# Regular Expressions

# First, Some Announcements!

# Second Midterm Logistics

- Our second midterm is next ***Monday, November 11th*** from ***7PM – 10PM***, locations TBA.
- 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.
- Room assignments to be posted late tonight.

# Preparing for the Exam

- The top skills that will serve you well on this exam:
  - ***Knowing how to set up a proof***. This is a recurring theme across functions, sets, graphs, pigeonhole, and induction.
  - ***Distinguishing between assuming and proving***. This similarly cuts across all of these topics.
  - ***Reading new definitions***. This is at the heart of mathematical reasoning.
  - ***Writing proofs in line with definitions***. Folks often ask about whether they're being rigorous enough. Often "rigorous enough" simply means "following what the definitions say."
- Our personal recommendation: when working through practice problems, pay super extra close attention to these areas.

# Preparing for the Exam

- As with the first midterm exam, we've posted a bunch of practice exams on the course website.
  - There are ten practice exams (yes, really!). We realistically don't expect anyone to complete them all. They're there to give you a feeling of what the exam might look like.
- Some general notes on preparing:
  - Q5 and Q6 on PS6, while technically on topics that aren't covered on the midterm, are great practice for the sorts of reasoning you'll need on the exam.
  - ***Keep the TAs in the loop when studying***. Ask for feedback on any proofs you write when getting ready for the exam.
  - Don't skip on biological care and maintenance. Exams can be stressful, but please make time for basic things like showering, eating, etc. and for self-care in whatever form that takes for you.
- ***You can do this***. Best of luck on the exam!

On to CS103!

# Recap from Last Time

# Regular Languages

- A language $L$ is called a ***regular language*** if there is a DFA or an NFA for $L$.
- ***Theorem:*** The following are equivalent:
  - $L$ is a regular language.
  - There is a DFA $D$ where $\mathscr{L}(D) = L$.
  - There is an NFA $N$ where $\mathscr{L}(N) = L$.
- In other words, knowing any one of the above three facts means you know the other two.

# 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

$$L_1L_2 = \{ x \mid \exists w_1 \in L_1.\ \exists w_2 \in L_2.\ x = w_1w_2 \}$$

- 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 $L$ = { aa, b }
- $LL$ is the set of strings formed by concatenating pairs of strings in $L$.

{ aaaa, aab, baa, bb }

- $LLL$ is the set of strings formed by concatenating triples of strings in $L$.

{ aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbaa, bbb}

- $LLLL$ 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}

# Language Exponentiation

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

$$L^0 = \{\varepsilon\} \qquad L^{n+1} = LL^n$$

- So, for example, { **aa**, **b** }$^3$ is the language

{ **aaaaaa**, **aaaab**, **aabaa**, **aabb**,
**baaaa**, **baab**, **bbaa**, **bbb**}

# 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:

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

- Intuitively, all possible ways of concatenating zero or more strings in $L$ together, possibly with repetition.

# The Kleene Closure

If $L$ = { **a**, **bb** }, then $L^*$ = {

ε,

**a**, **bb**,

**aa**, **abb**, **bba**, **bbbb**,

**aaa**, **aabb**, **abba**, **abbbb**, **bbaa**, **bbabb**, **bbbba**, **bbbbbb**,

…

}

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

# Closure Properties

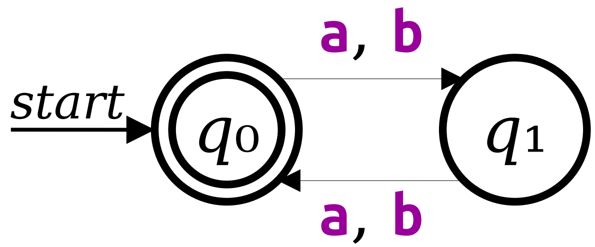
- ***Theorem:*** If $L_1$ and $L_2$ are regular languages over an alphabet $\Sigma$, then so are the following languages:
  - $L_1 \cup L_2$
  - $L_1L_2$
  - $L_1^*$
- These (and other) properties are called ***closure properties of the regular languages***.

New Stuff!

# Another View of Regular Languages

# Devices for Articulating Regular Languages

- Finite Automata




- Set (or other Mathematical) Notation

**{ *w* ∈ Σ* | *w*'s length is even }**

- State Transition Table

| | a | b |
|---|---|---|
| $q_0$ | $q_1$ | $q_1$ |
| $q_1$ | $q_0$ | $q_0$ |

- ***New!*** Regular Expressions

# Devices for Articulating Regular Languages

- Finite Automata




- Set (or other Mathematical) Notation

**{ $w \in \Sigma^*$ | $w$'s length is even }**

- State Transitio

- ***New!*** Regular

**Note:** This one is not unique to regular languages! We can express non-regular languages with set builder notation, as well. More on that another day, when we explore other families of 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.

# 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.*

# Atomic Regular Expressions

- The regular expressions begin with three simple building blocks.
- The symbol **Ø** is a regular expression that represents the empty language Ø.
- For any **a** ∈ Σ, the symbol **a** is a regular expression for the language {**a**}.
- The symbol **ε** is a regular expression that represents the language {ε}.
  - ***Remember: {ε} ≠ Ø!***
  - ***Remember: {ε} ≠ ε!***

# Compound Regular Expressions

- If $R_1$ and $R_2$ are regular expressions, $\boldsymbol{R_1R_2}$ 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, $\boldsymbol{R_1 \cup R_2}$ is a regular expression for the *union* of the languages of $R_1$ and $R_2$.
- If $R$ is a regular expression, $\boldsymbol{R^*}$ is a regular expression for the *Kleene closure* of the language of $R$.
- If $R$ is a regular expression, $\boldsymbol{(R)}$ 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*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:
  - $\mathscr{L}(\boldsymbol{\varepsilon}) = \{\varepsilon\}$
  - $\mathscr{L}(\boldsymbol{\emptyset}) = \emptyset$
  - $\mathscr{L}(\mathbf{a}) = \{\mathbf{a}\}$
  - $\mathscr{L}(R_1 R_2) = \mathscr{L}(R_1)\ \mathscr{L}(R_2)$
  - $\mathscr{L}(R_1 \cup R_2) = \mathscr{L}(R_1) \cup \mathscr{L}(R_2)$
  - $\mathscr{L}(R^{*}) = \mathscr{L}(R)^{*}$
  - $\mathscr{L}((R)) = \mathscr{L}(R)$

Worthwhile activity: Apply this recursive definition to

**a(b∪c)((d))**

and see what you get.

# Designing Regular Expressions

- Let Σ = {**a**, **b**}.
- Let $L$ = { $w$ ∈ Σ* | $w$ 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**
**bbbbbabbbbaabbbb**

# Designing Regular Expressions

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

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

The length of a string w is denoted |w|

# Designing Regular Expressions

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

$\Sigma\Sigma\Sigma\Sigma$

aaaa
baba
bbbb
baaa

# Designing Regular Expressions

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

$\Sigma^4$

**aaaa**
**baba**
**bbbb**
**baaa**

# Designing Regular Expressions

- Let Σ = {**a**, **b**}.
- Let $L$ = { $w$ ∈ Σ* | $w$ contains at most one **a** }.

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

**Σ*aΣ***
**b*ab* ∪ b***
**b*(a ∪ ε)b***
**b*a*b* ∪ b***
**b*(a* ∪ ε)b***

Answer at ***https://cs103.stanford.edu/pollev***

# Designing Regular Expressions

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

**b*(a ∪ ε)b***

**bbbbabbb**
**bbbbbb**
**abbb**
**a**

# Designing Regular Expressions

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

$$\mathtt{a}^+ \; (\mathtt{.a}^+)^* \; \mathtt{@} \; \mathtt{a}^+ \, \mathtt{.a}^+ \; (\mathtt{.a}^+)^*$$

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

# A More Elaborate Design

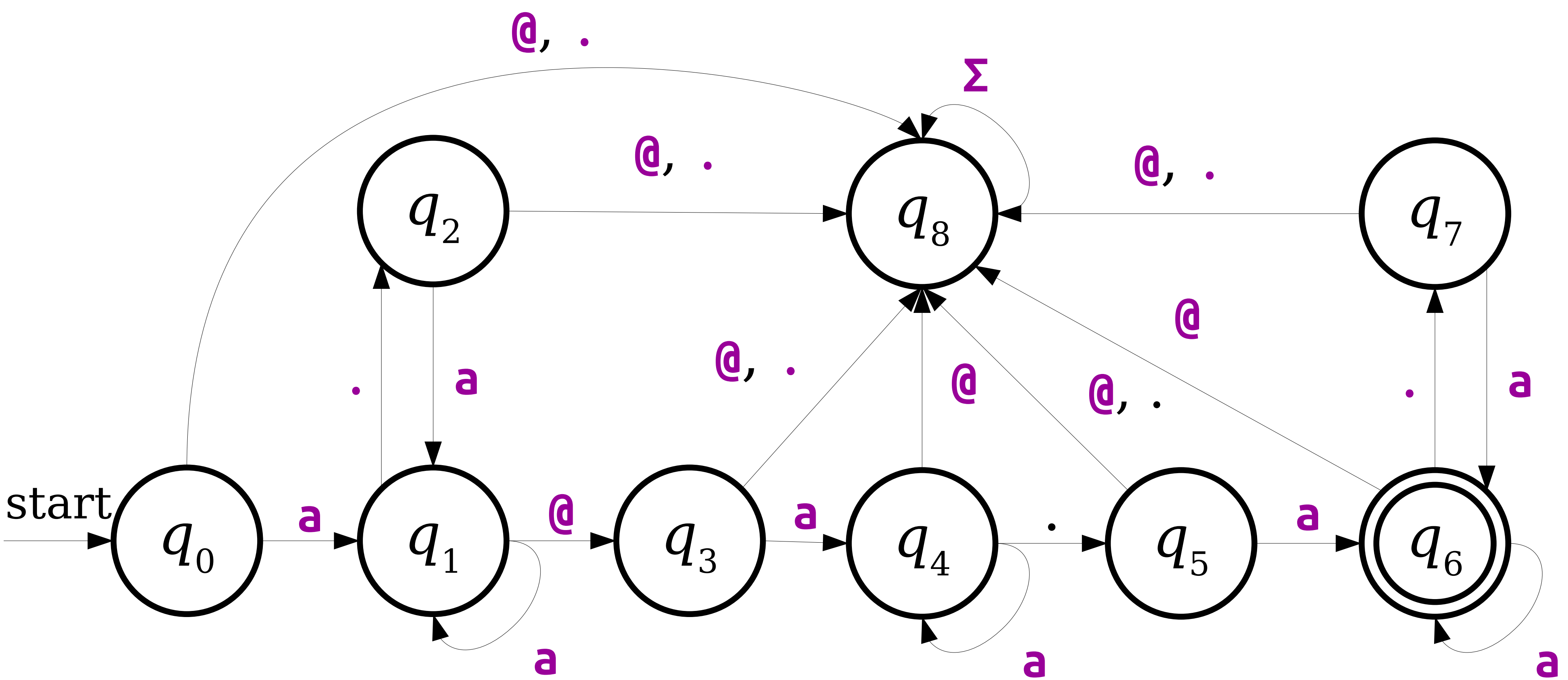
- Let Σ = { **a**, **.**, **@** }, where **a** represents "some letter."
- Let's make a regex for email addresses.

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

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

# For Comparison

**a⁺(.a⁺)*@a⁺(.a⁺)⁺**



# 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

# The Power of Regular Expressions

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

# The Power of Regular Expressions

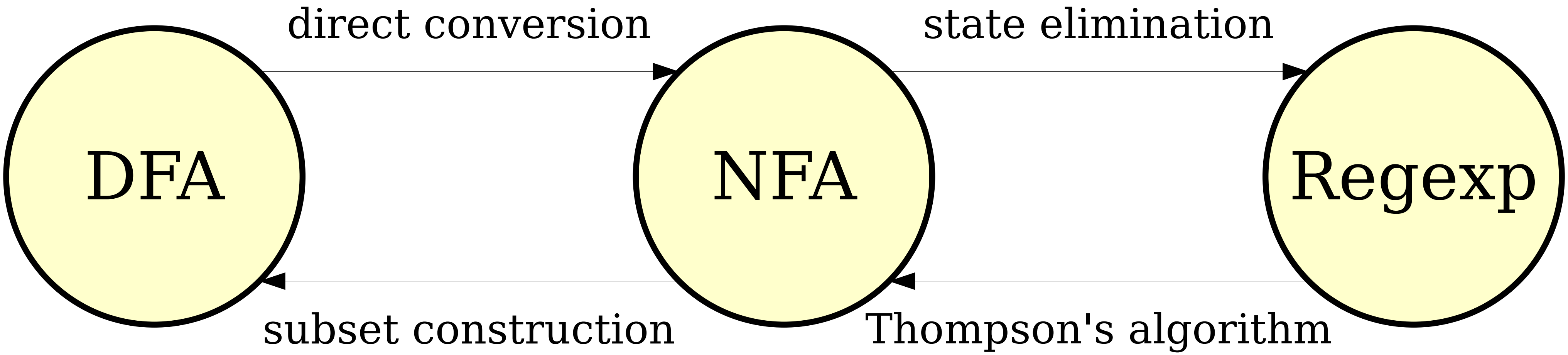
***Theorem:*** If $L$ is a regular language, then there is a regular expression for $L$.

***This is not obvious!***

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

***See appendix!***
***("state elimination")***

# 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 $\mathscr{L}(R) = L$.

# 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!

# Your Action Items

- ***Read "Guide to Regexes"***
  - There's a lot of information and advice there about how to write regular expressions, plus a bunch of worked exercises.
- ***Read "Guide to State Elimination"***
  - It's a beautiful algorithm. The Guide goes into a lot more detail than what we did here.

# Next Time

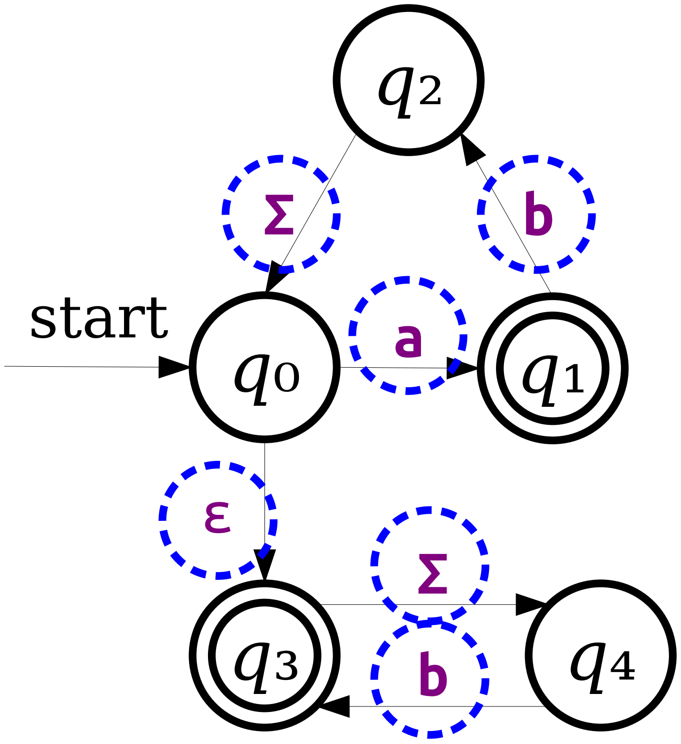
- ***Intuiting Regular Languages***
  - What makes a language regular?
- ***The Myhill-Nerode Theorem***
  - The limits of regular languages.

# Appendix:

State Elimination (NFA → Regex)

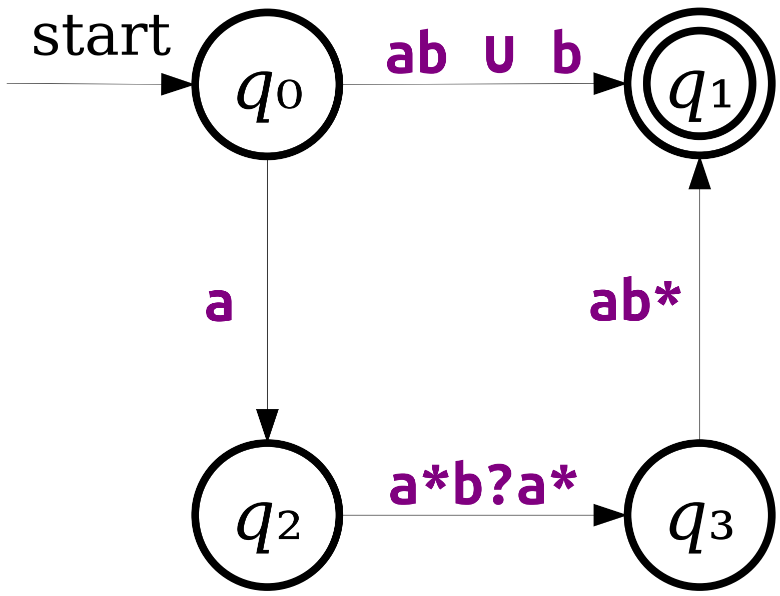
See also: *Guide to State Elimination*

# Generalizing NFAs




These are all regular expressions!

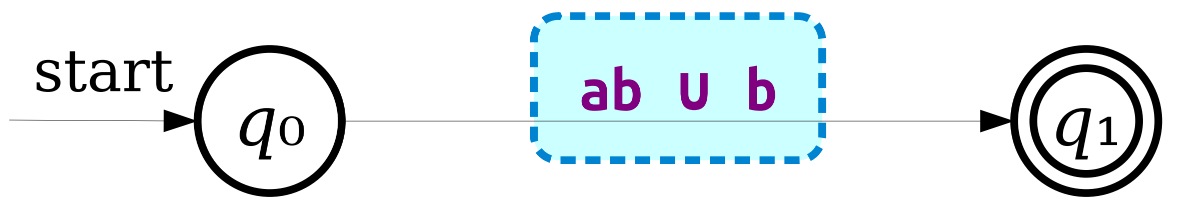
# Generalizing NFAs




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

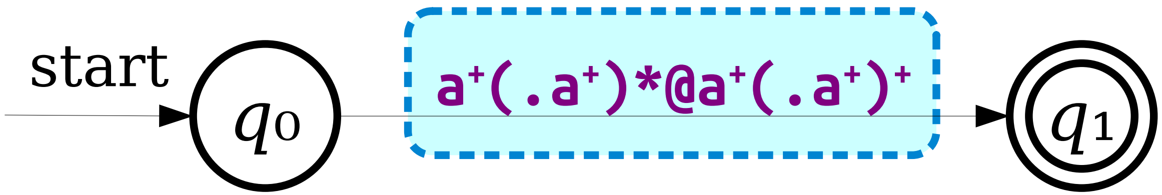
***Key Idea 1:*** Imagine that we can label transitions in an NFA with arbitrary regular expressions.

# Generalizing NFAs




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

# Generalizing NFAs

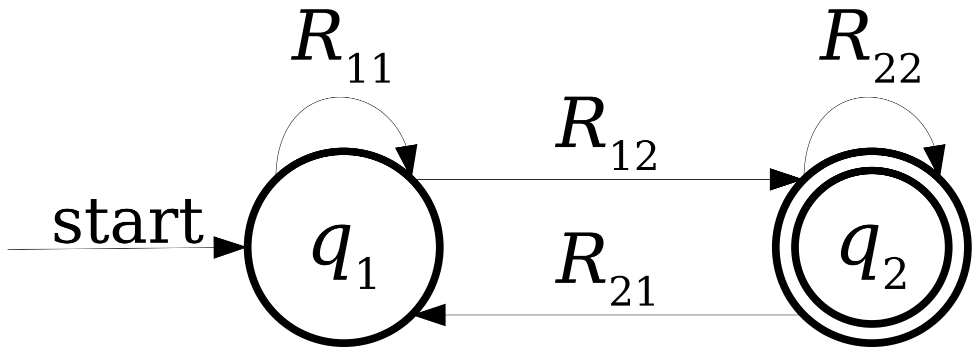



Is there a simple regular expression for the language of this generalized NFA?

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

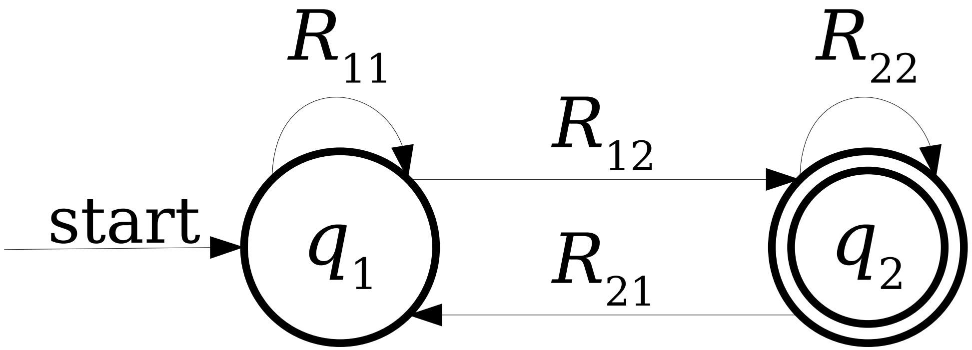

...then we can easily read off a regular expression for the original NFA.

# From NFAs to Regular Expressions



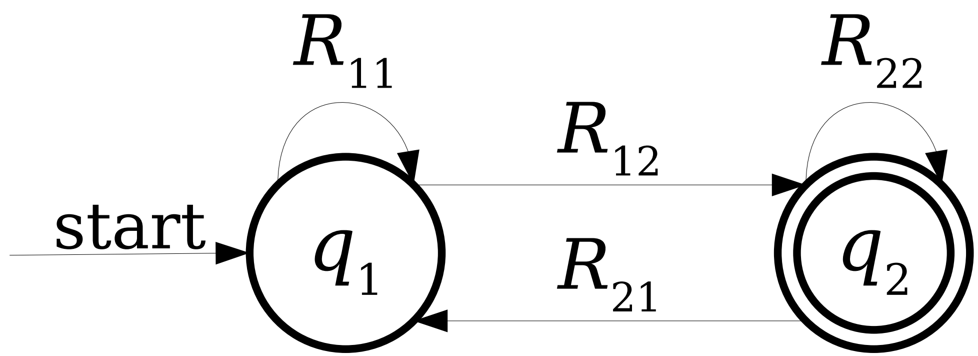

Here, $R_{11}$, $R_{12}$, $R_{21}$, and $R_{22}$ are arbitrary regular expressions.

# From NFAs to Regular Expressions

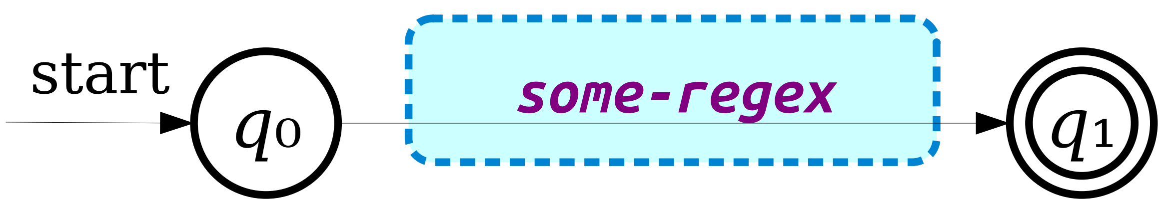



Question: Can we get a clean regular expression from this NFA?

# From NFAs to Regular Expressions




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



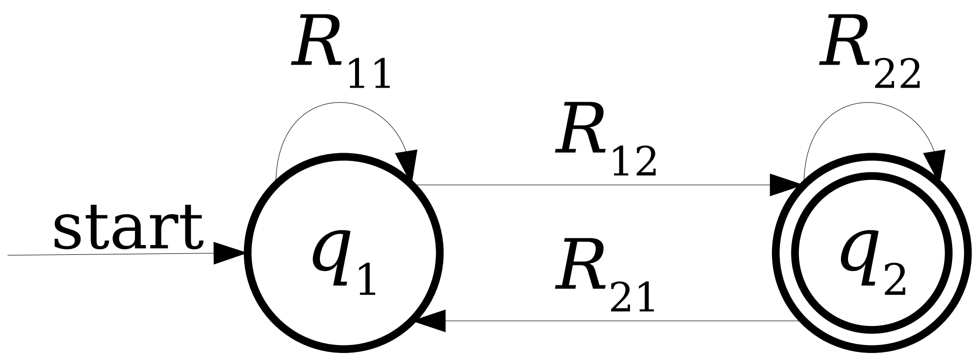
# The State-Elimination Algorithm

- 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

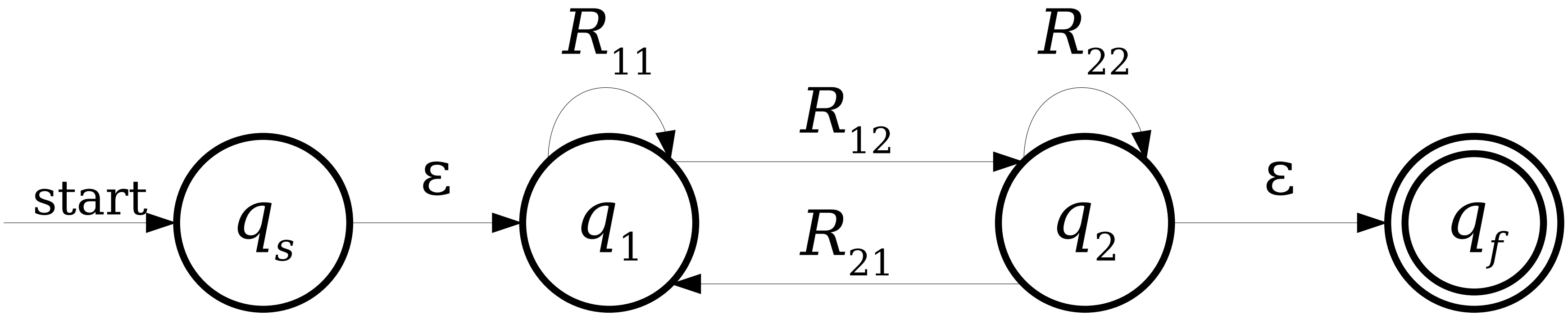
- To eliminate a state $q$ from the automaton, do the following for each pair of states $q_0$ and $q_1$, where there's a transition from $q_0$ into $q$ and a transition from $q$ into $q_1$:
  - Let $R_{in}$ be the regex on the transition from $q_0$ to $q$.
  - Let $R_{out}$ be the regex on the transition 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 $R_1 \cup R_2 \cup \ldots \cup R_k$.

# From NFAs to Regular Expressions

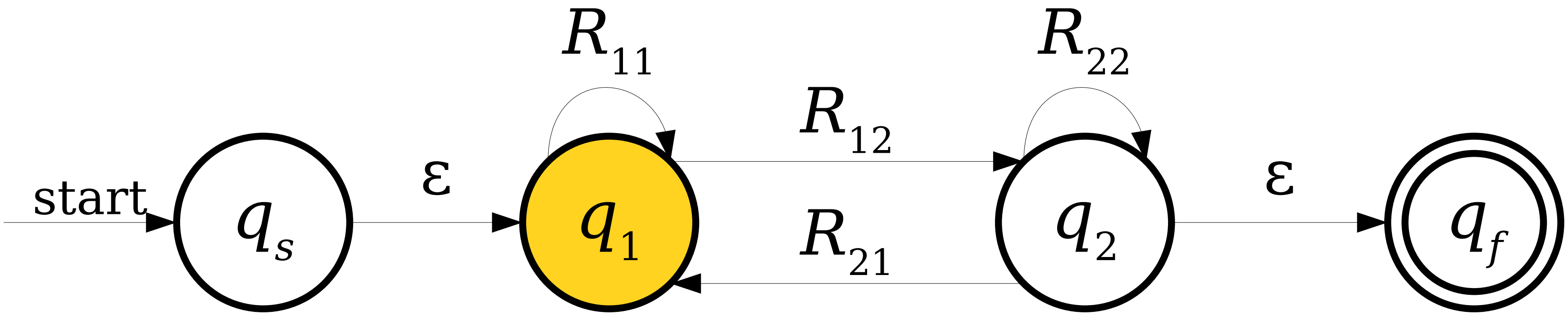



The first step is going to be a bit weird...

# From NFAs to Regular Expressions

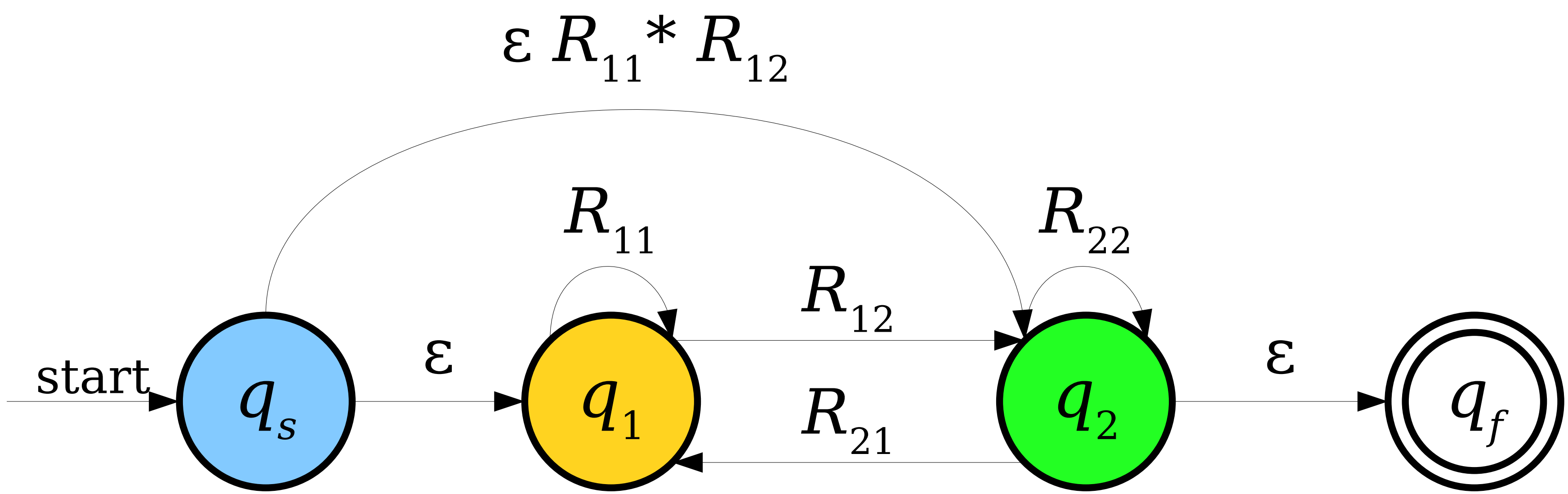


# From NFAs to Regular Expressions



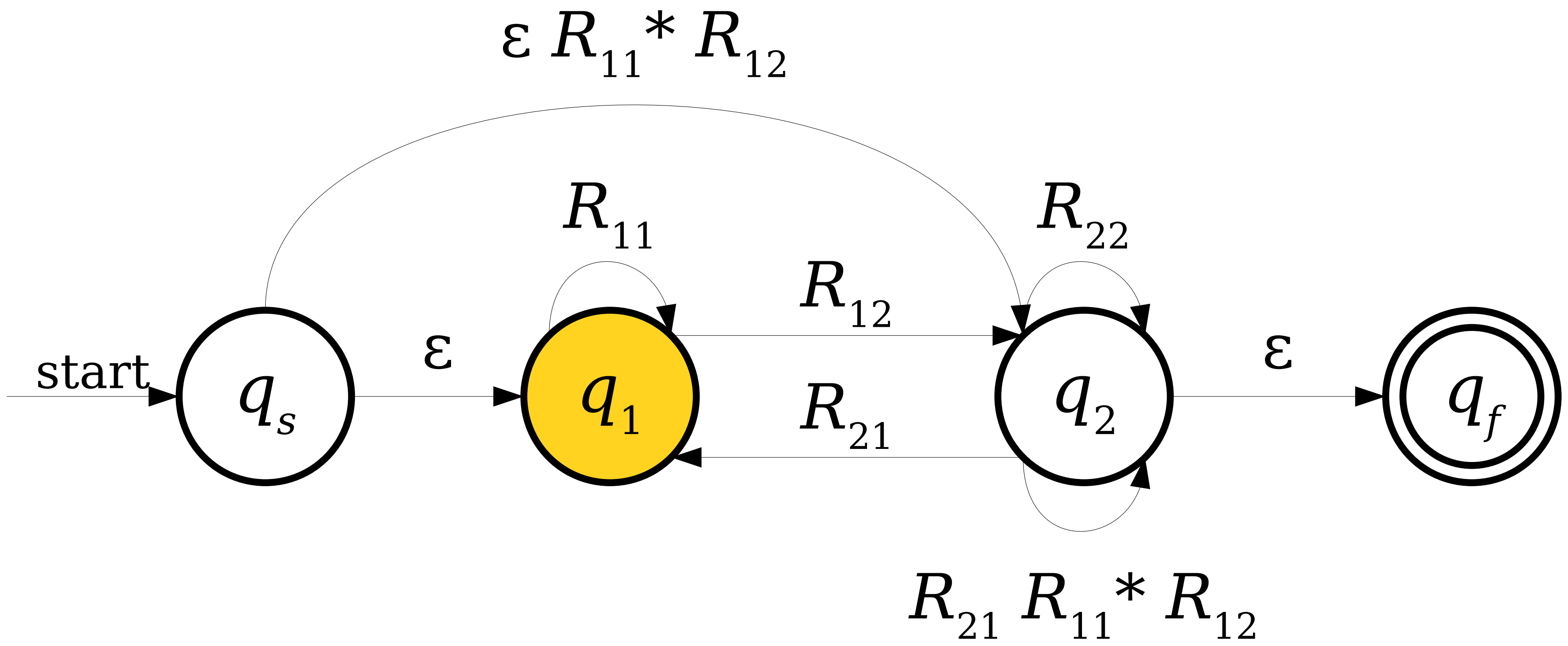

Could we eliminate this state from the NFA?

# From NFAs to Regular Expressions

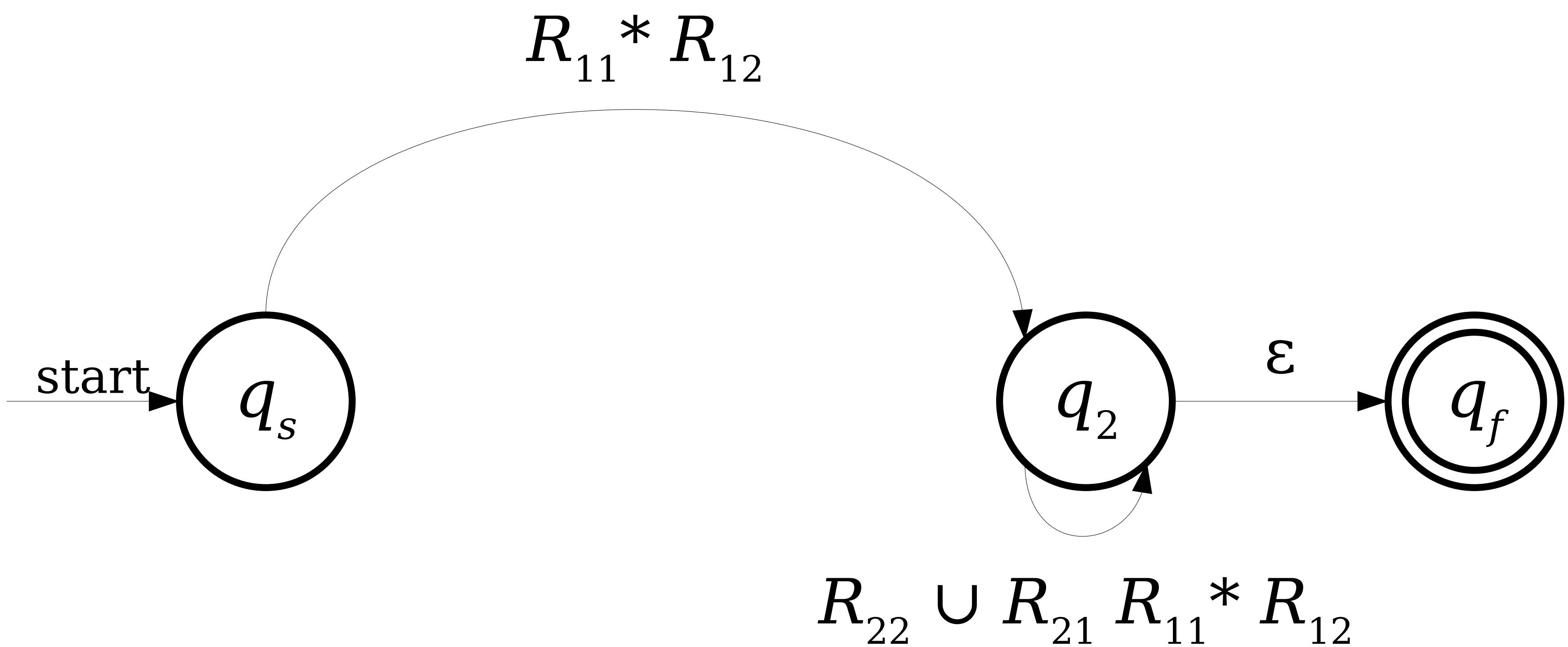



Note: We're using concatenation and Kleene closure in order to skip this state.

# From NFAs to Regular Expressions

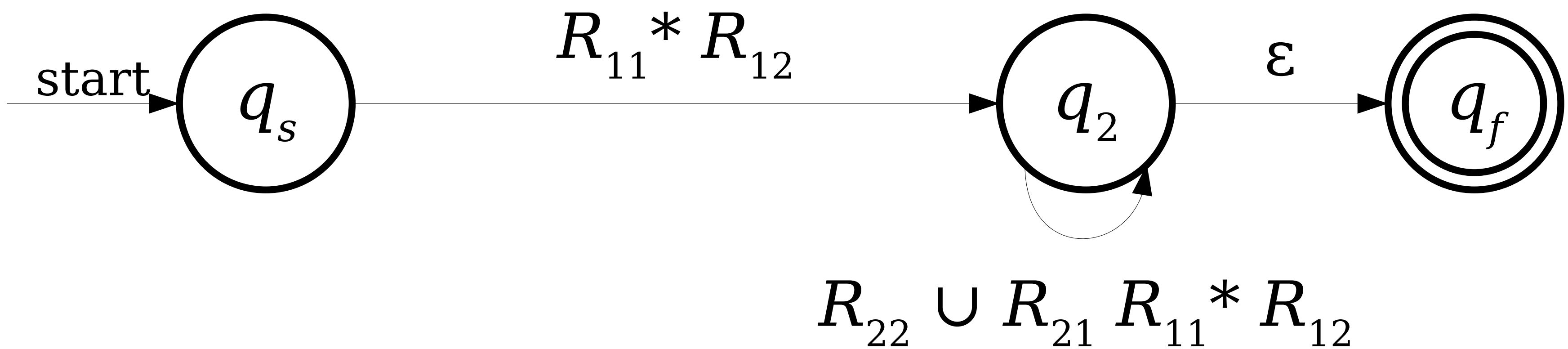


# From NFAs to Regular Expressions

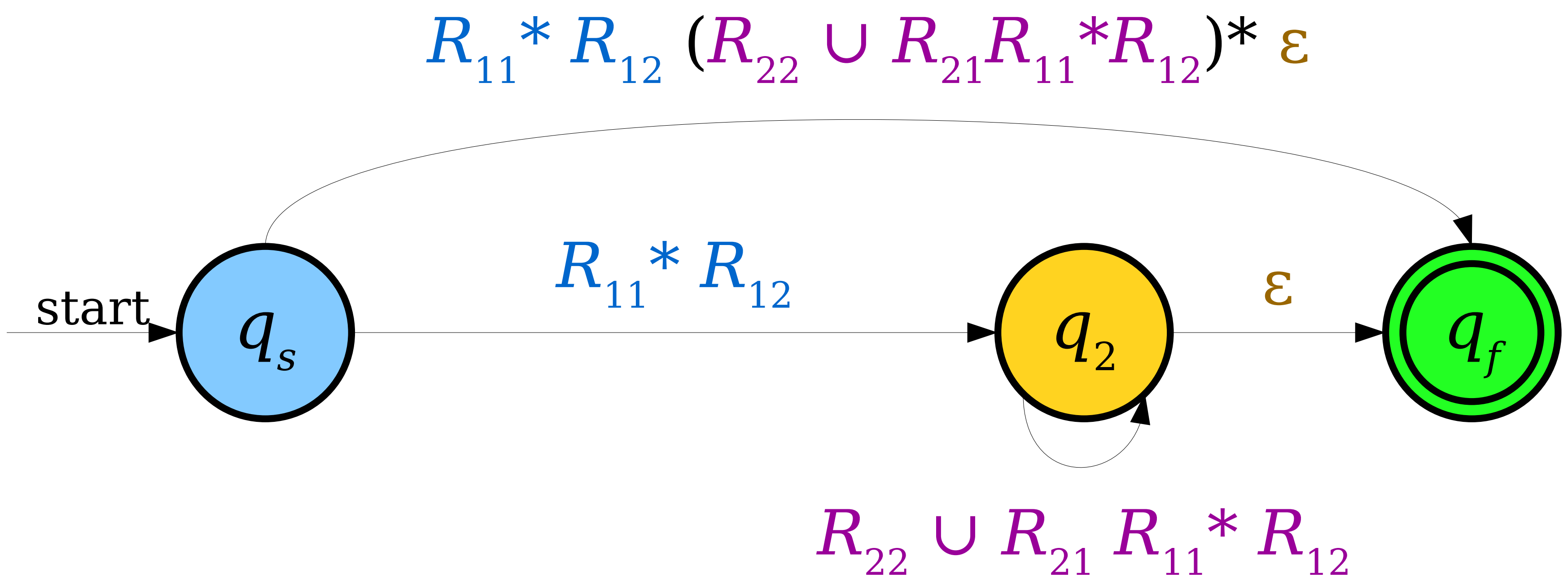



Note: We're using union to combine these transitions together.

# From NFAs to Regular Expressions



# From NFAs to Regular Expressions



# From NFAs to Regular Expressions