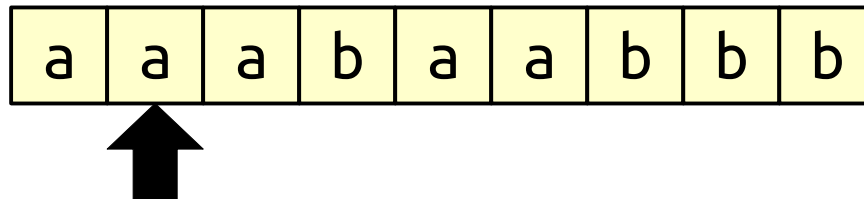
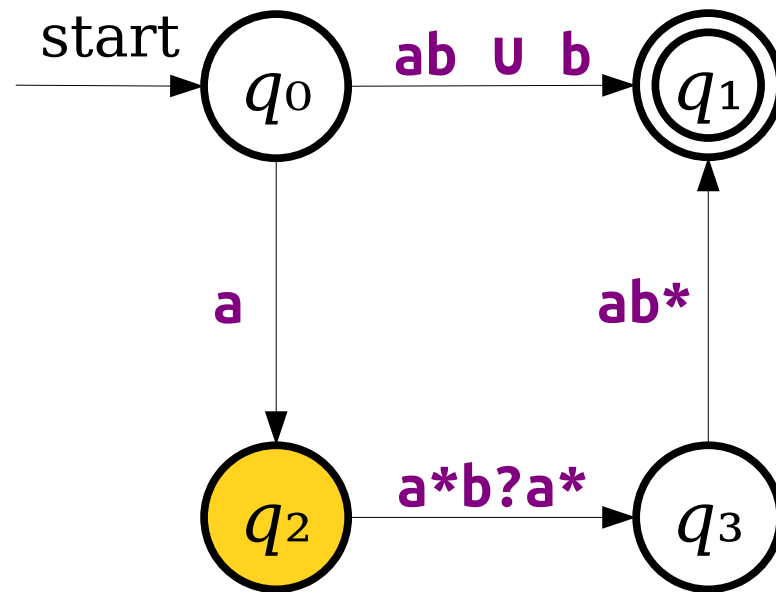
Generalizing NFAs



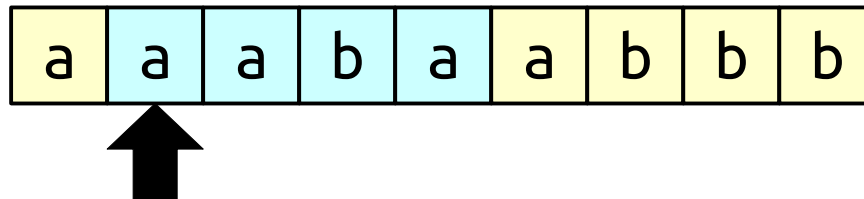
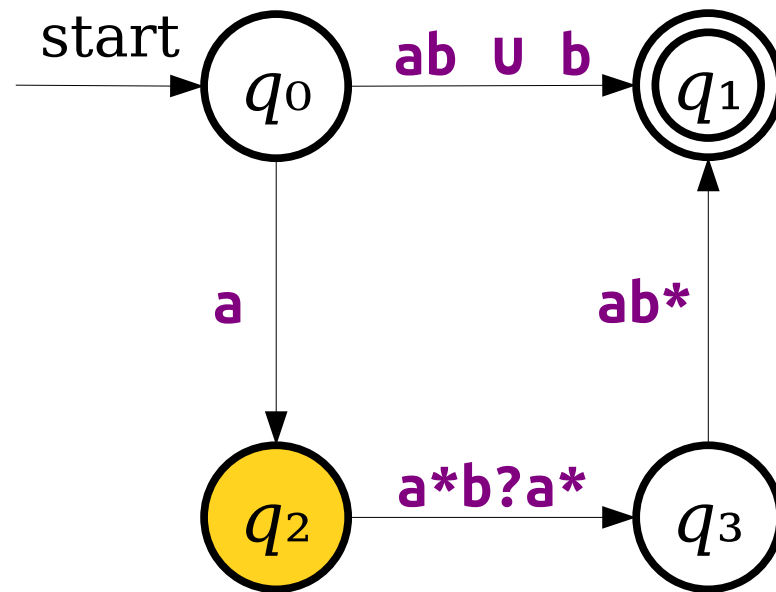
Generalizing NFAs



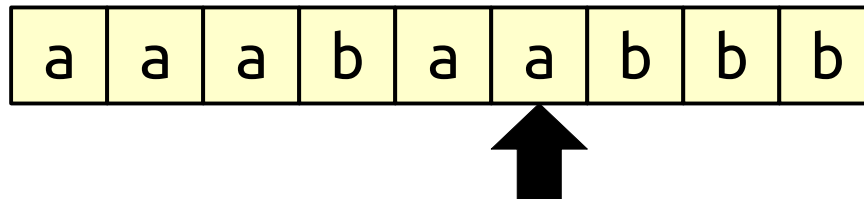
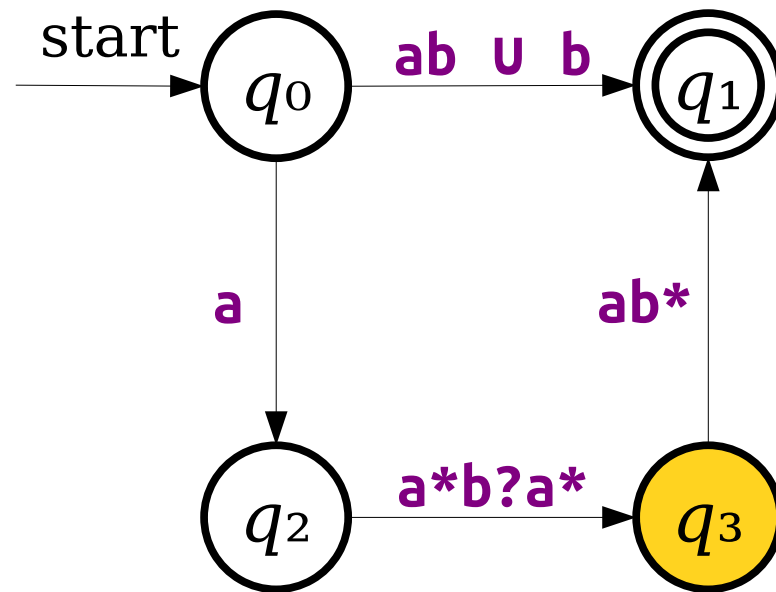
Generalizing NFAs



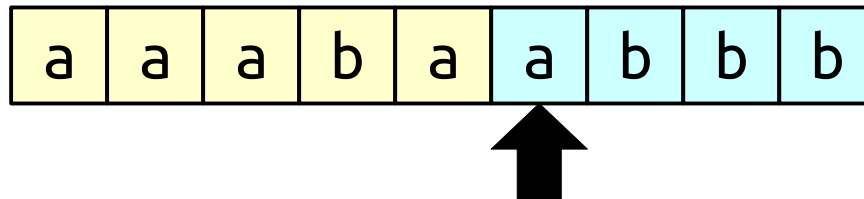
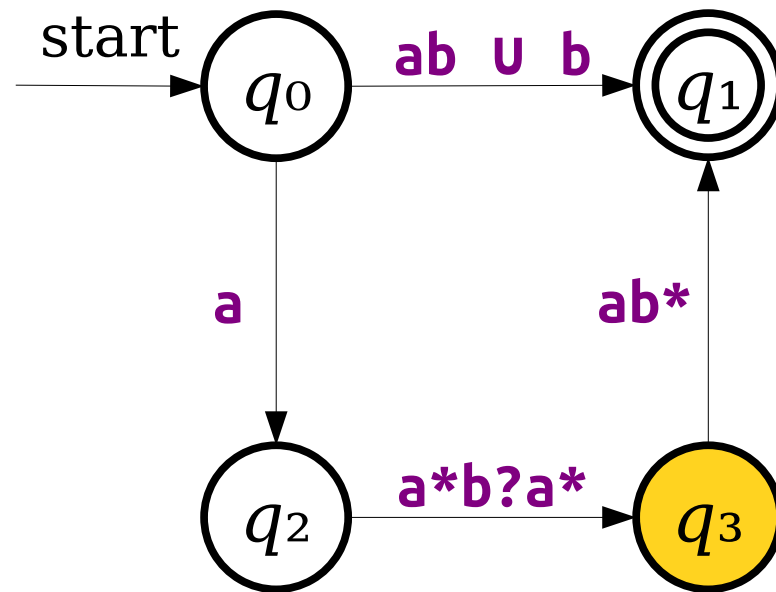
Generalizing NFAs



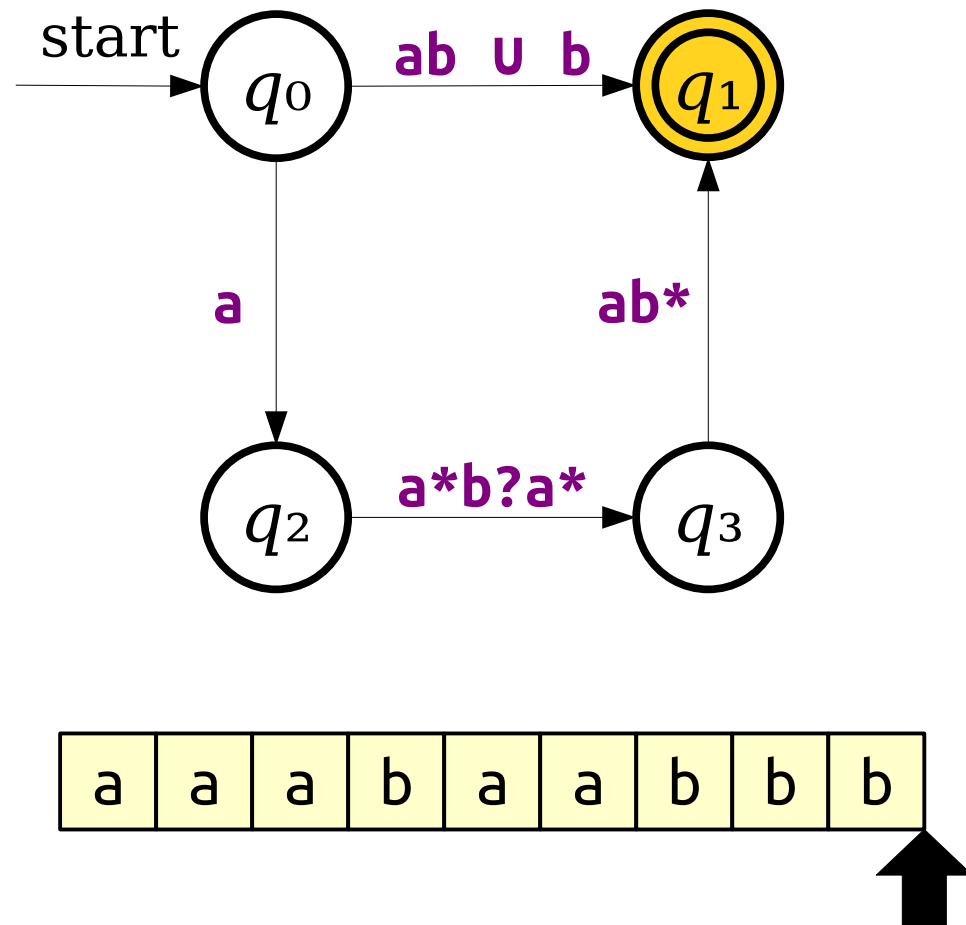
Generalizing NFAs



Generalizing NFAs



Generalizing NFAs



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

Generalizing NFAs

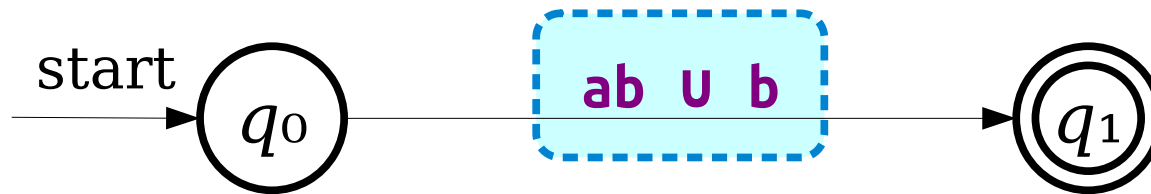


Generalizing NFAs



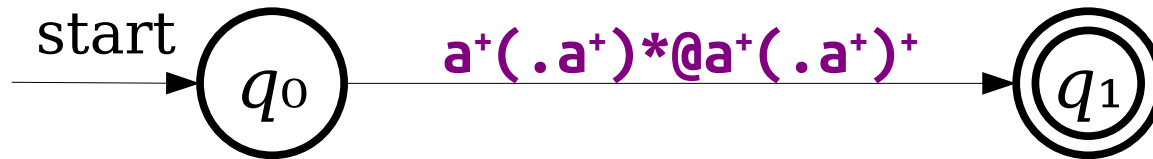
Is there a simple
regular expression for
the language of this
generalized NFA?

Generalizing NFAs

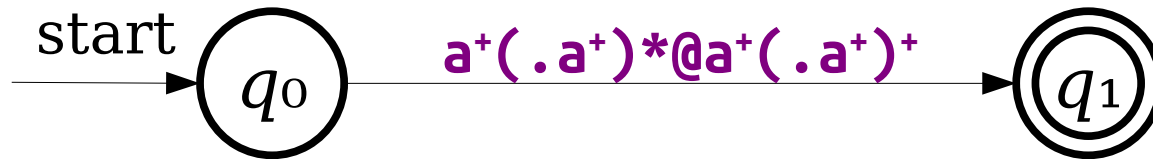


Is there a simple
regular expression for
the language of this
generalized NFA?

Generalizing NFAs

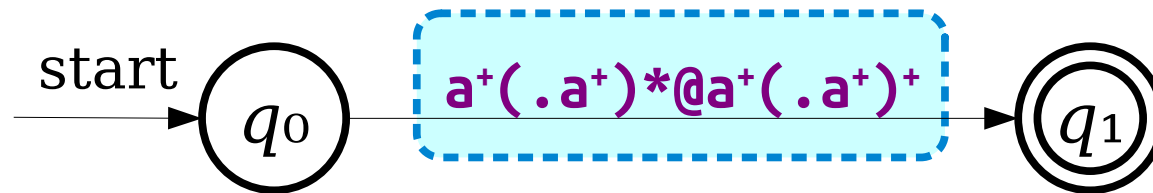


Generalizing NFAs



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

Generalizing NFAs



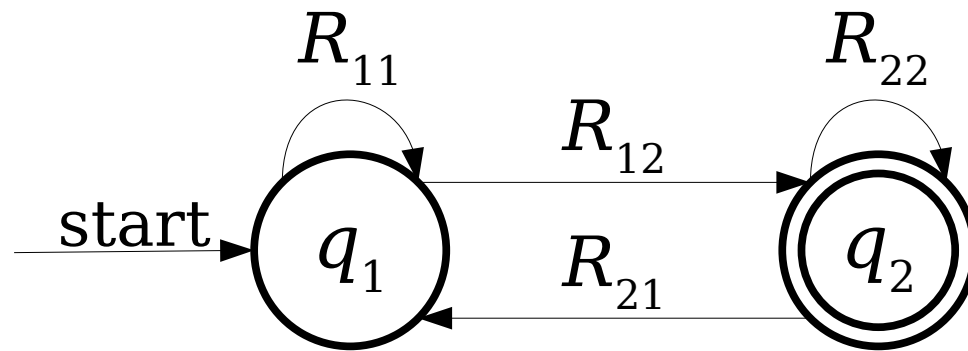
Is there a simple
regular expression for
the language of this
generalized NFA?

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

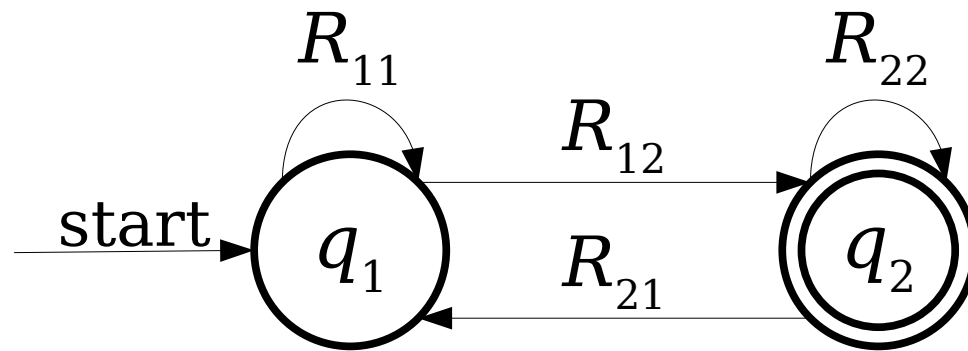


...then we can easily read off a regular expression for the original NFA.

From NFAs to Regular Expressions

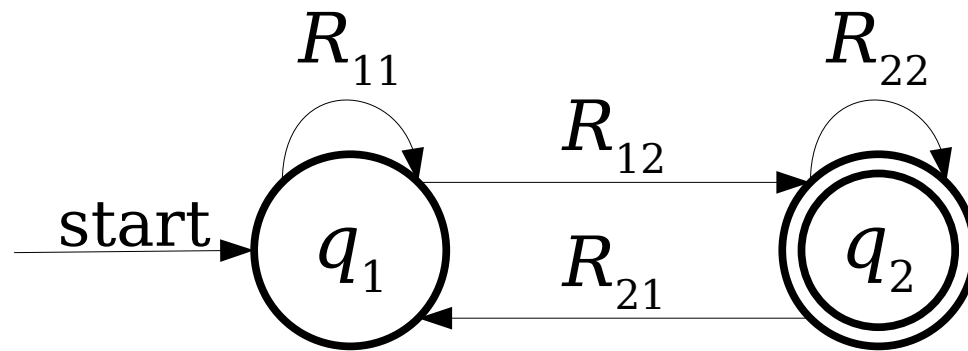


From NFAs to Regular Expressions



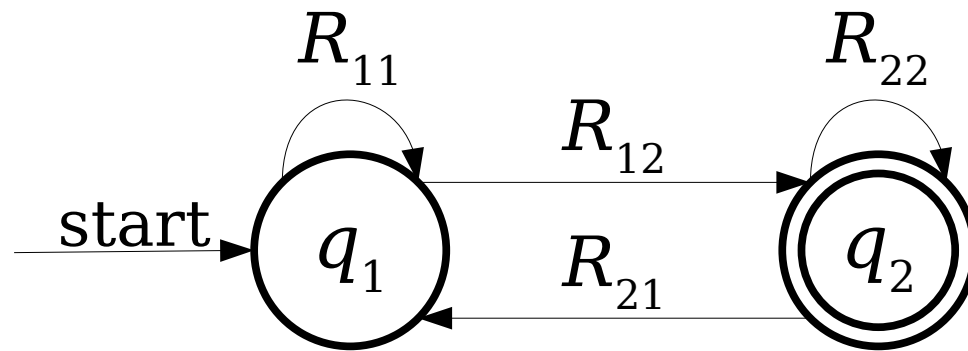
Here, R_{11} , R_{12} , R_{21} , and R_{22} are arbitrary regular expressions.

From NFAs to Regular Expressions

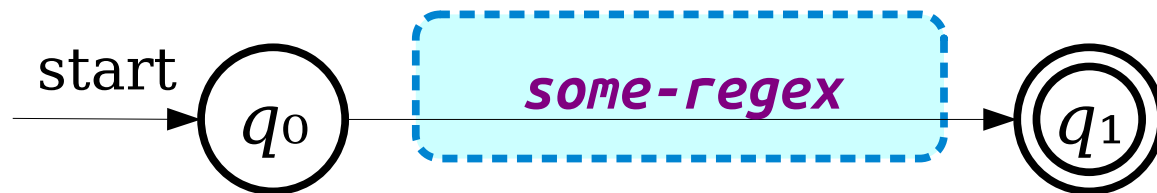


Question: Can we get a clean regular expression from this NFA?

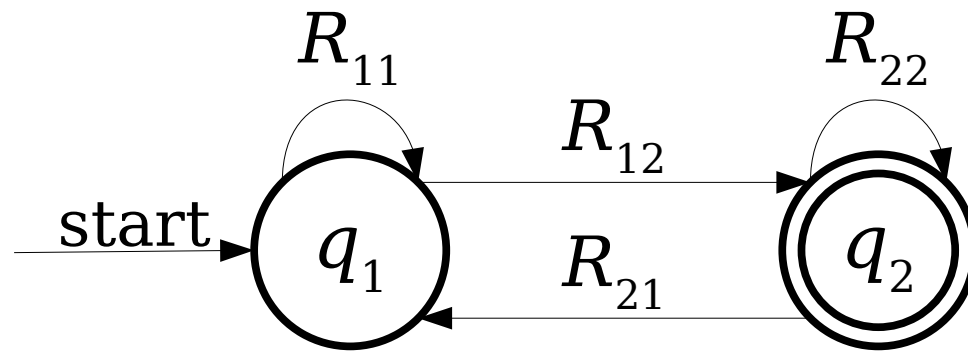
From NFAs to Regular Expressions



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

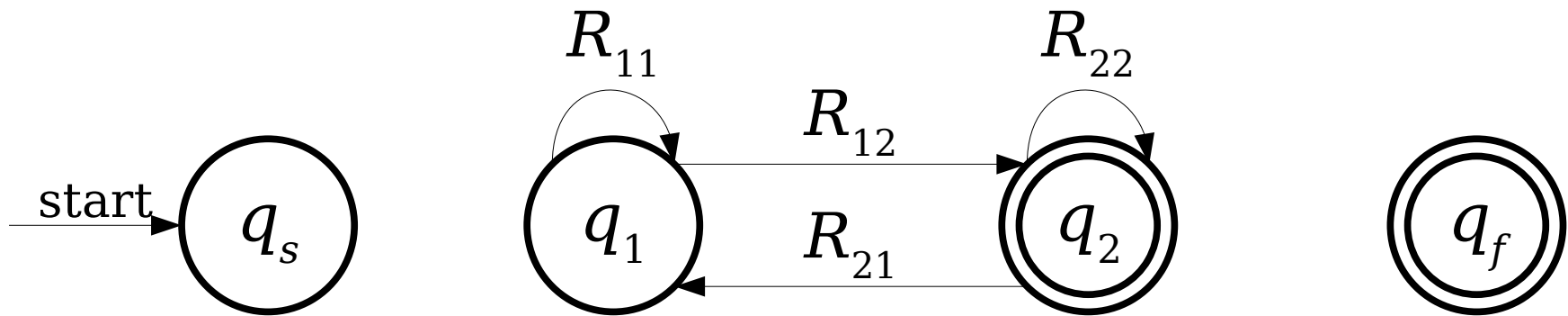


From NFAs to Regular Expressions

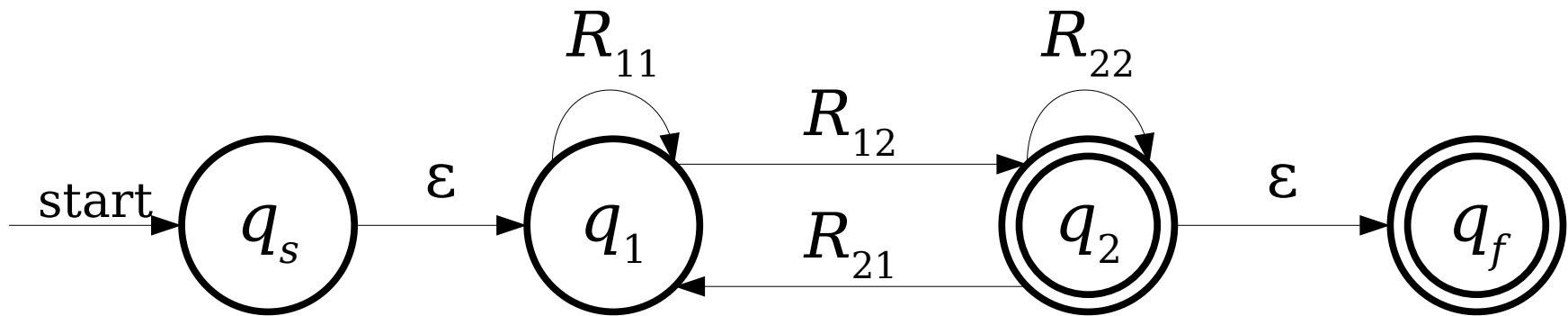


The first step is going to be a bit weird...

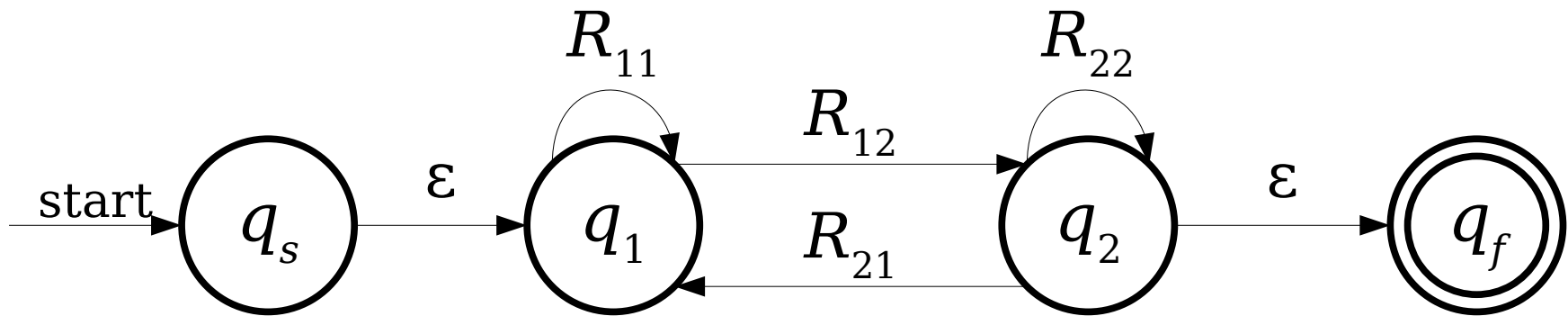
From NFAs to Regular Expressions



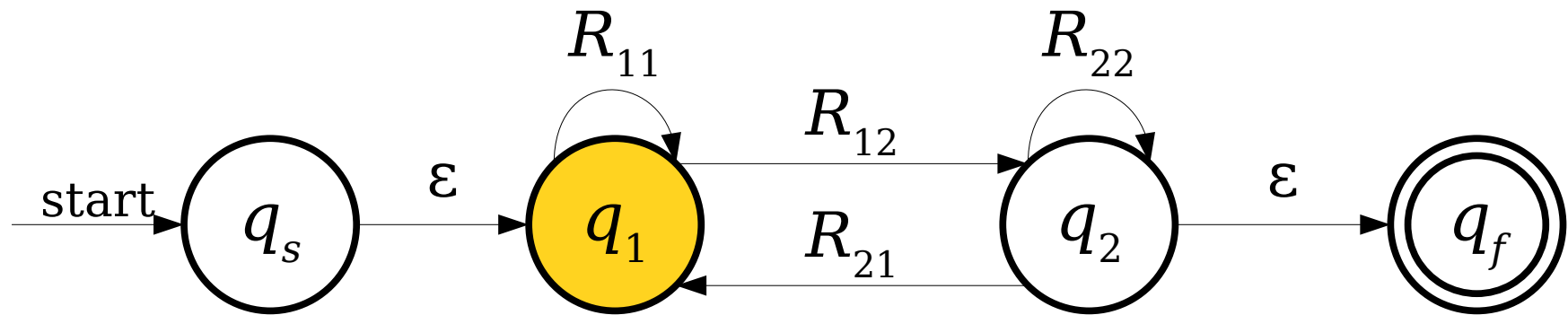
From NFAs to Regular Expressions



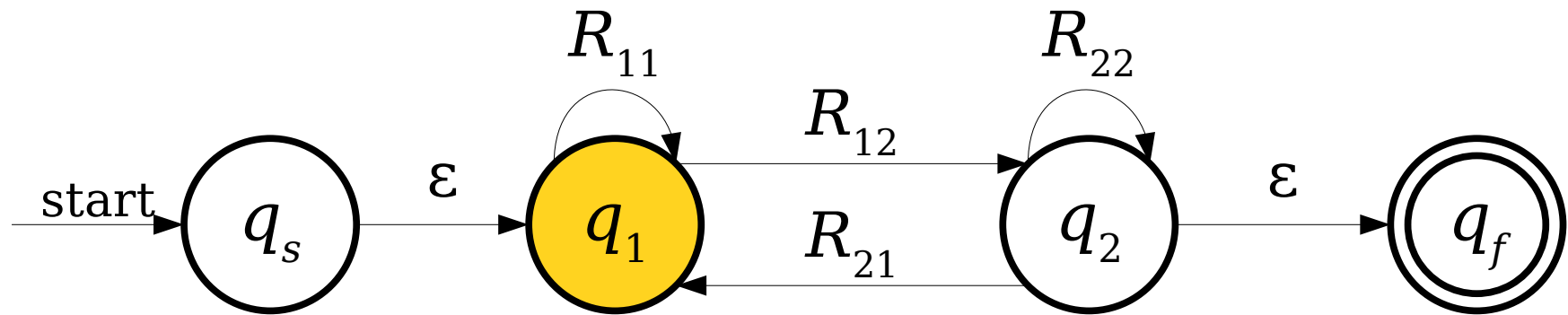
From NFAs to Regular Expressions



From NFAs to Regular Expressions

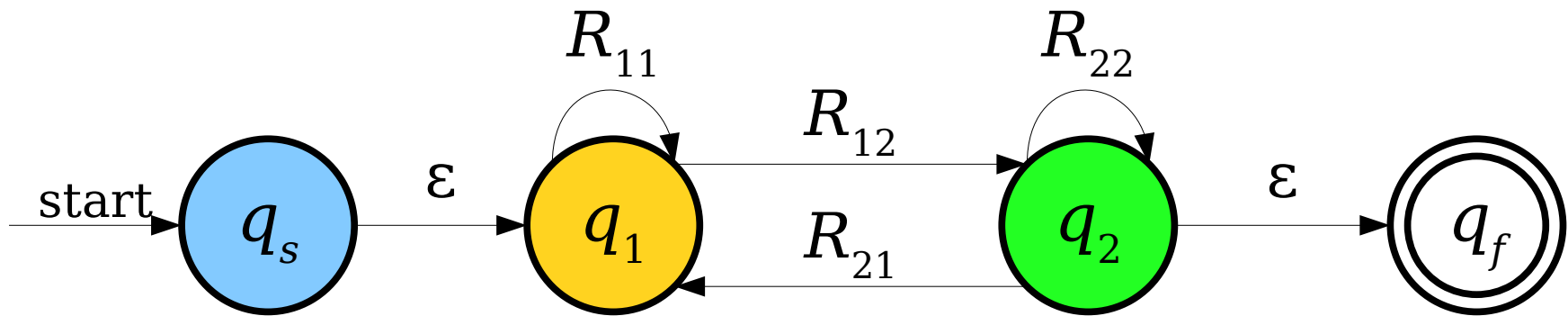


From NFAs to Regular Expressions

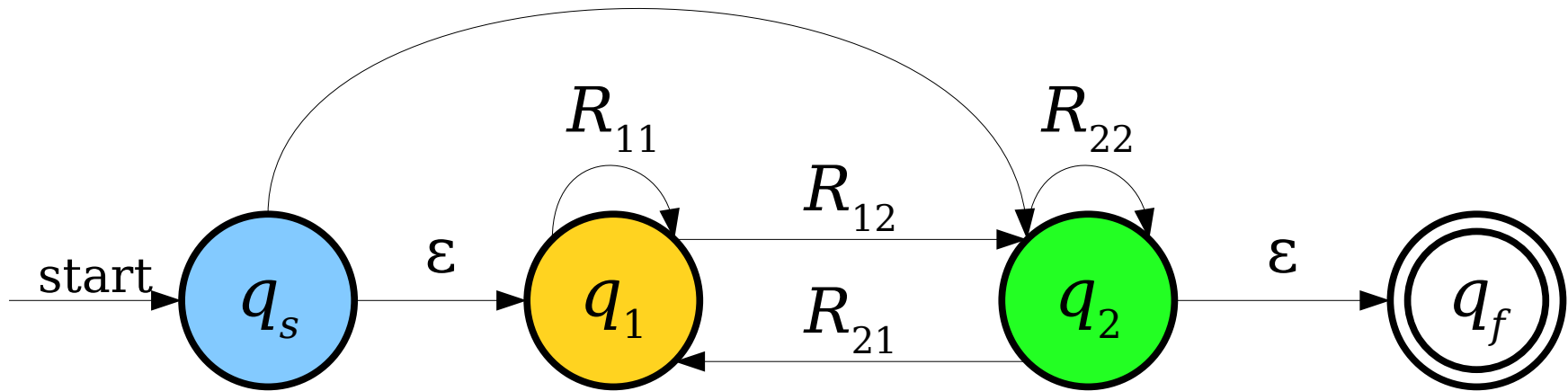


Could we eliminate
this state from
the NFA?

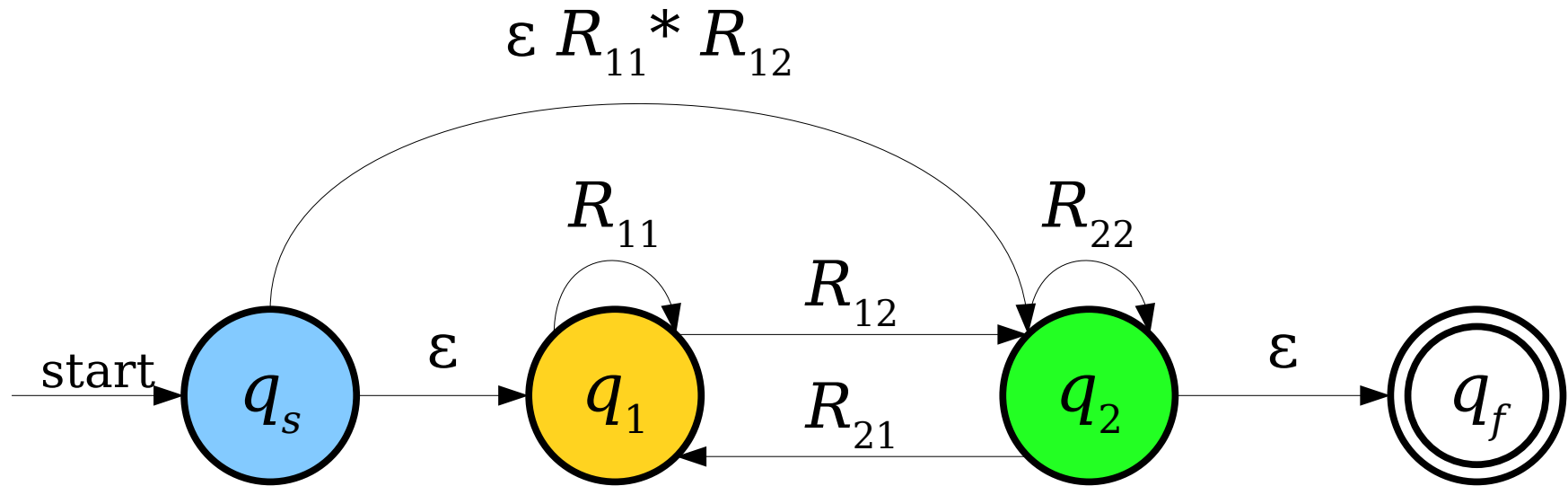
From NFAs to Regular Expressions



From NFAs to Regular Expressions

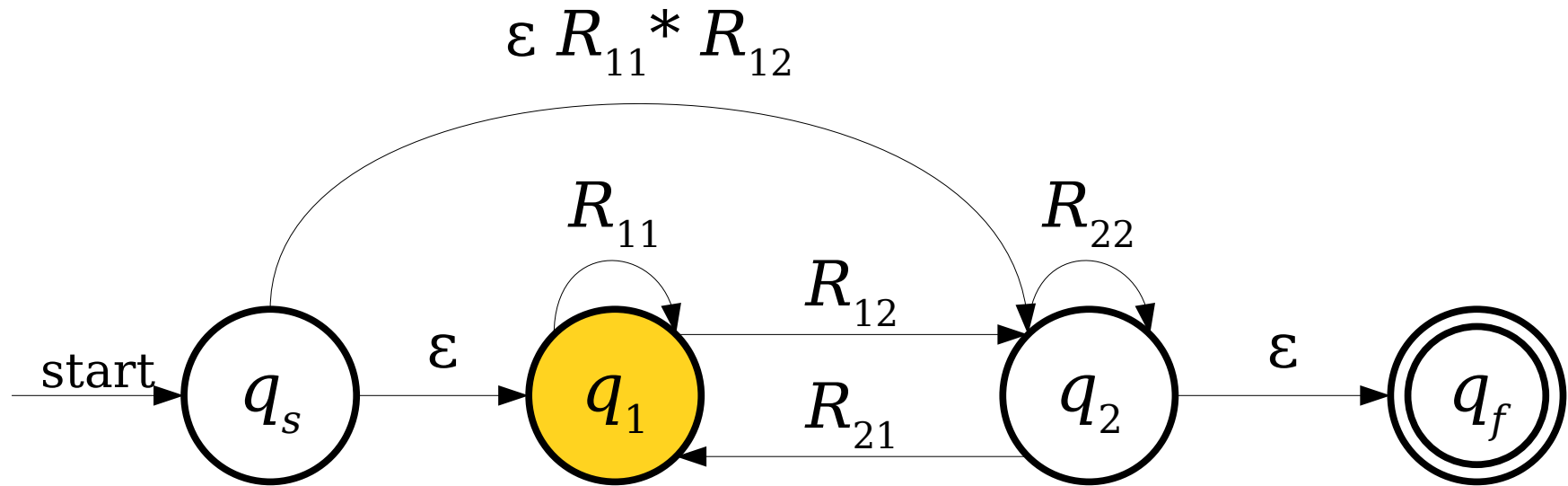


From NFAs to Regular Expressions

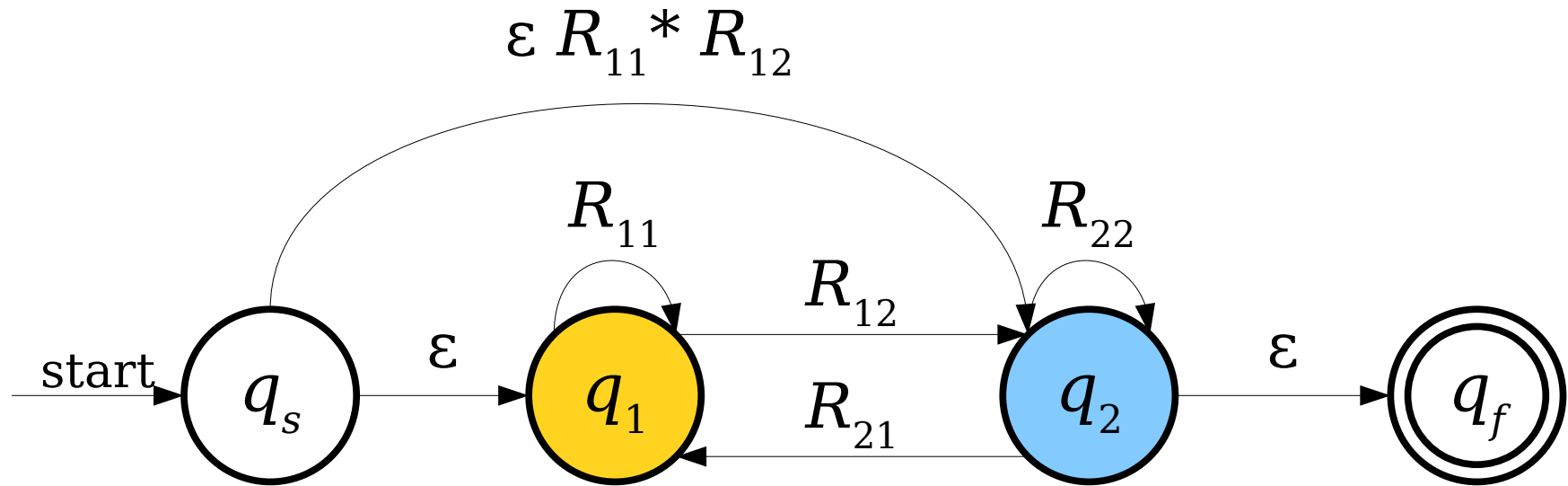


Note: We're using
concatenation and
Kleene closure in order
to skip this state.

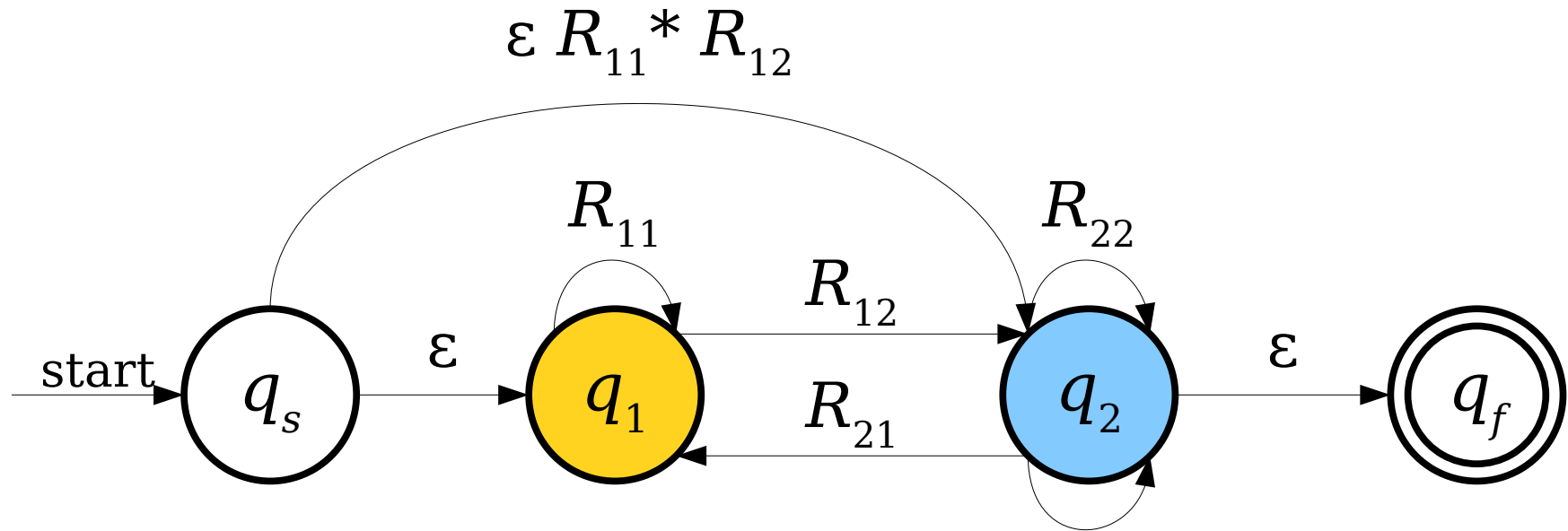
From NFAs to Regular Expressions



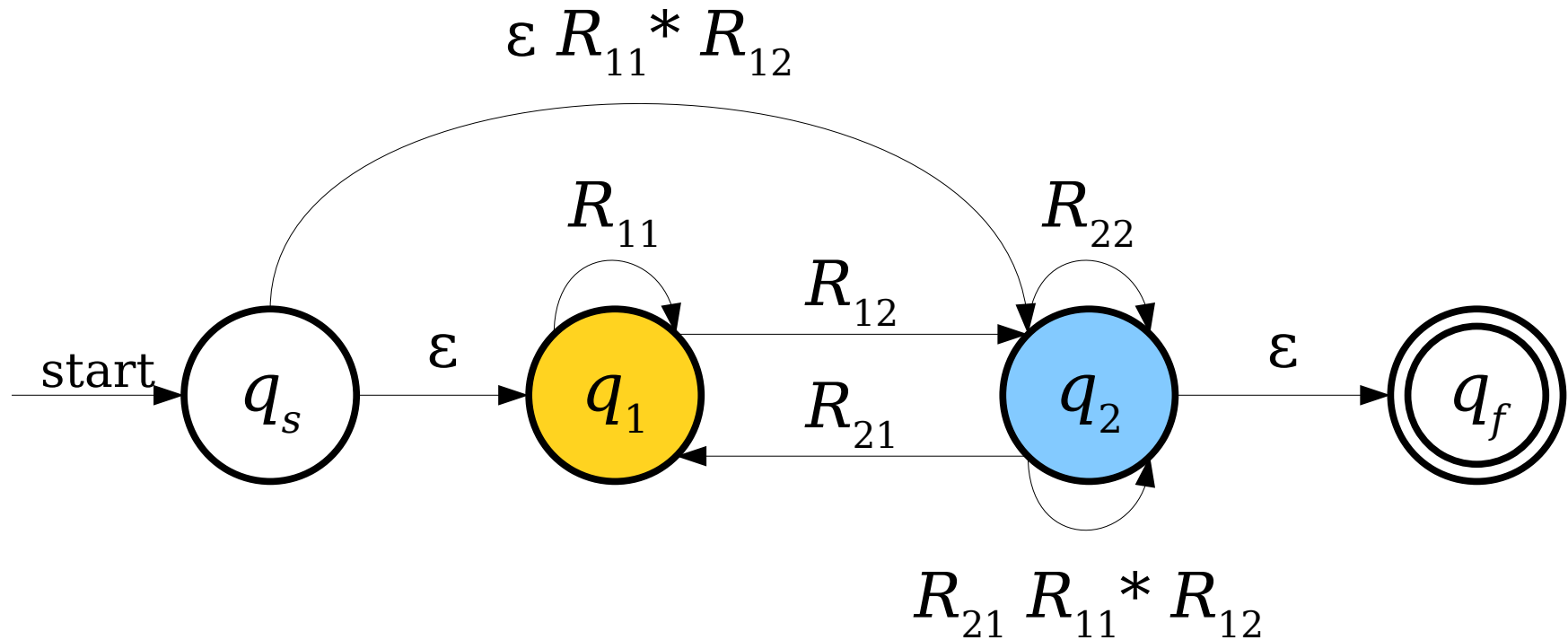
From NFAs to Regular Expressions



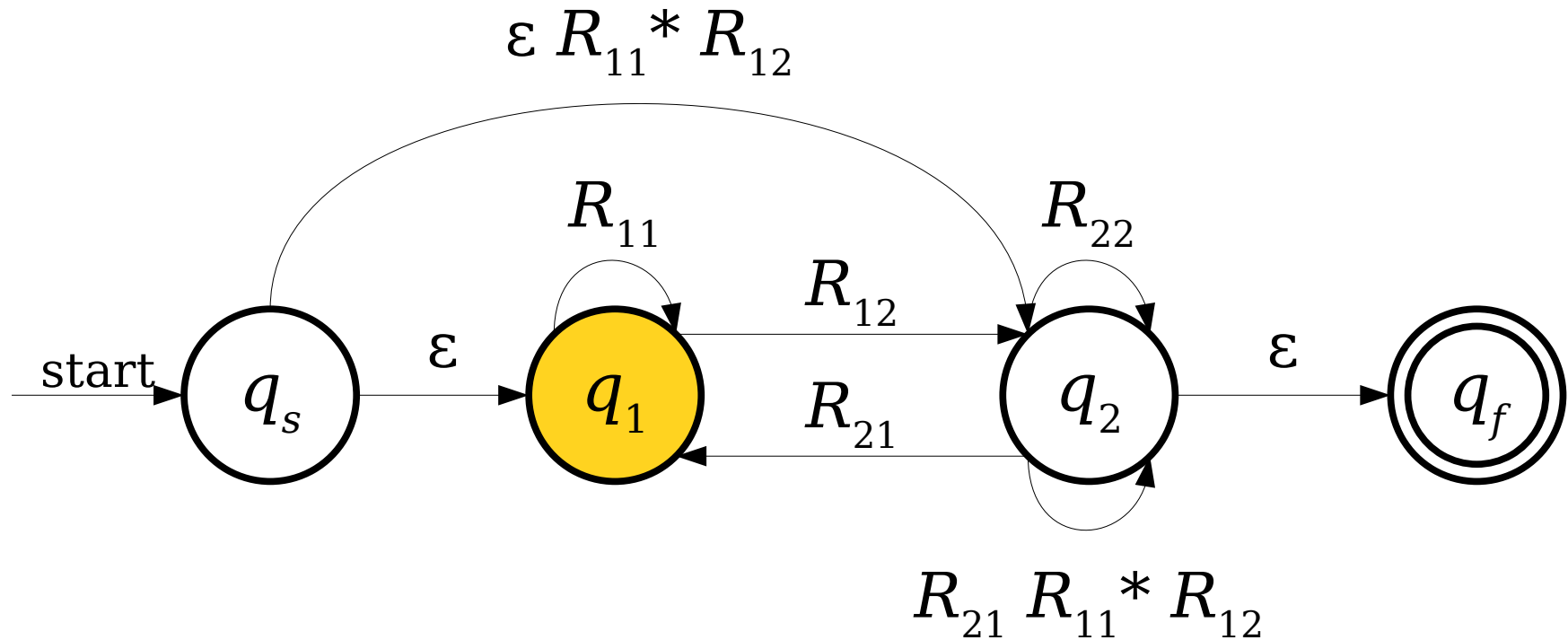
From NFAs to Regular Expressions



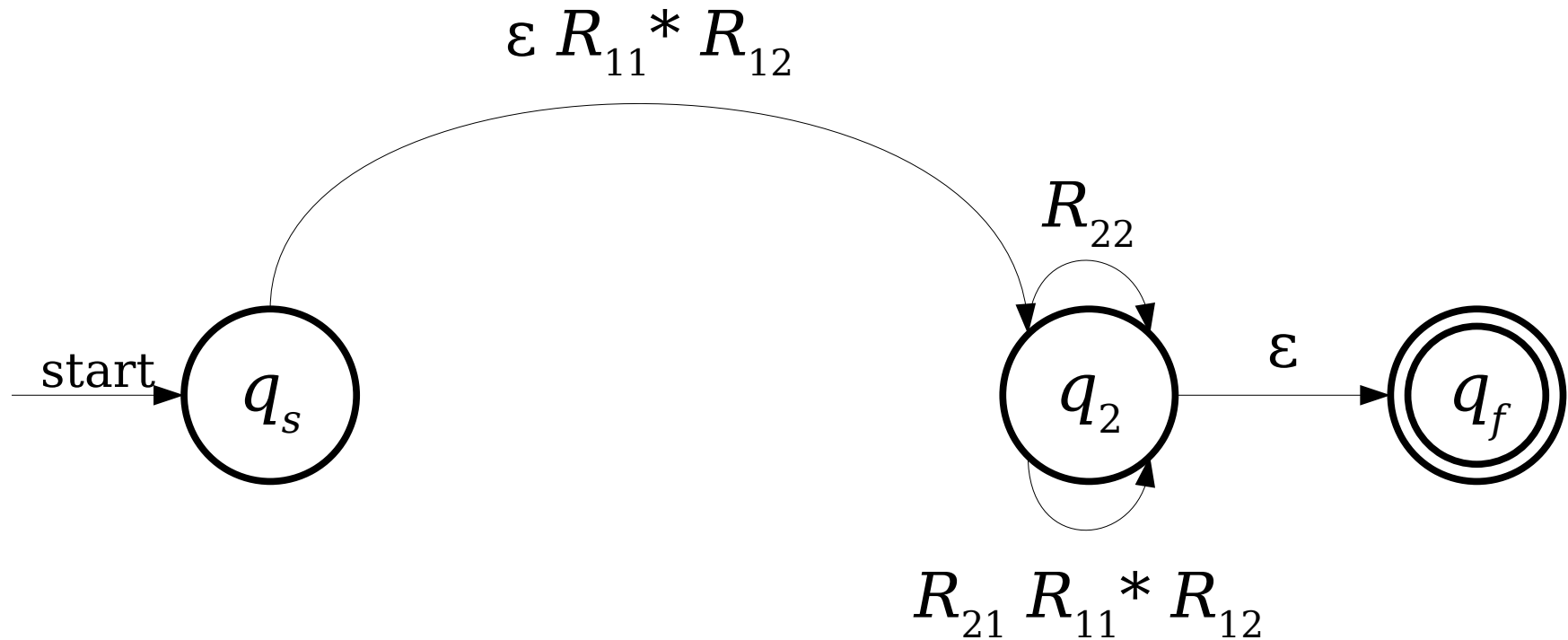
From NFAs to Regular Expressions



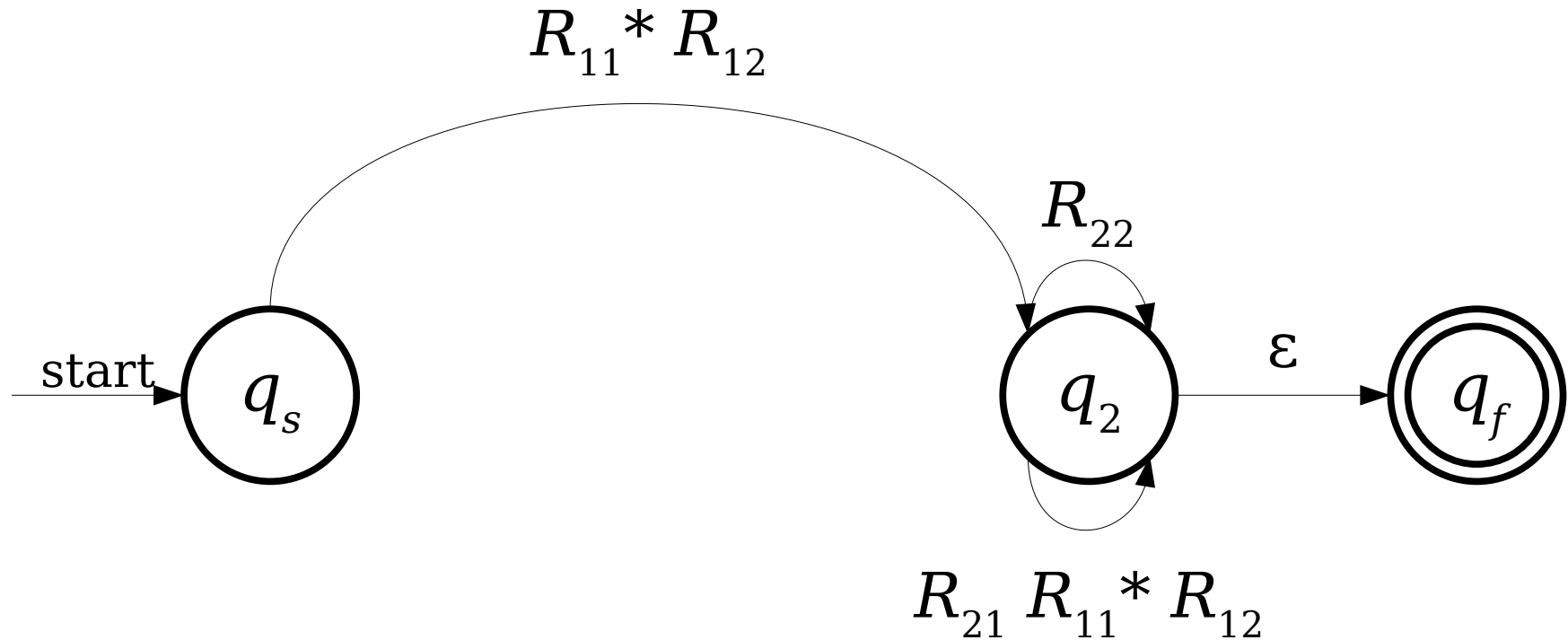
From NFAs to Regular Expressions



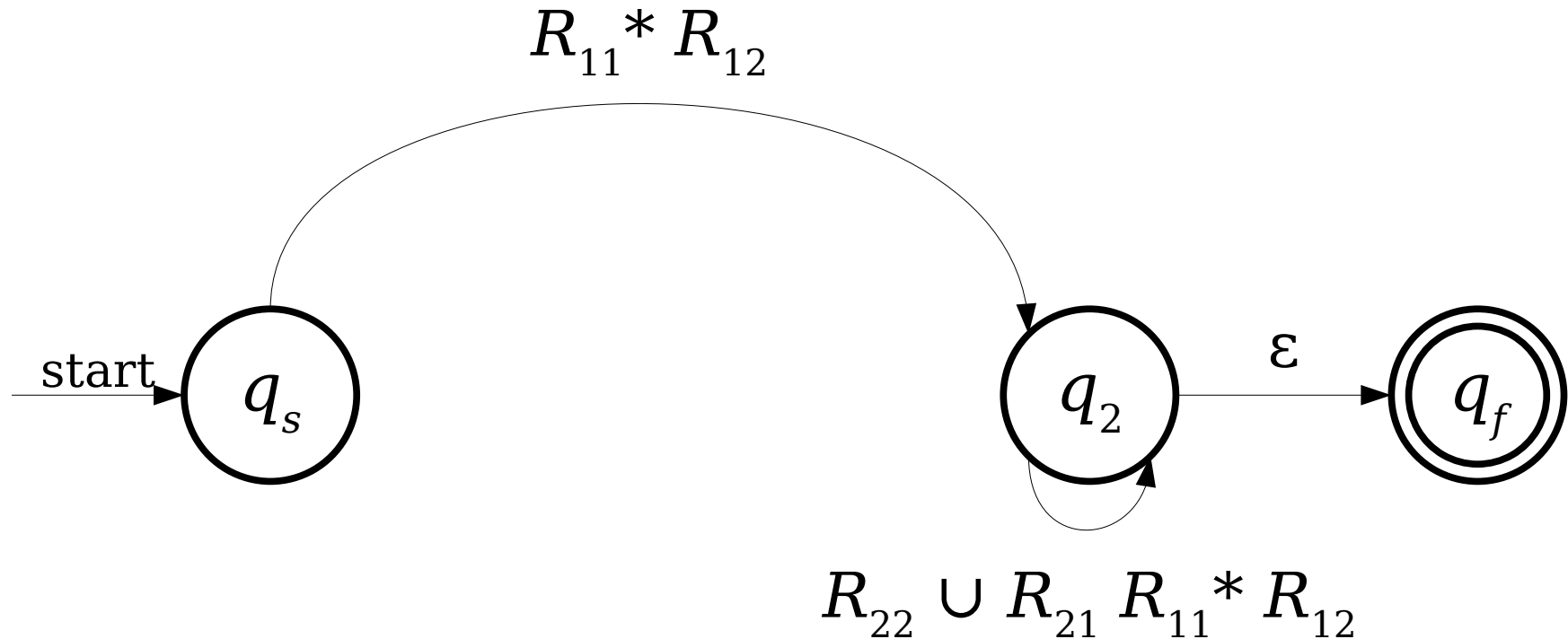
From NFAs to Regular Expressions



From NFAs to Regular Expressions

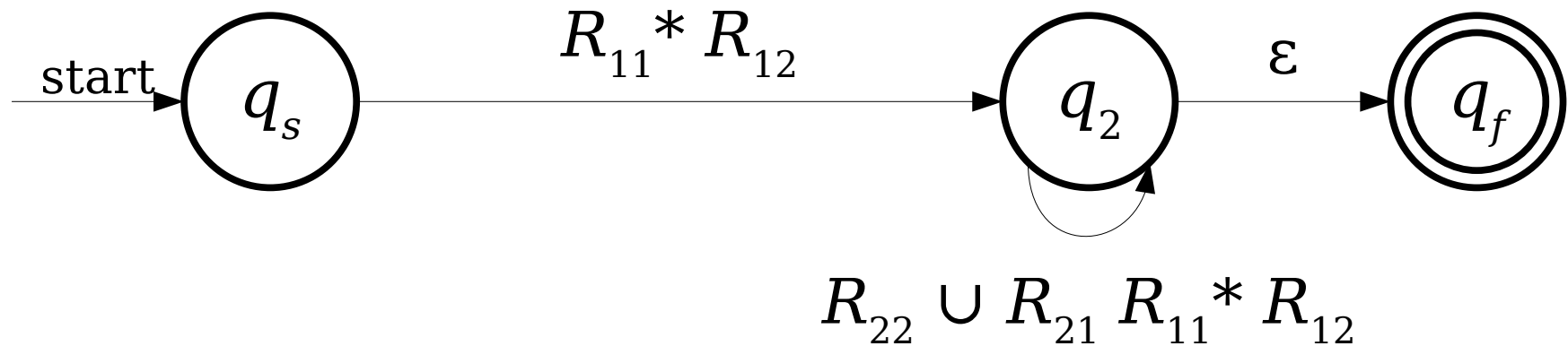


From NFAs to Regular Expressions

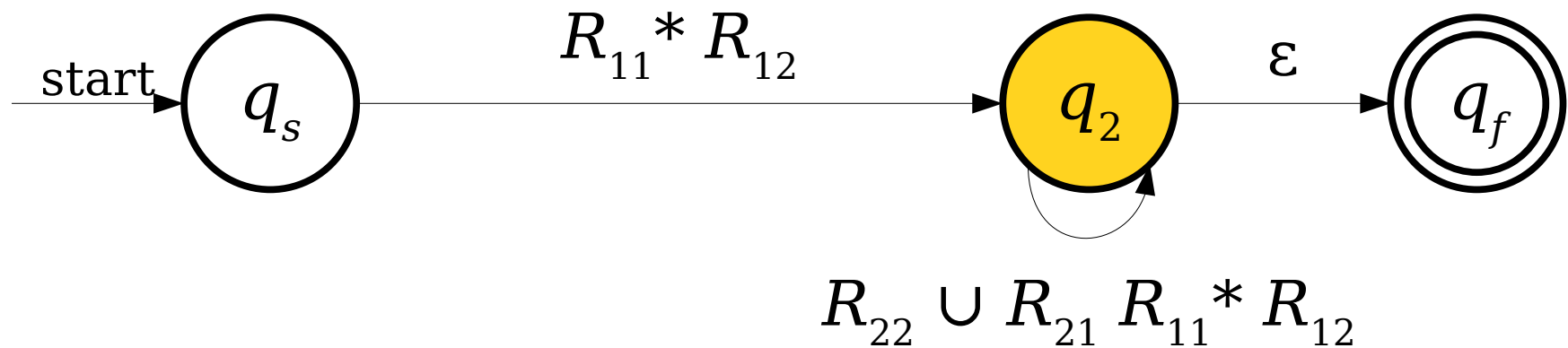


Note: We're using **union** to combine these transitions together.

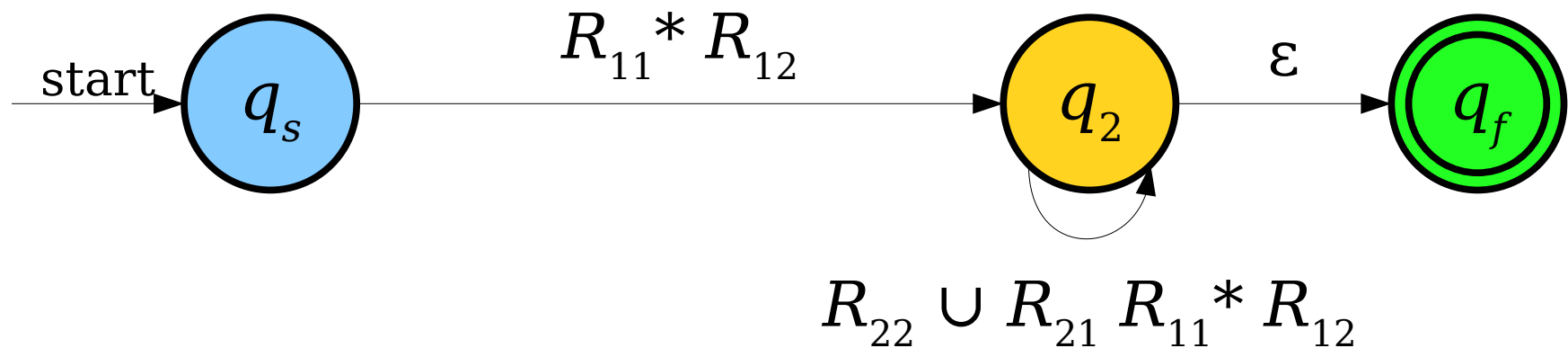
From NFAs to Regular Expressions



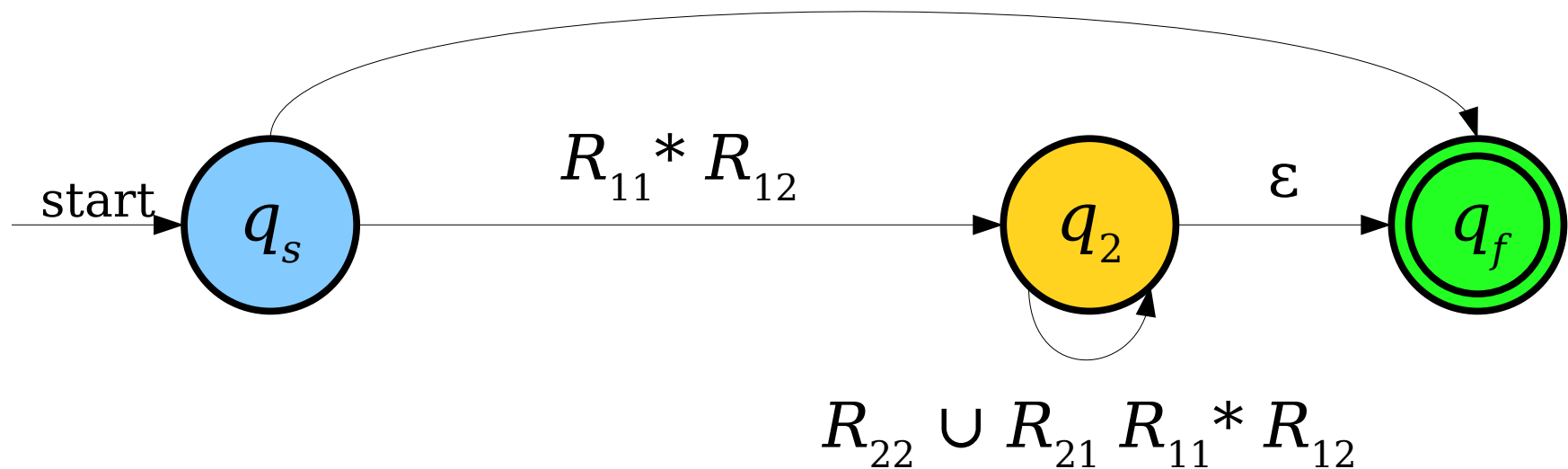
From NFAs to Regular Expressions



From NFAs to Regular Expressions

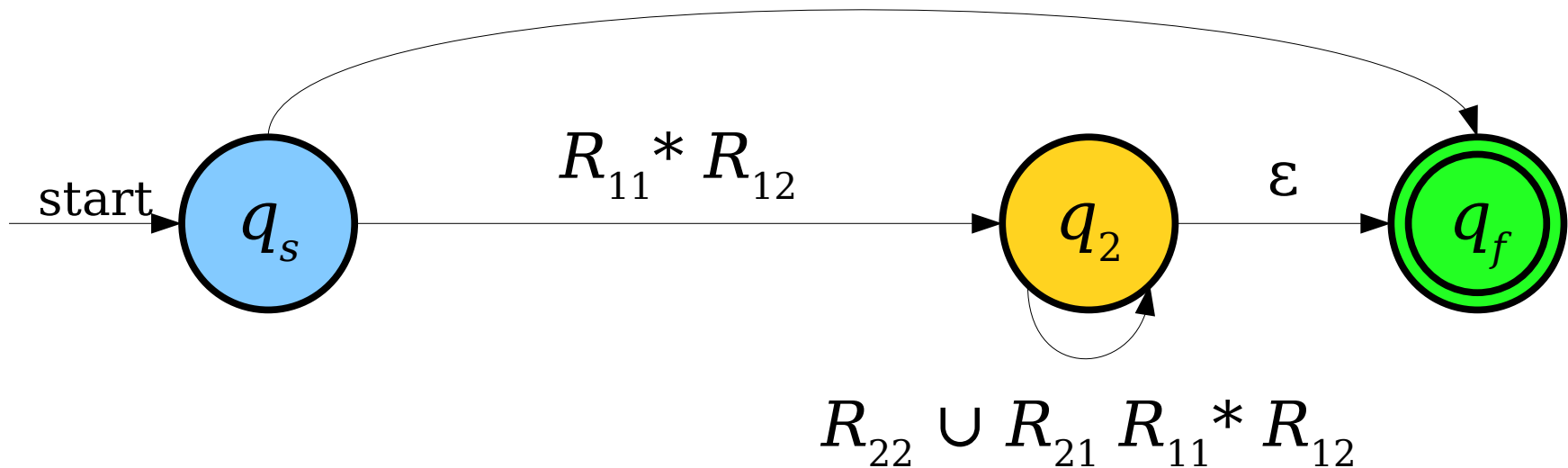


From NFAs to Regular Expressions

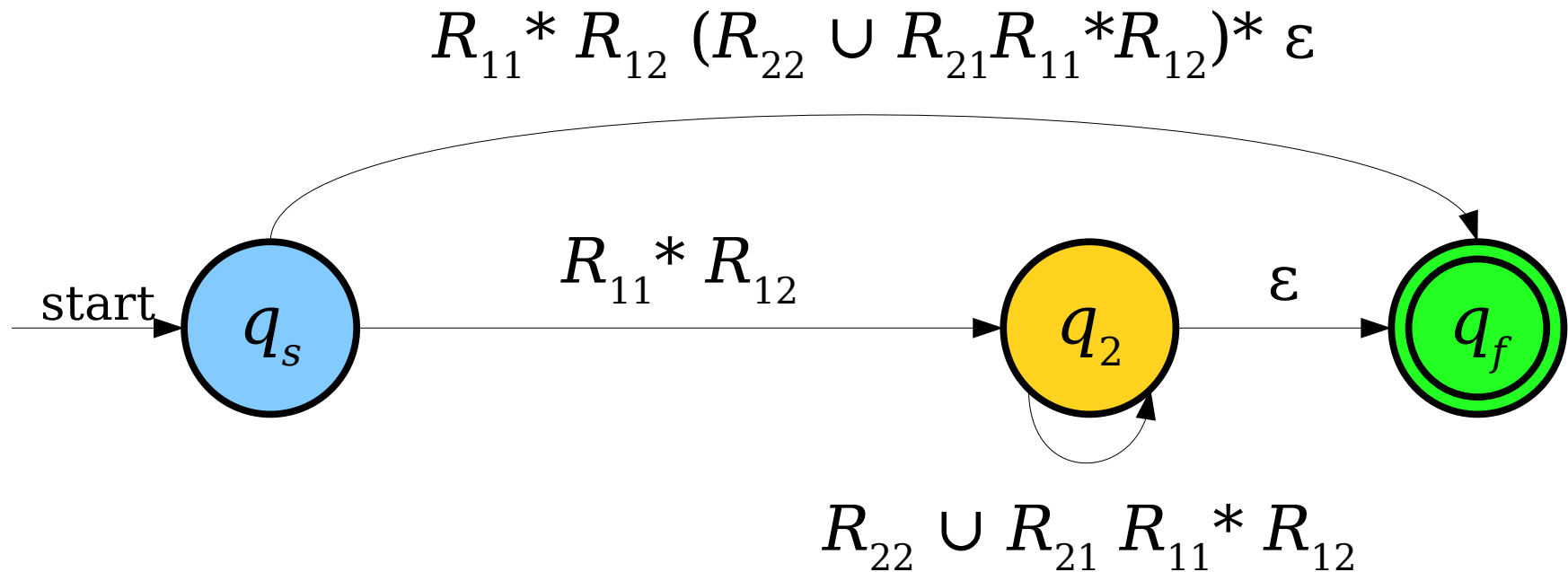


From NFAs to Regular Expressions

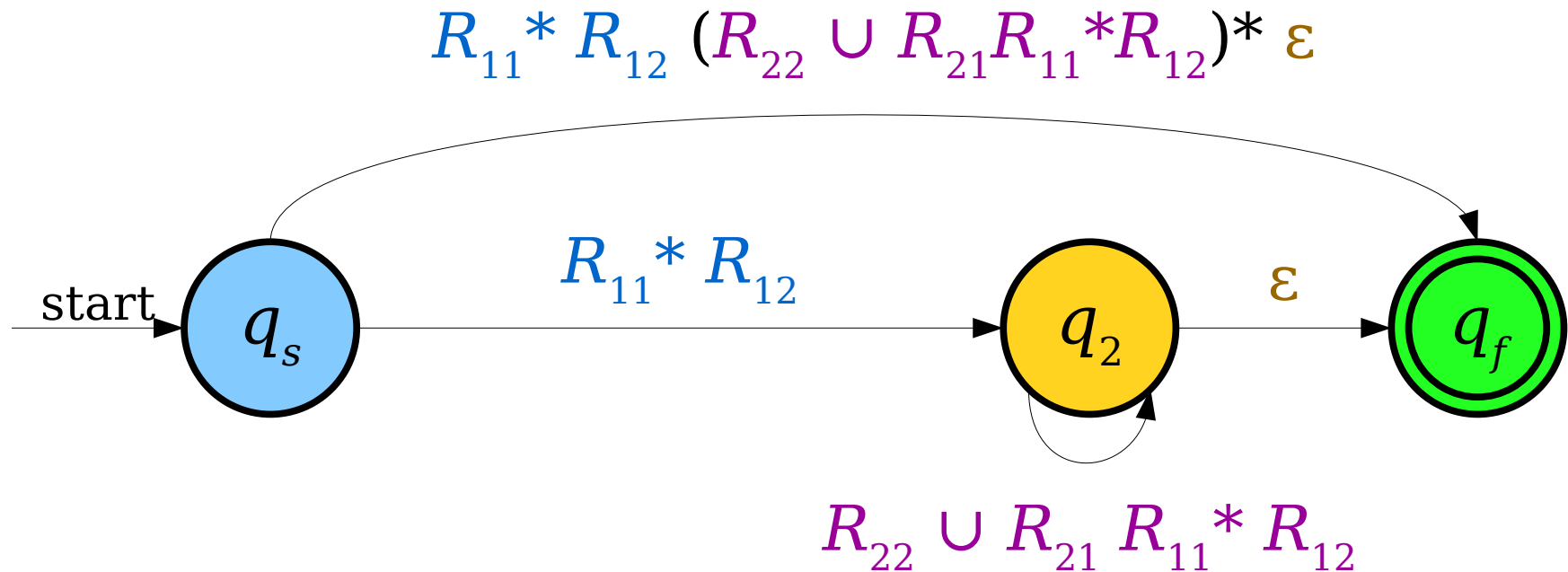
What should we put on this transition?



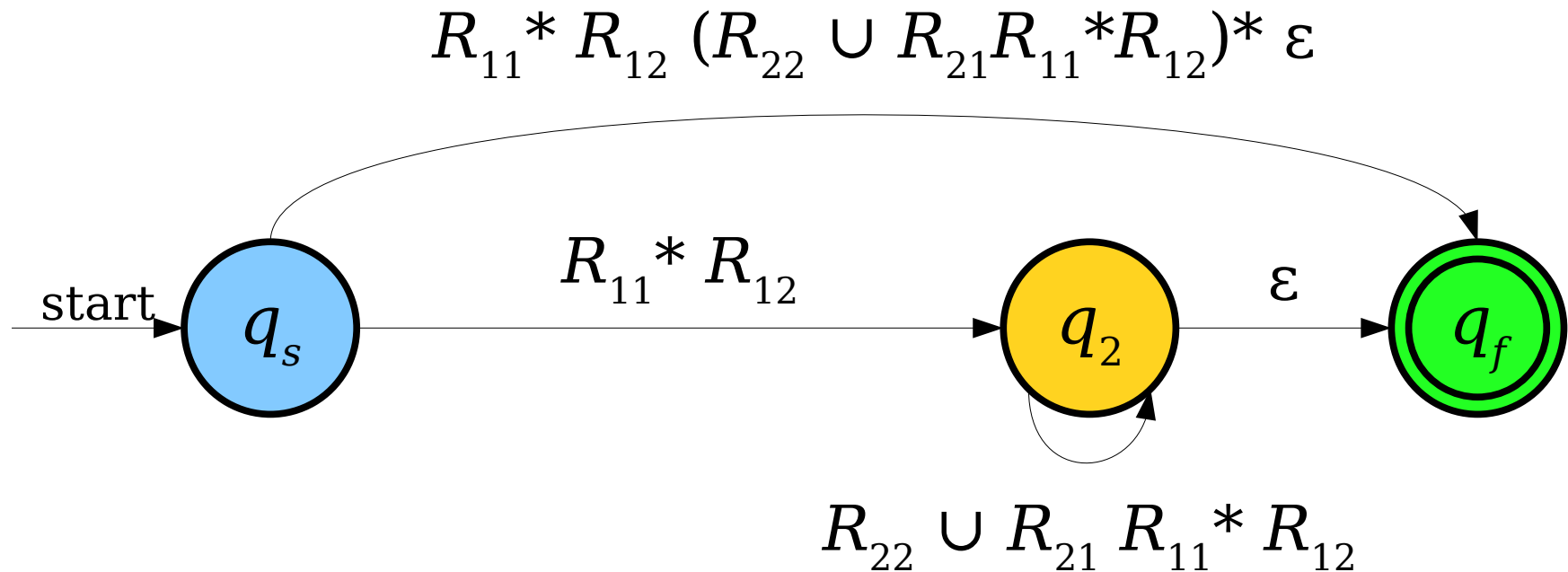
From NFAs to Regular Expressions



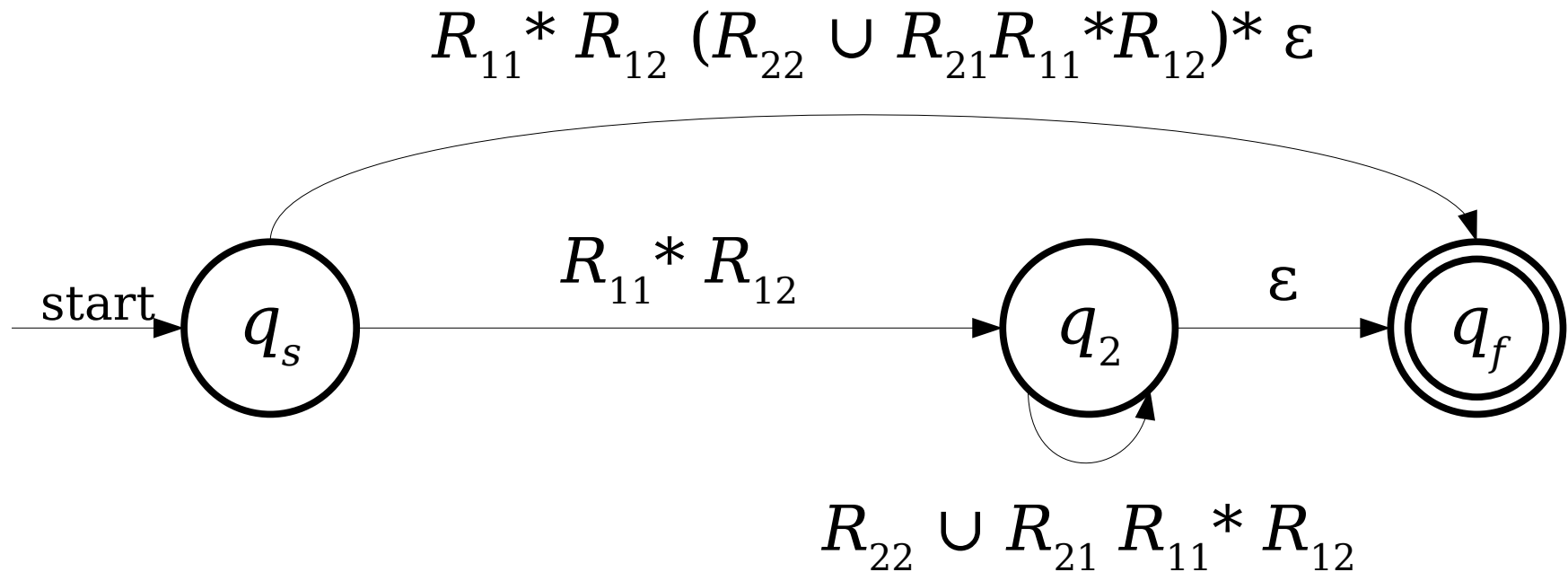
From NFAs to Regular Expressions



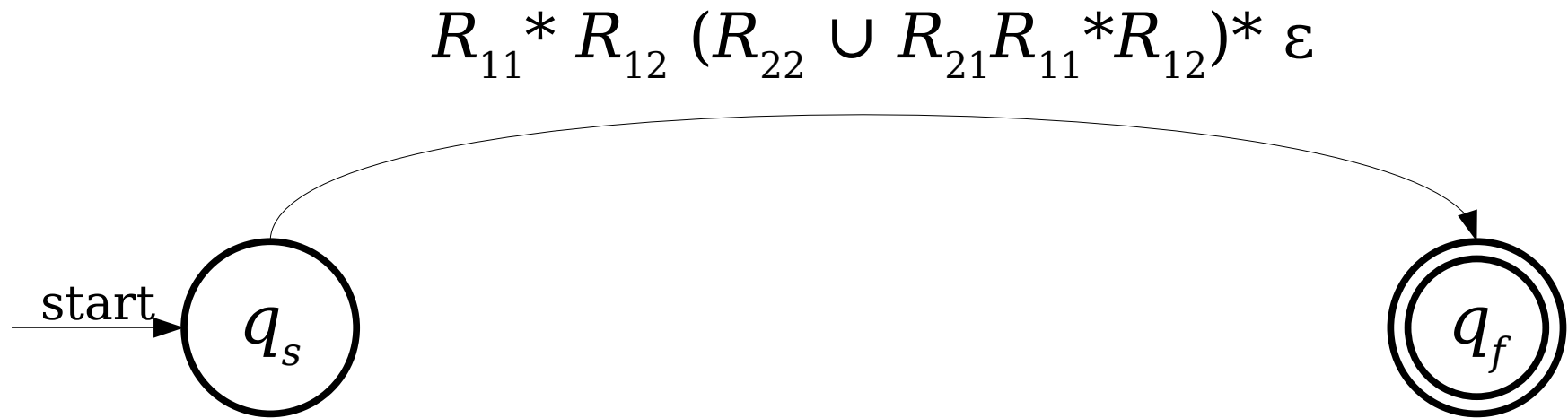
From NFAs to Regular Expressions



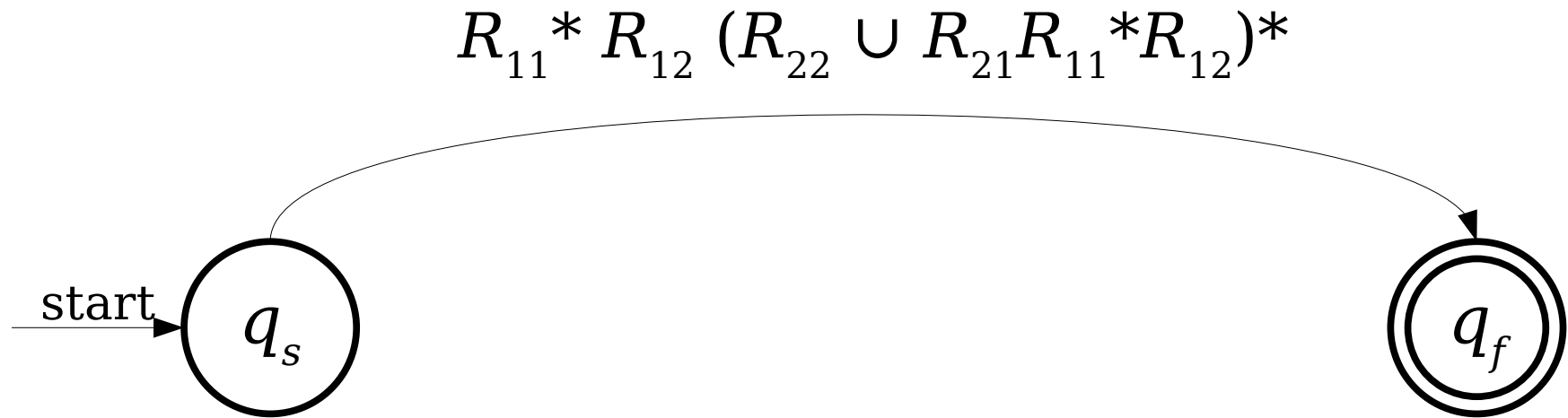
From NFAs to Regular Expressions



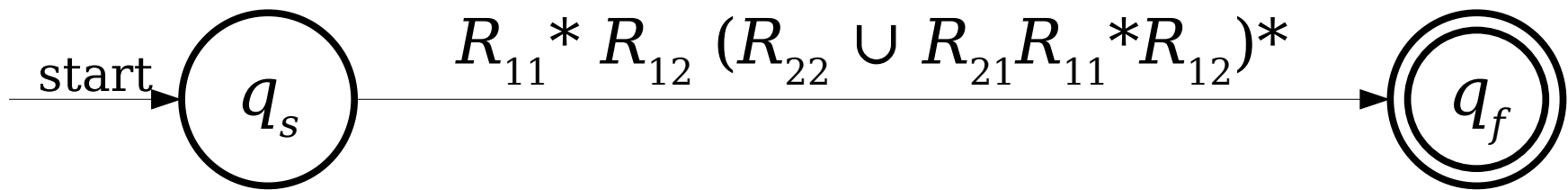
From NFAs to Regular Expressions



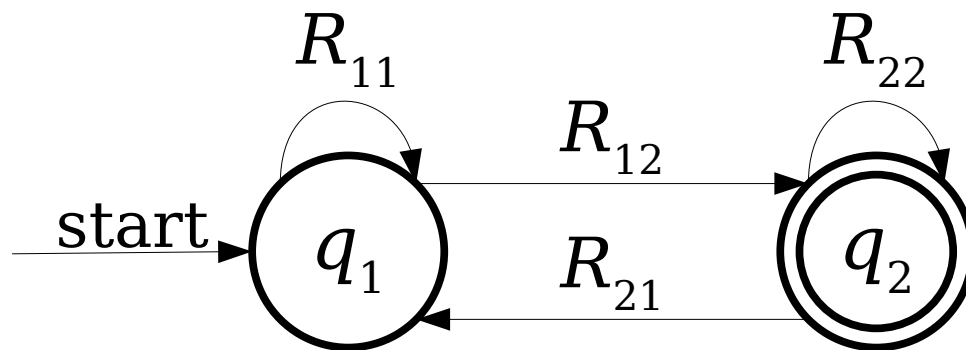
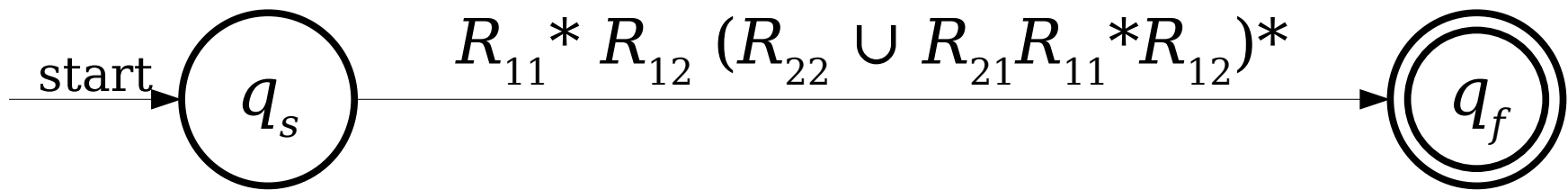
From NFAs to Regular Expressions



From NFAs to Regular Expressions



From NFAs to Regular Expressions



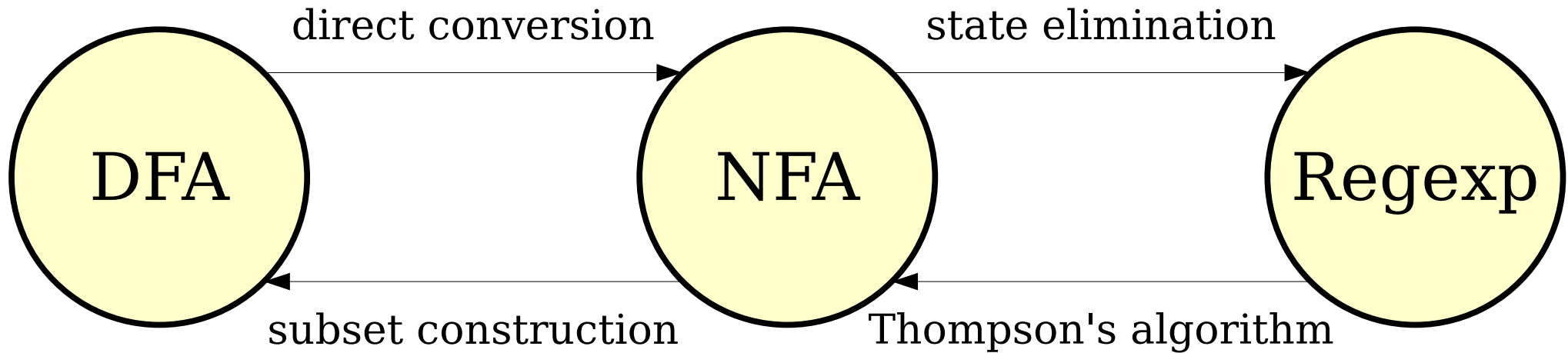
The State-Elimination Algorithm

- 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, \dots, R_k , replace them with a single transition labeled $R_1 \cup R_2 \cup \dots \cup R_k$.

Our Transformations



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.