Regular Expressions

Recap from Last Time

Regular Languages

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

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

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

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** }

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:

$L^0 = \{\varepsilon\}$

The set containing just the empty string.

Idea: Any string formed by concatenating zero strings together is the empty string.

$L^{n+1} = LL^n$

Idea: Concatenating (n+1) strings together works by concatenating n strings, then concatenating one more.

Question: Why define $L^0 = \{\varepsilon\}$?

Question: What is $\varnothing^0$?

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:

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

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

Question: What is $\emptyset^0$?

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 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:

- 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.

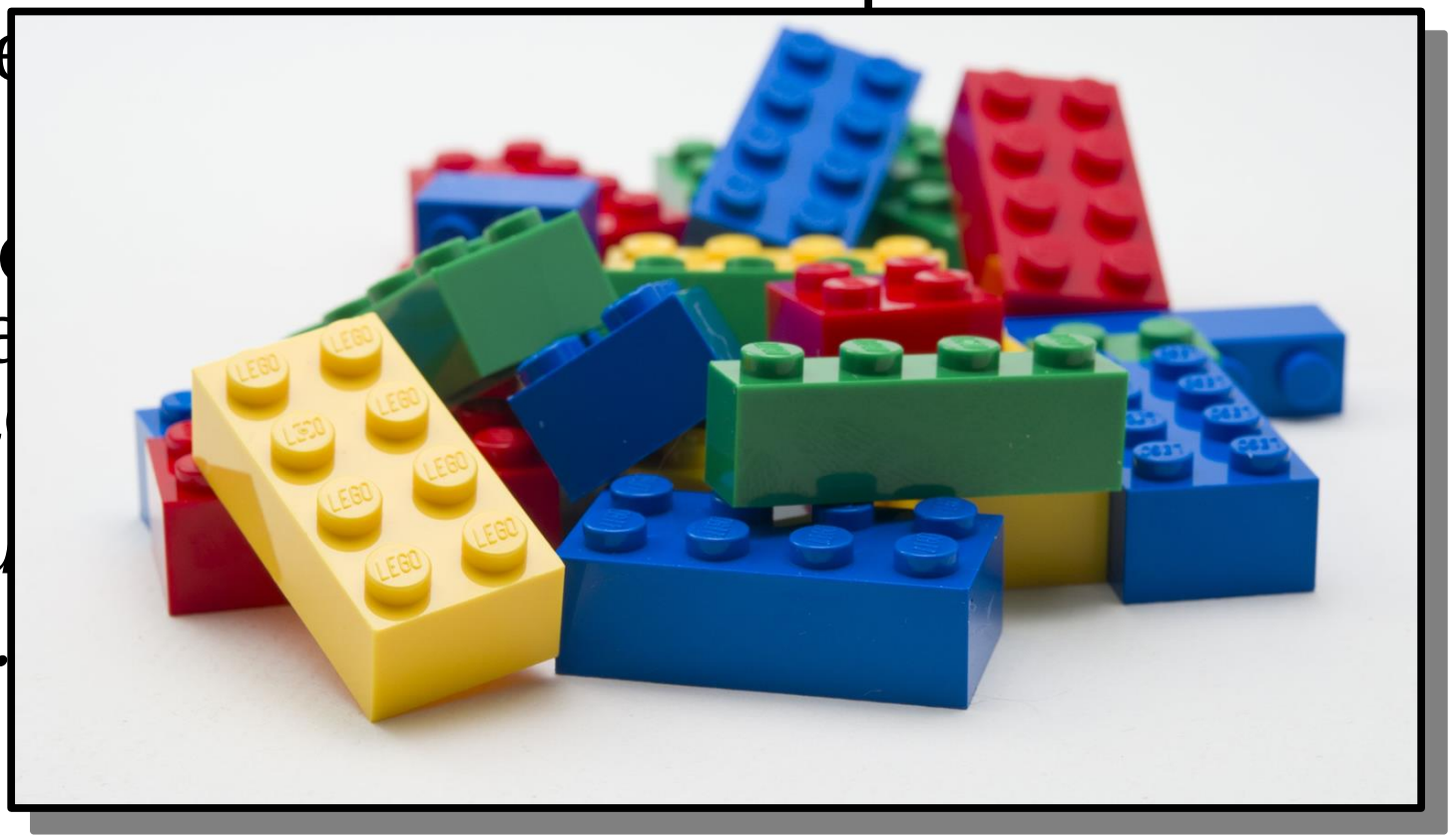
A bottom-up approach to the regular languages.

Constructing Regular Languages

Idea: Build up all regular languages as follows:

- Start with a small set of simple language regular.
- Using cl simple la elaborat

A bottom-u languages.



Regular Expressions

Regular expressions are a way of describing a language via a string representation.

They're used extensively in software systems for string processing and as the basis for tools like grep, flex, sed or awk.

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 **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, $\boldsymbol{R_1 R_2}$ is a regular expression for the *concatenation* of the languages of R_1 and R_2.

If R_1 and R_2 are regular expressions, $\boldsymbol{R_1 \cup R_2}$ 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.

Regular Expression Examples

The regular expression **hello∪goodbye** represents the regular language { **hello**, **goodbye** }.

The regular expression **helloo*** represents the regular language { **hello**, **helloo**, **hellooo**, … }.

The regular expression **(bye)*** represents the regular language { **ε**, **bye**, **byebye**, **byebyebye**, … }.

Operator Precedence

Regular expression operator precedence:

$$(R)$$

$$R*$$

$$R_1 R_2$$

$$R_1 \cup R_2$$

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

Regular Expressions, Formally

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

Formally:

$\mathcal{L}(\boldsymbol{\varepsilon}) = \{\varepsilon\}$

$\mathcal{L}(\boldsymbol{\emptyset}) = \emptyset$

$\mathcal{L}(\mathbf{a}) = \{\mathbf{a}\}$

$\mathcal{L}(R_1 R_2) = \mathcal{L}(R_1)\ \mathcal{L}(R_2)$

$\mathcal{L}(R_1 \cup R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$

$\mathcal{L}(R^*) = \mathcal{L}(R)^*$

$\mathcal{L}(\mathbf{(}R\mathbf{)}) = \mathcal{L}(R)$

Worthwhile activity: Apply this recursive definition to

a(b∪c)((d))

and see what you get.

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 $\}$.

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

Designing Regular Expressions

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

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

(a ∪ b)*aa(a ∪ b)*

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 $\}$.

(a ∪ b)*aa(a ∪ b)*

bbabbbbaabab

aaaa

bbbbbabbbbbaabbbbb

Designing Regular Expressions

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

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

Σ*aaΣ*

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

Designing Regular Expressions

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

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

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

Designing Regular Expressions

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

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

ΣΣΣΣ

Designing Regular Expressions

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

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

$\Sigma\Sigma\Sigma\Sigma$

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

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

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

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

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

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

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

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

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

Designing Regular Expressions

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

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

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

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

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

b*(a ∪ ε)b*

bbbbabbb

bbbbbb

abbb

a

Designing Regular Expressions

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

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

b*(a ∪ ε)b*

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.

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

Let's make a regex for email addresses.

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

Let's make a regex for email addresses.

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

Let's make a regex for email addresses.

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

Let's make a regex for email addresses.

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

cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

A More Elaborate Design

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

Let's make a regex for email addresses.

$$\mathbf{a}^+ \quad (\mathbf{.a}^+)^* \quad \mathbf{@} \quad \mathbf{a}^+ \qquad (\mathbf{.a}^+)^+$$

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.

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

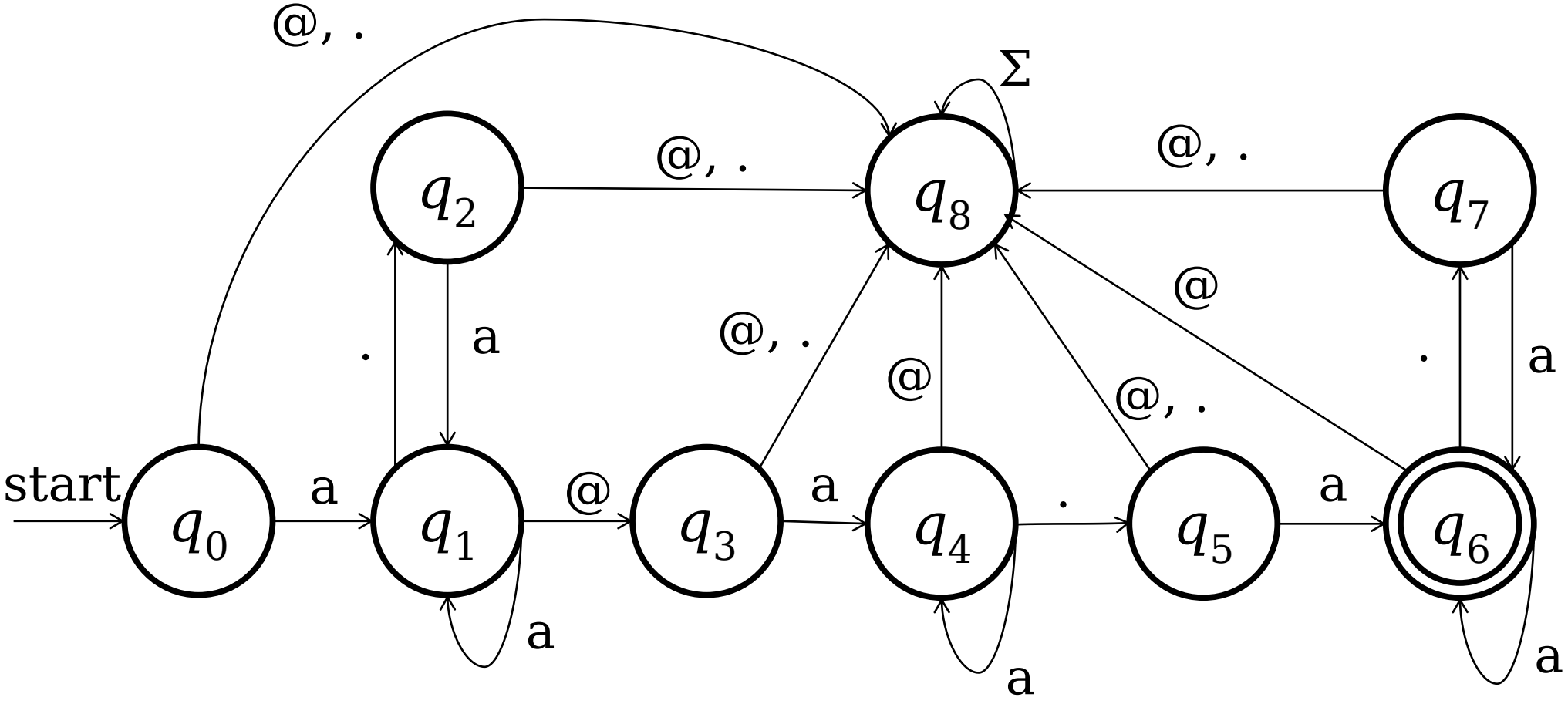
cs103@cs.stanford.edu

first.middle.last@mail.site.org

dot.at@dot.com

For Comparison

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



Shorthand Summary

R^n is shorthand for $RR \ldots R$ (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

- Midterm is graded, results are being released after lecture.
- Solutions will be posted this afternoon.
- The solutions set will include common mistakes and a breakdown of the grades so you can measure where you are.

Problem Set

- Problem Set Four is due at 11:59PM on Thursday.
- You can now answer the regex question, so all the material needed for this assignment has been covered.
  - Regex tool will be online after lecture today.
- This problem set is huge: come by office hours or ask questions on Campuswire!

Have a question?

- Tradition from CS 103 classes of yore.
- Ask anonymous questions on any topic and vote on other student questions. Top voted questions will get answered in Friday's lecture.
- Go to sli.do and input code G517.

Back to CS103!

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!

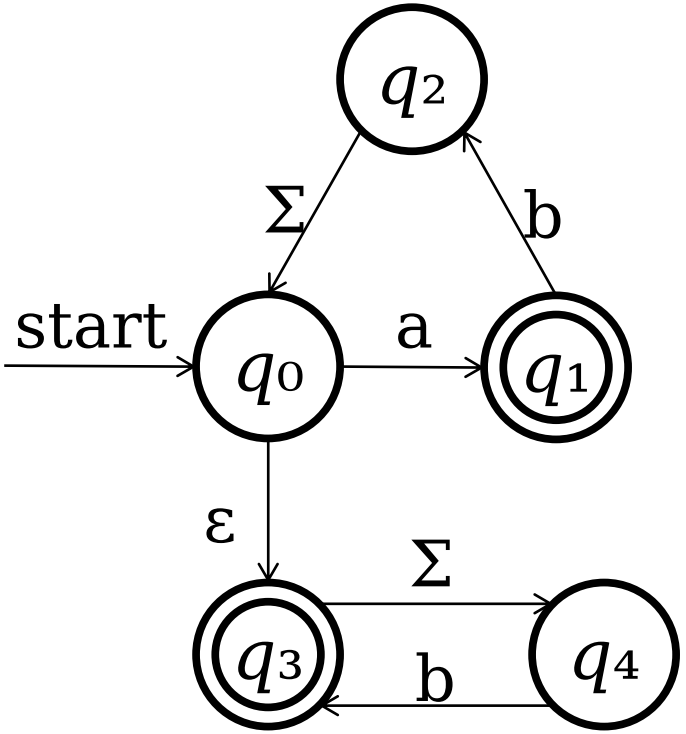
The Power of Regular Expressions

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

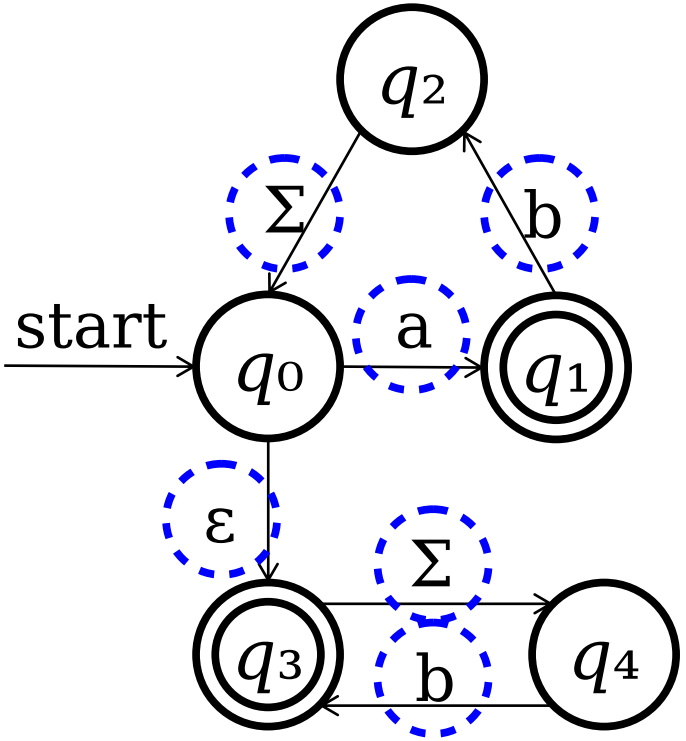
This is not obvious!

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

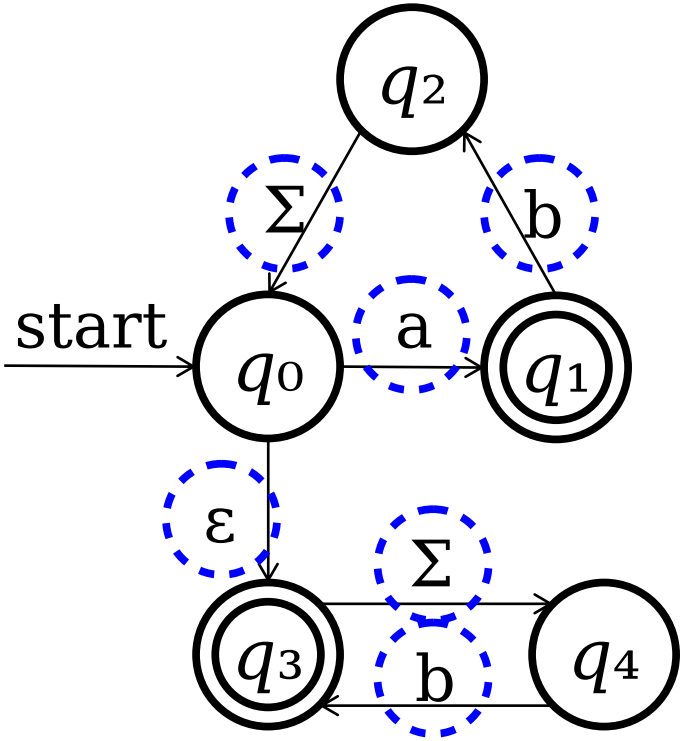
Generalizing NFAs



Generalizing NFAs

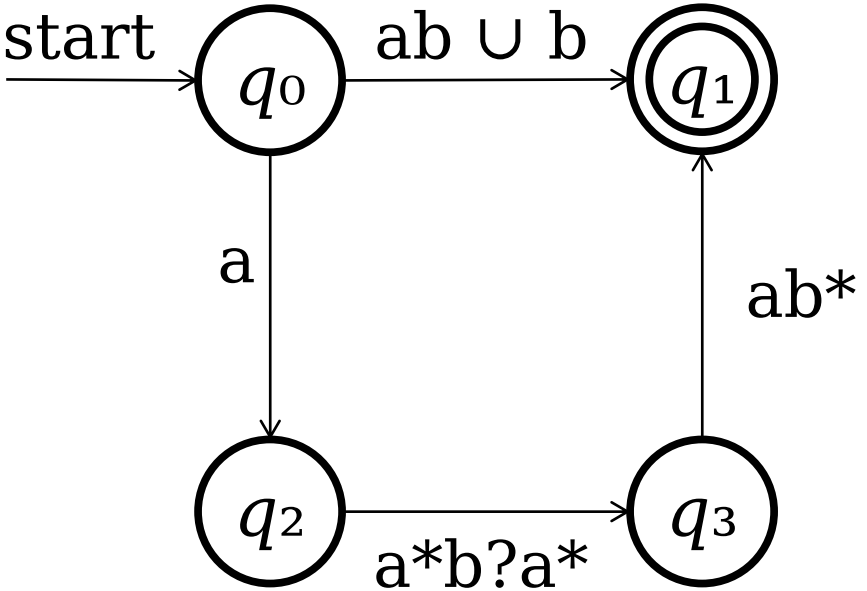


Generalizing NFAs



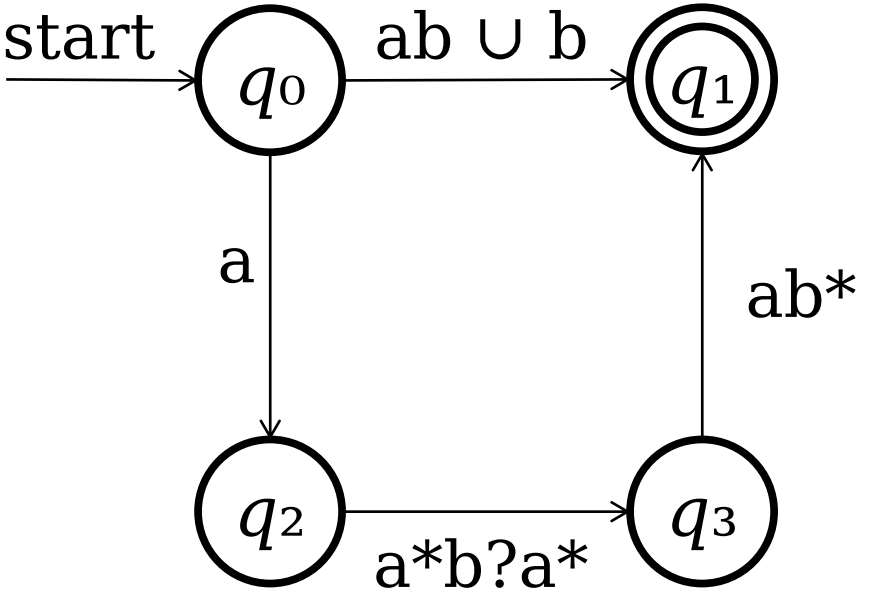
These are all regular expressions!

Generalizing NFAs



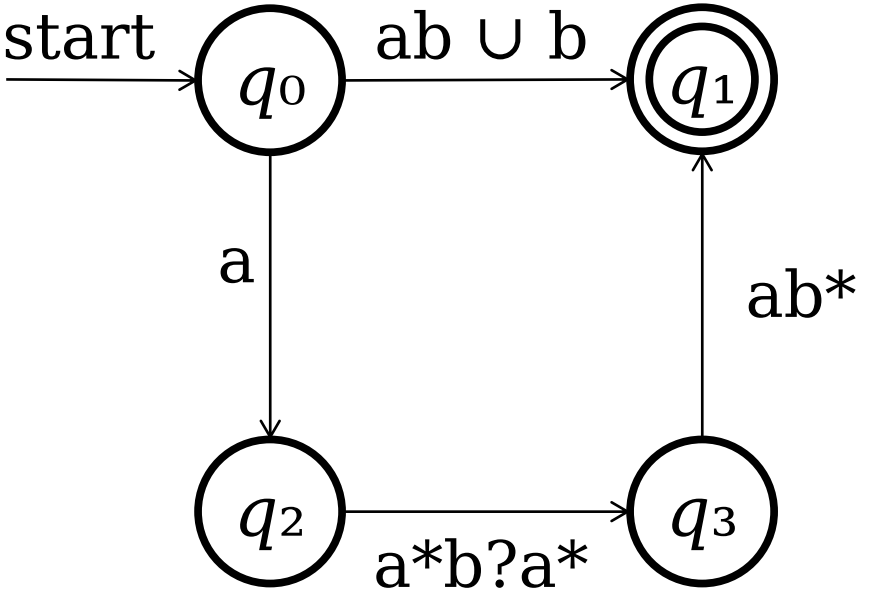
Note: Actual NFAs aren't allowed to have transitions like these. This is just a thought experiment.

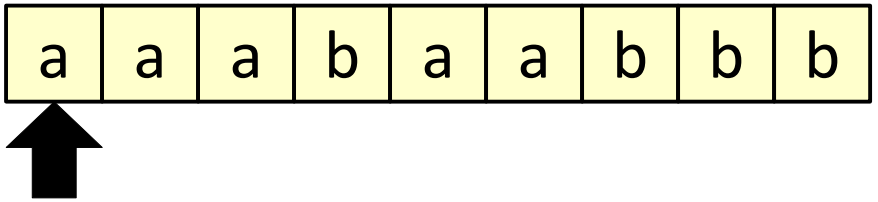
Generalizing NFAs



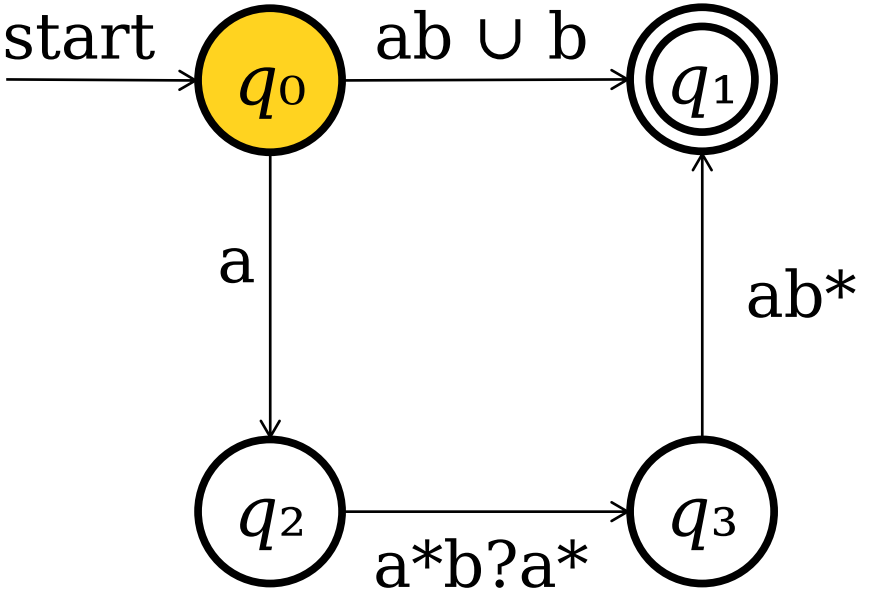
a	a	a	b	a	a	b	b	b

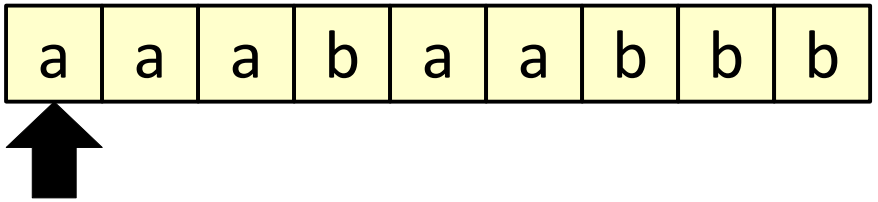
Generalizing NFAs



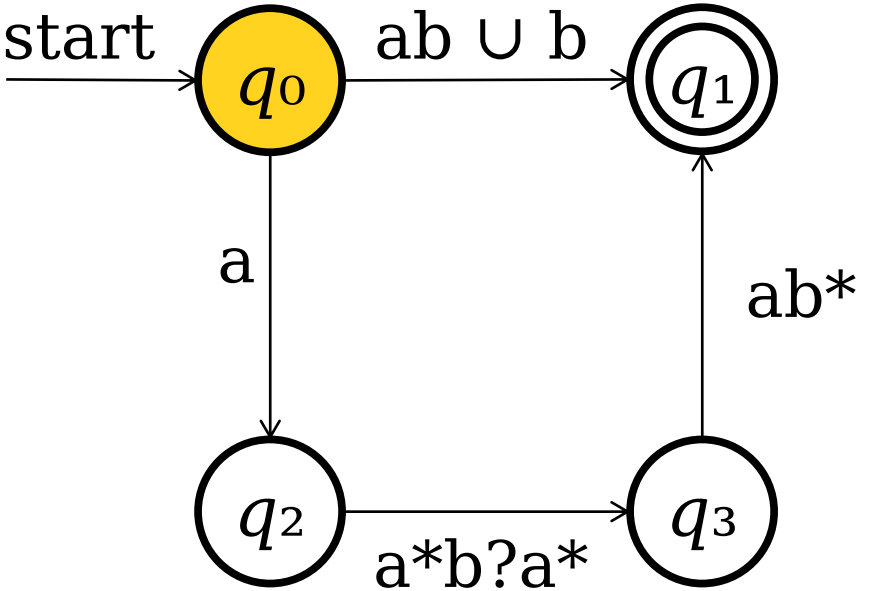


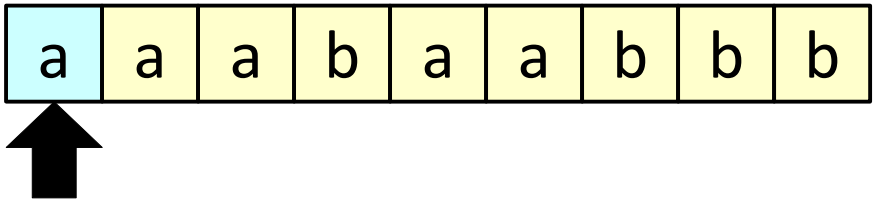
Generalizing NFAs



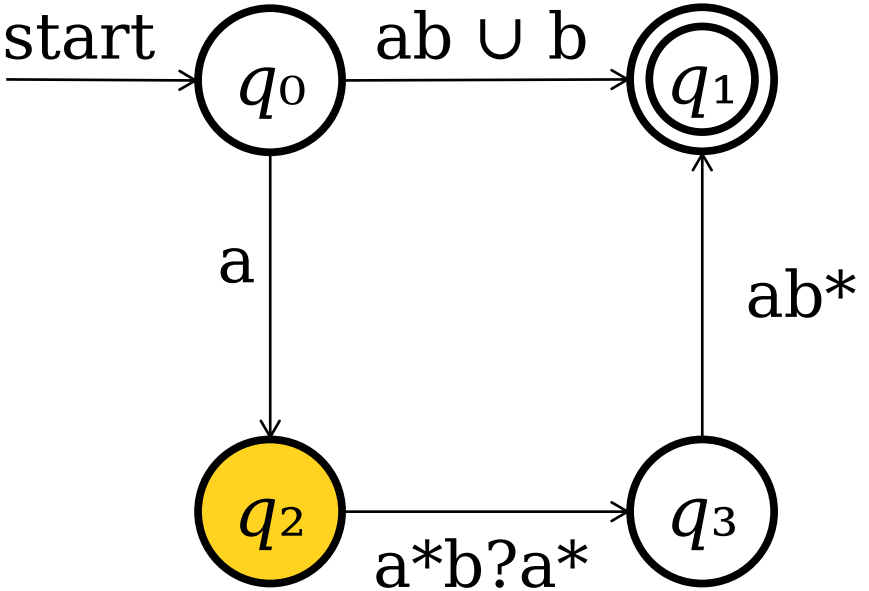


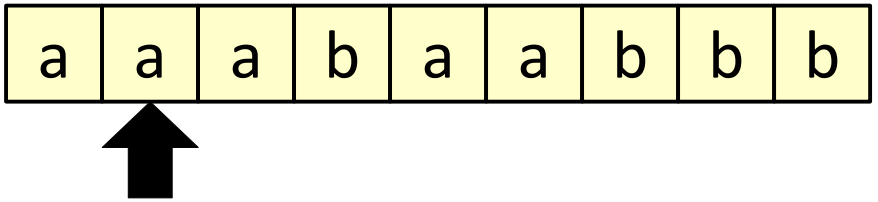
Generalizing NFAs



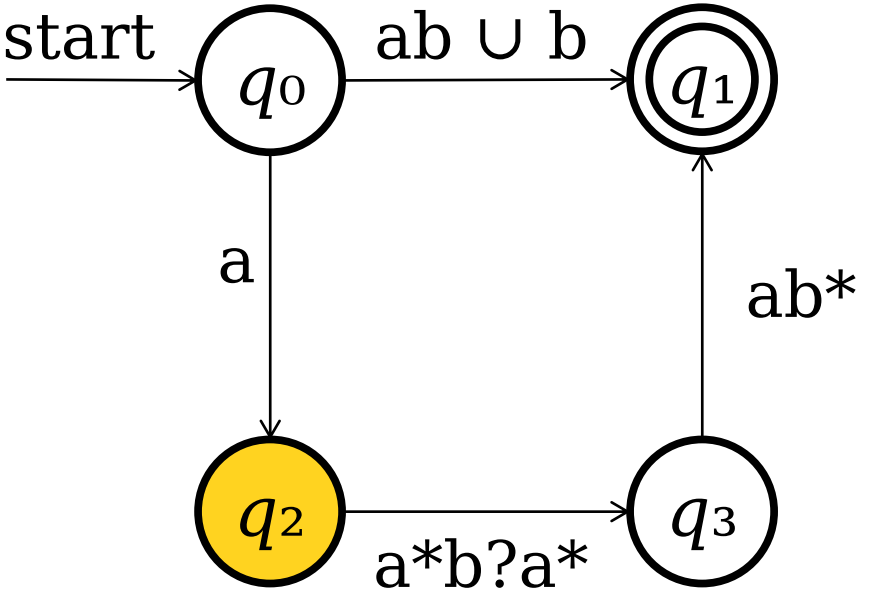


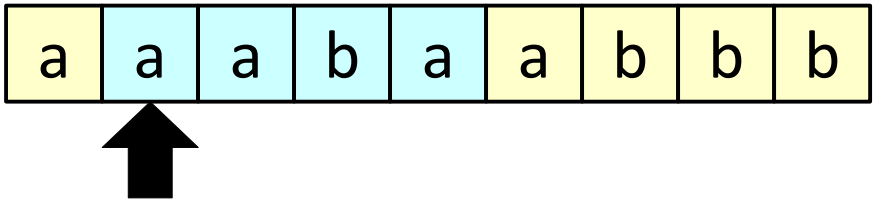
Generalizing NFAs



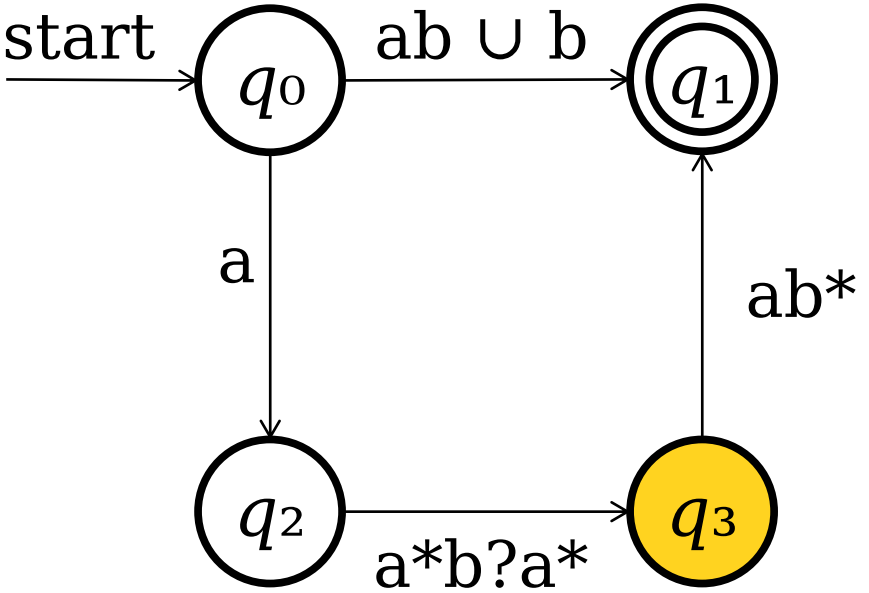


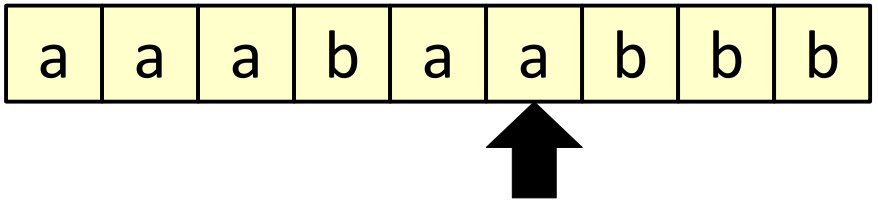
Generalizing NFAs



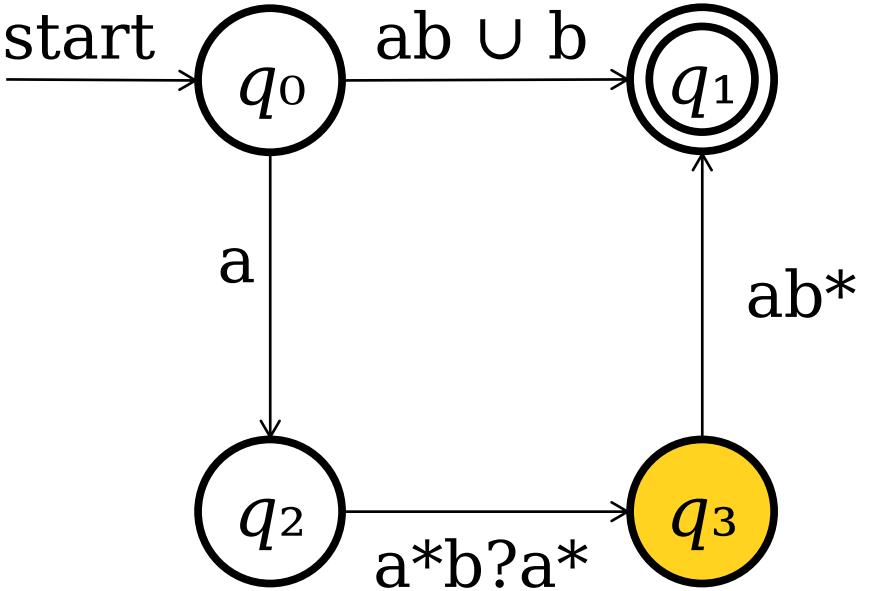


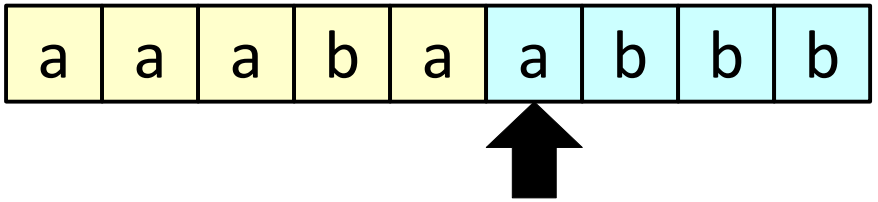
Generalizing NFAs



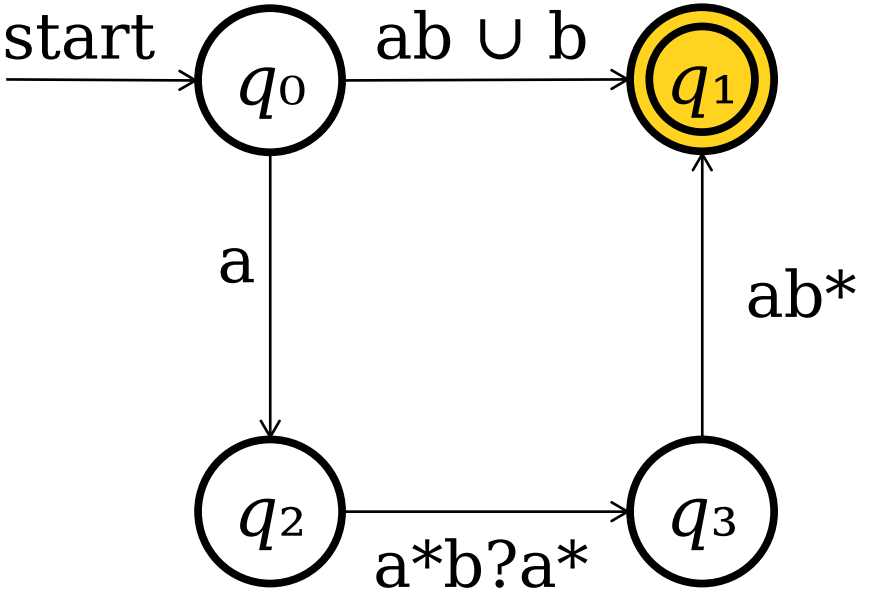


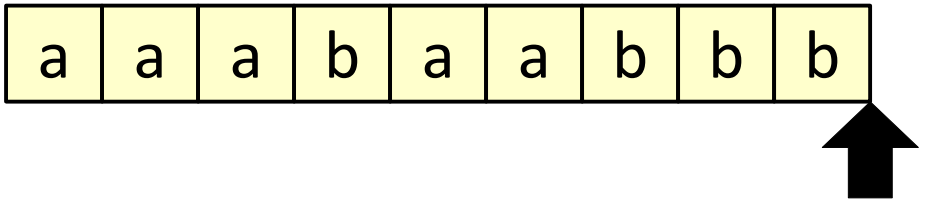
Generalizing NFAs





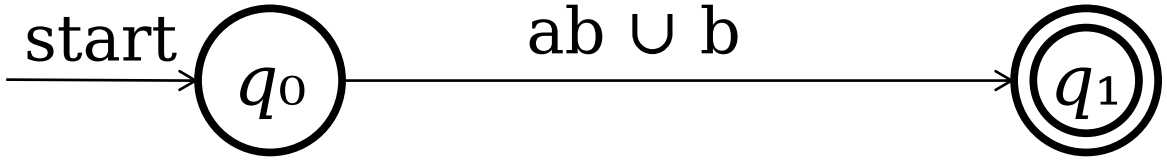
Generalizing NFAs



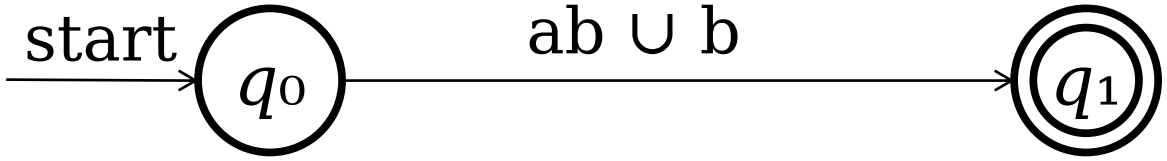


Key Idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions.

Generalizing NFAs

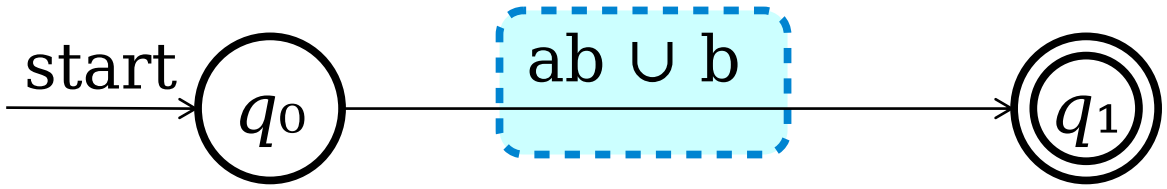


Generalizing NFAs



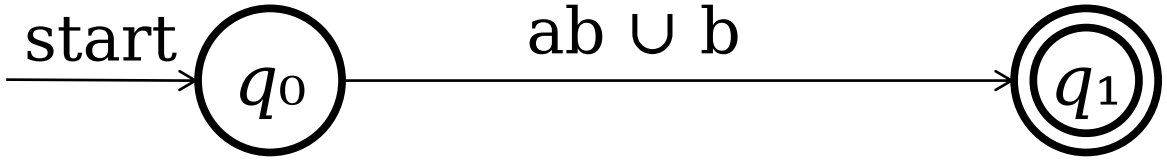
Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs

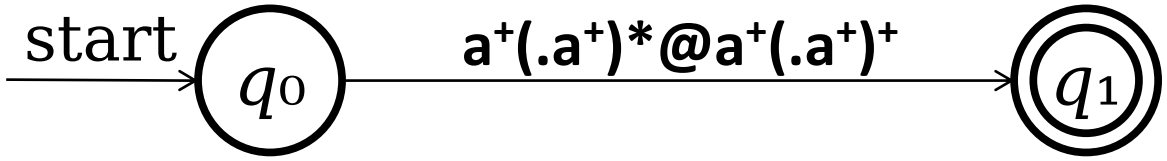


Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs

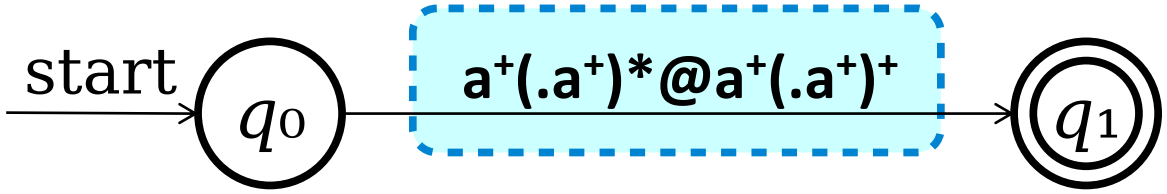


Generalizing NFAs



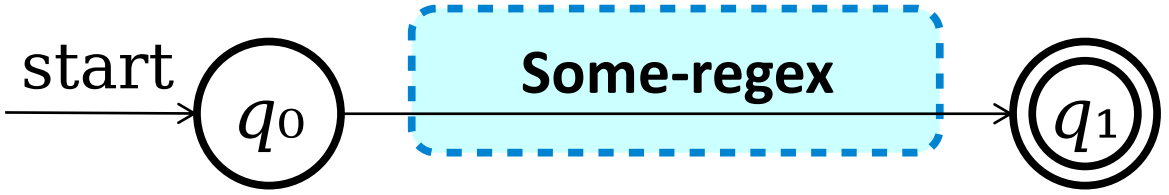
Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs



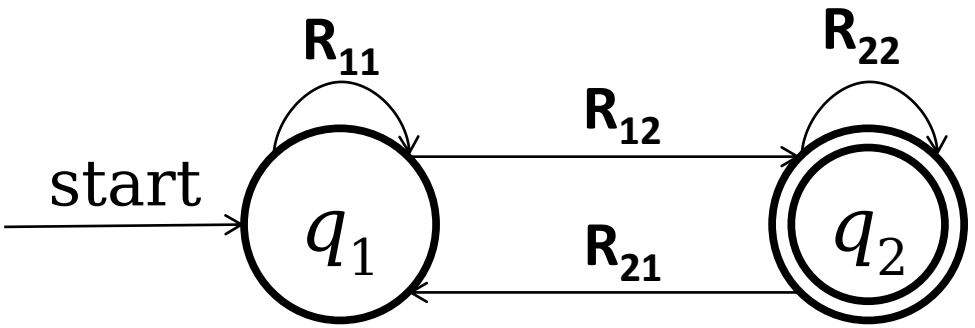
Is there a simple regular expression for the language of this generalized NFA?

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

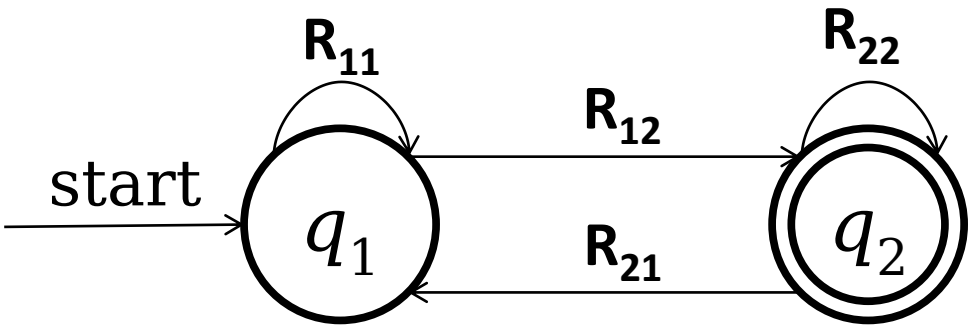


...then we can easily read off a regular expression for the original NFA.

From NFAs to Regular Expressions

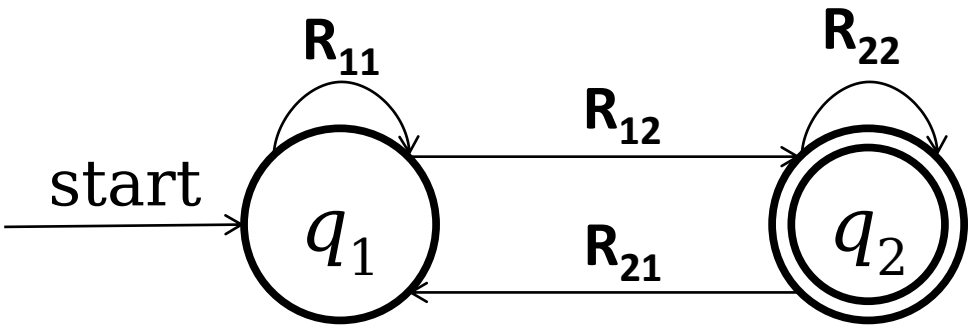


From NFAs to Regular Expressions



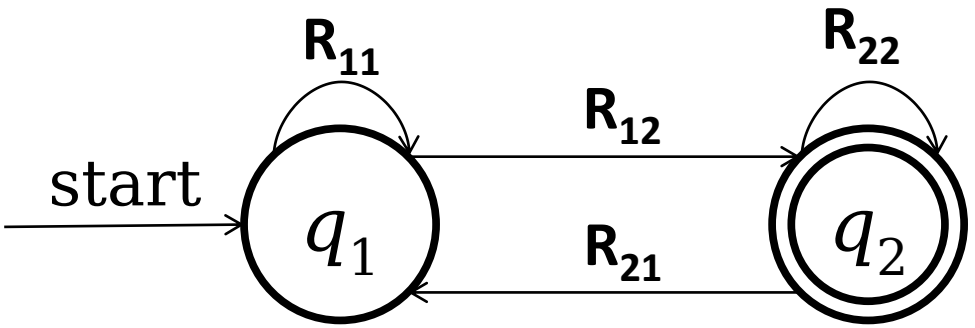
Here, R_{11}, R_{12}, R_{21}, and R_{22} are arbitrary regular expressions.

From NFAs to Regular Expressions



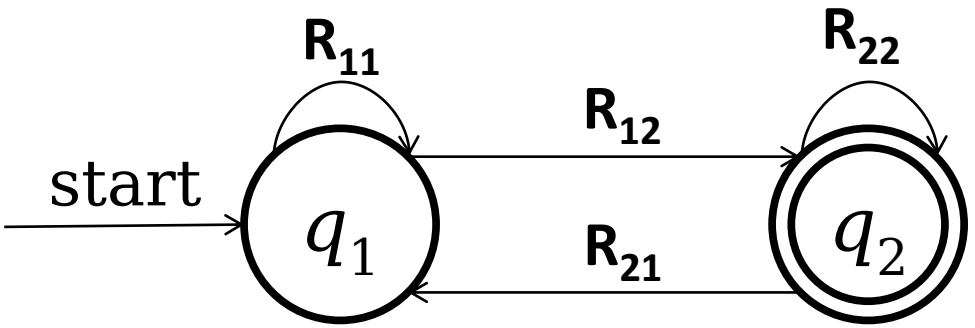
Question: Can we get a clean regular expression from this NFA?

From NFAs to Regular Expressions



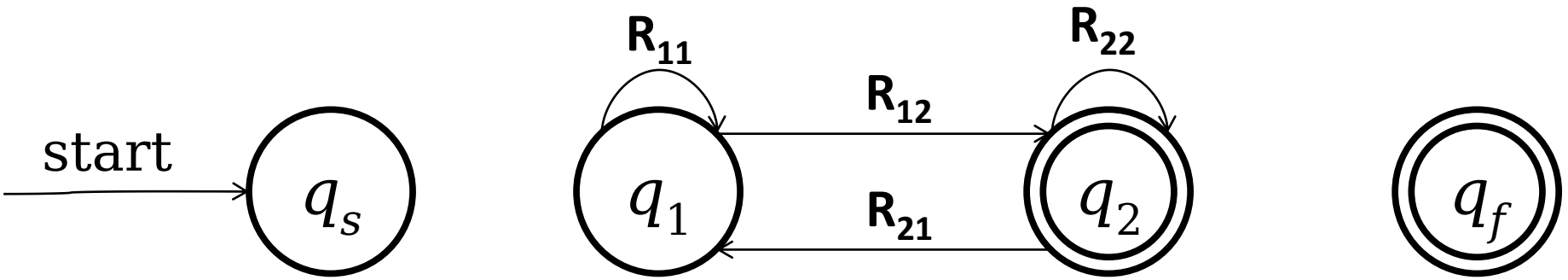
Key Idea 3: Somehow transform this NFA so that it looks like this:

From NFAs to Regular Expressions

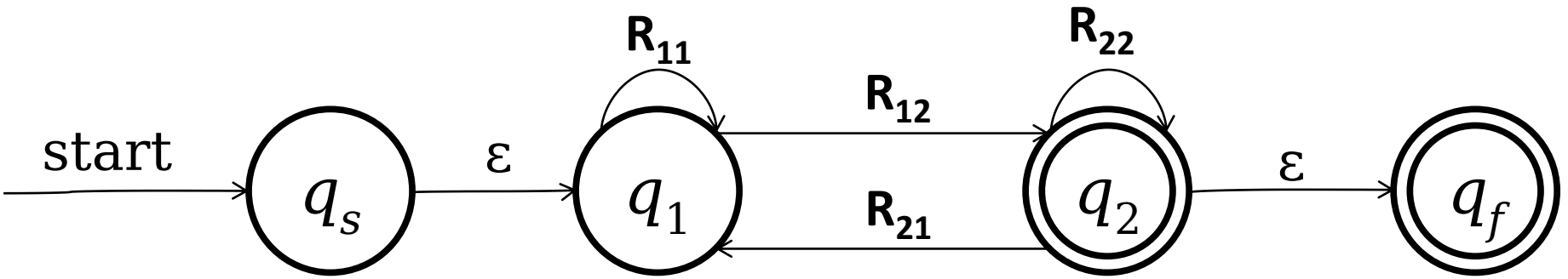


The first step is going to be a bit weird...

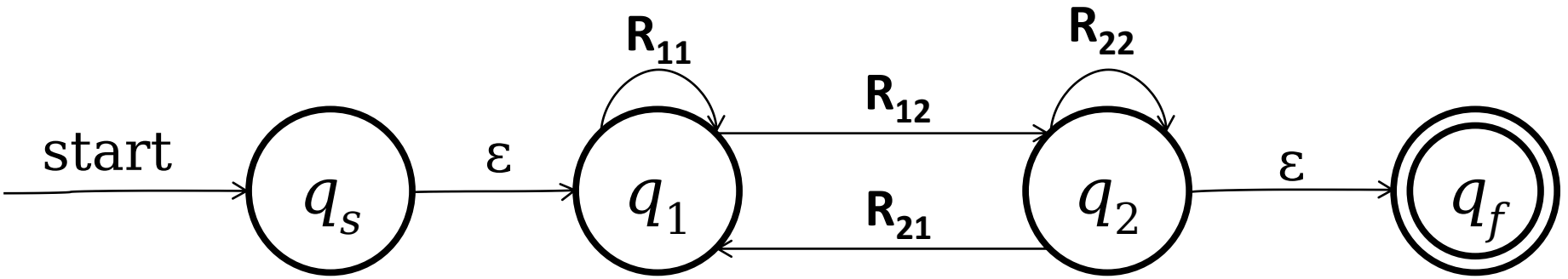
From NFAs to Regular Expressions



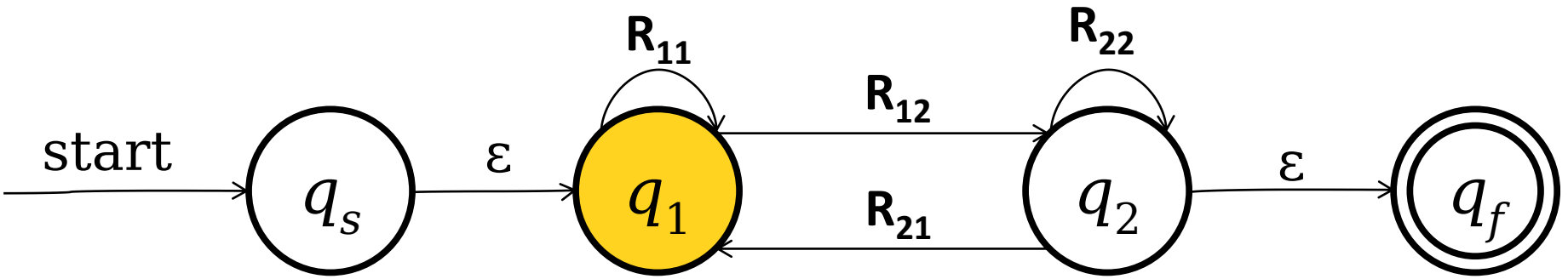
From NFAs to Regular Expressions



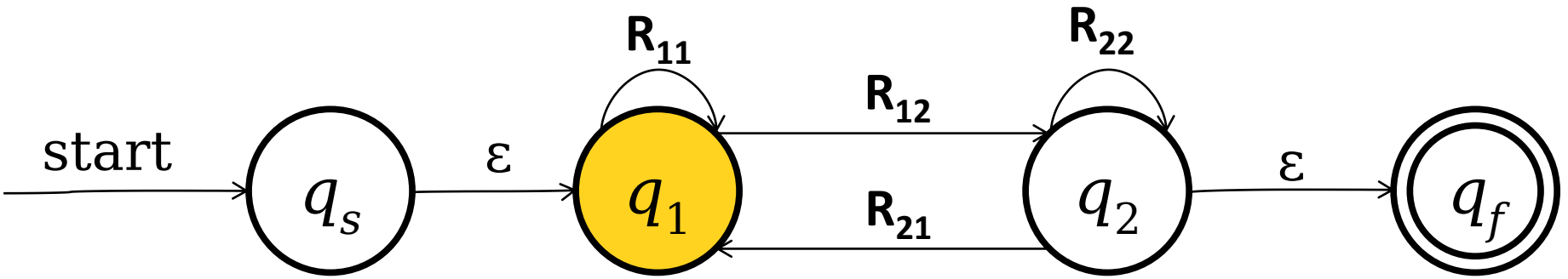
From NFAs to Regular Expressions



From NFAs to Regular Expressions

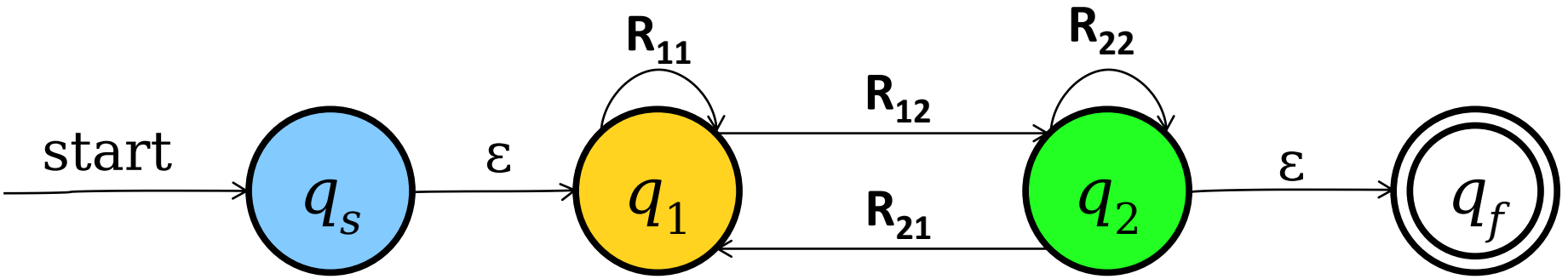


From NFAs to Regular Expressions

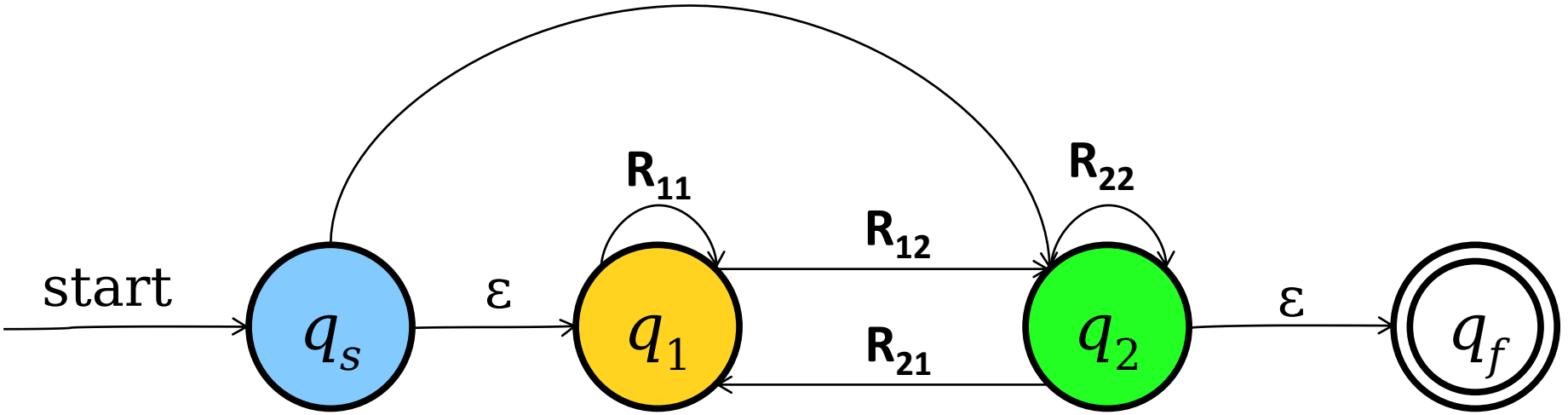


Could we eliminate this state from the NFA?

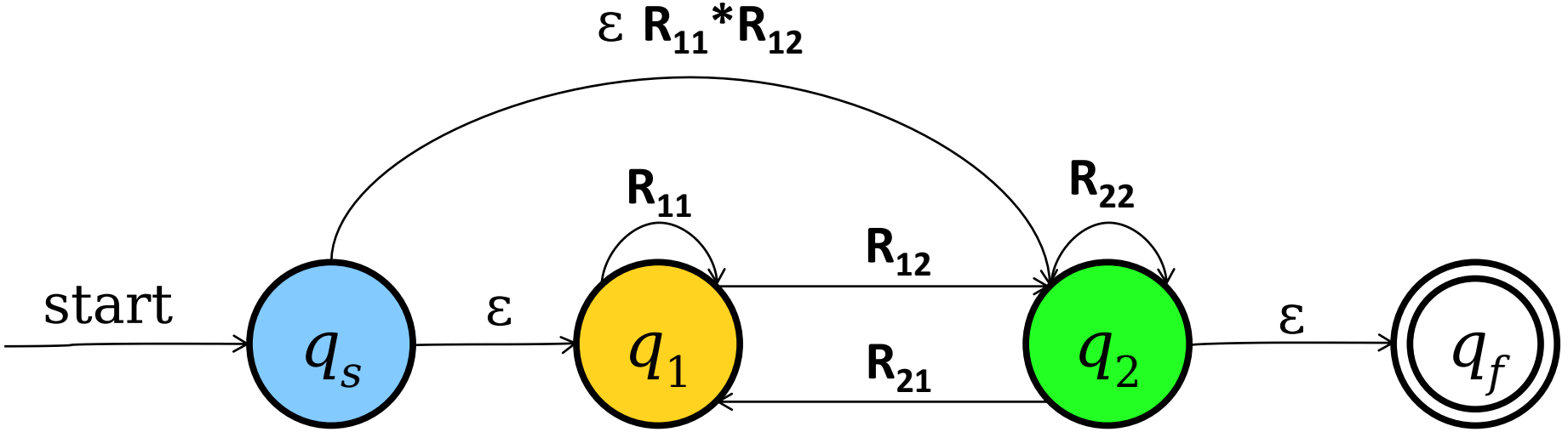
From NFAs to Regular Expressions



From NFAs to Regular Expressions

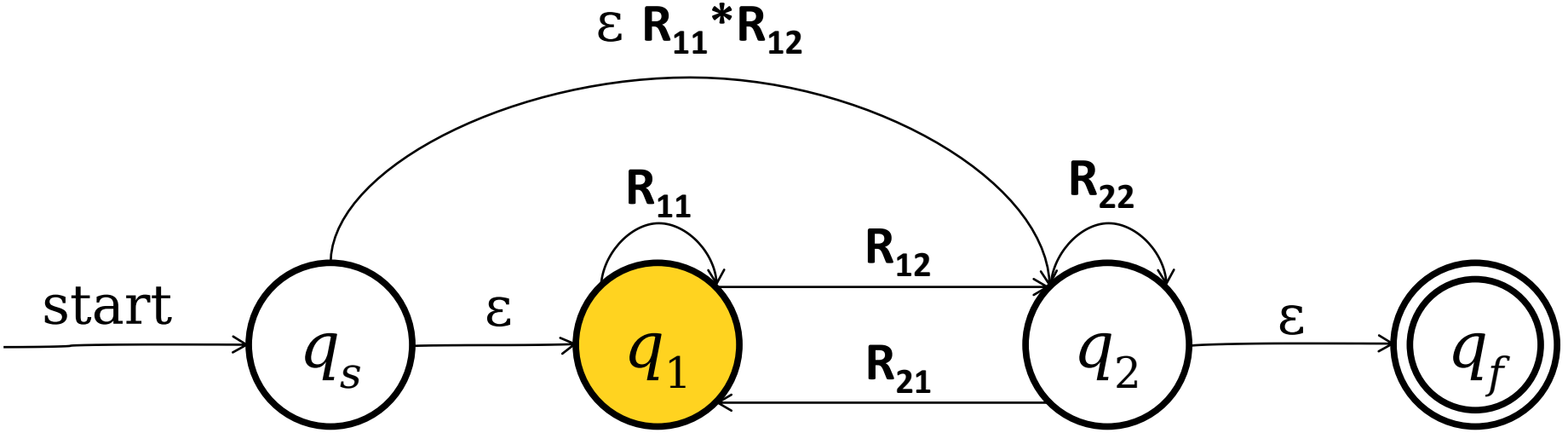


From NFAs to Regular Expressions

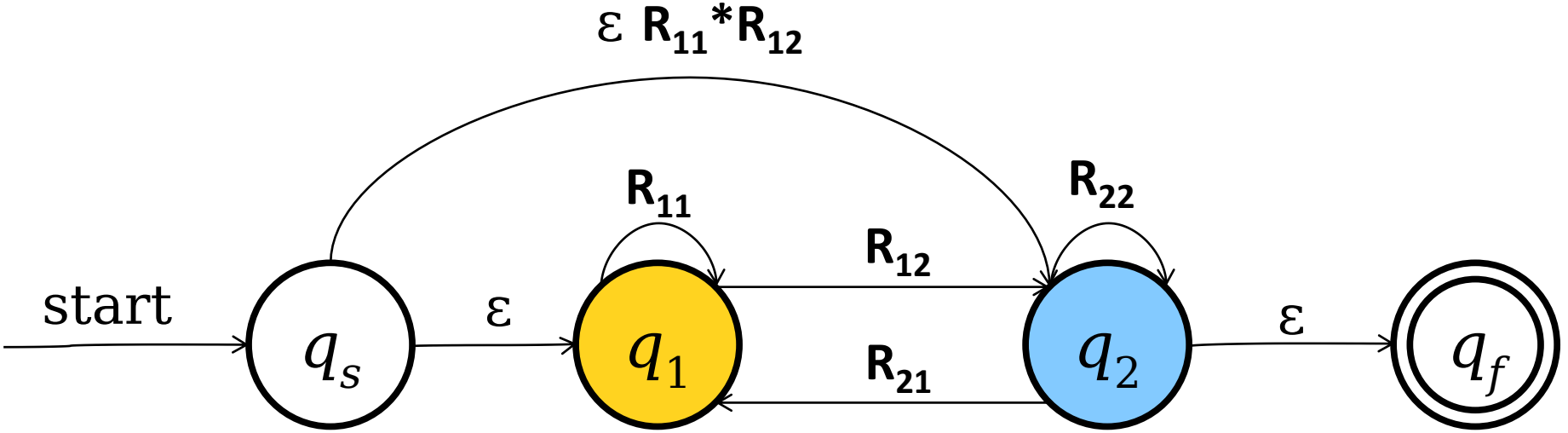


Note: We're using **concatenation** and **Kleene closure** in order to skip this state.

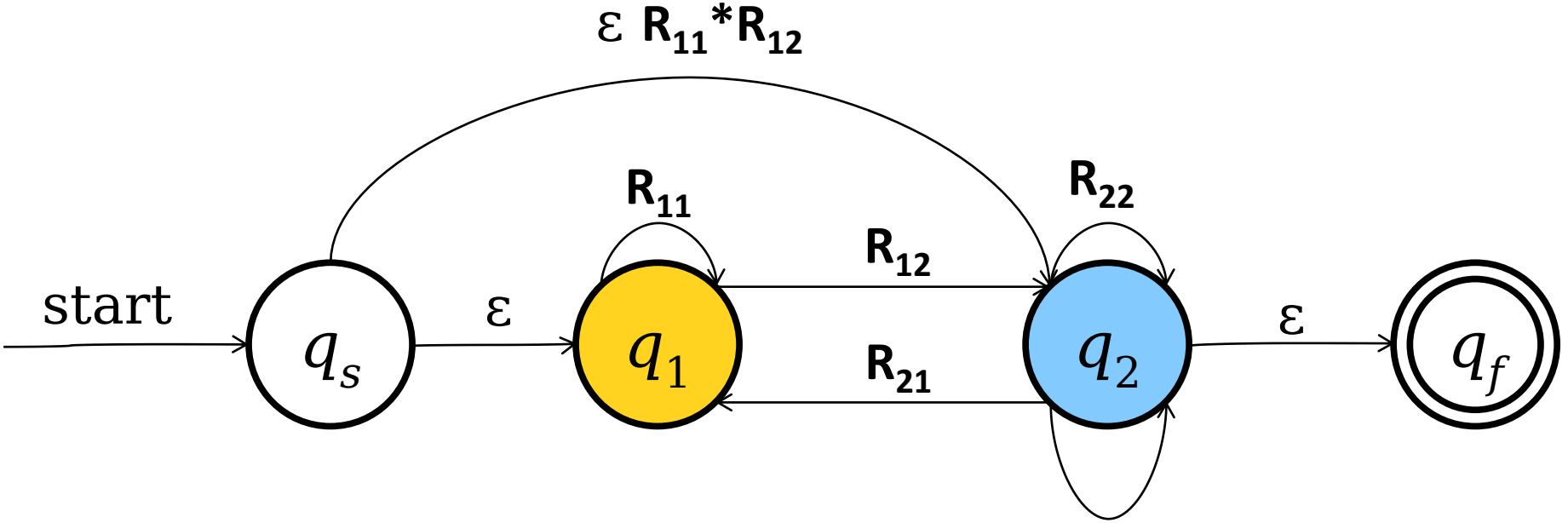
From NFAs to Regular Expressions



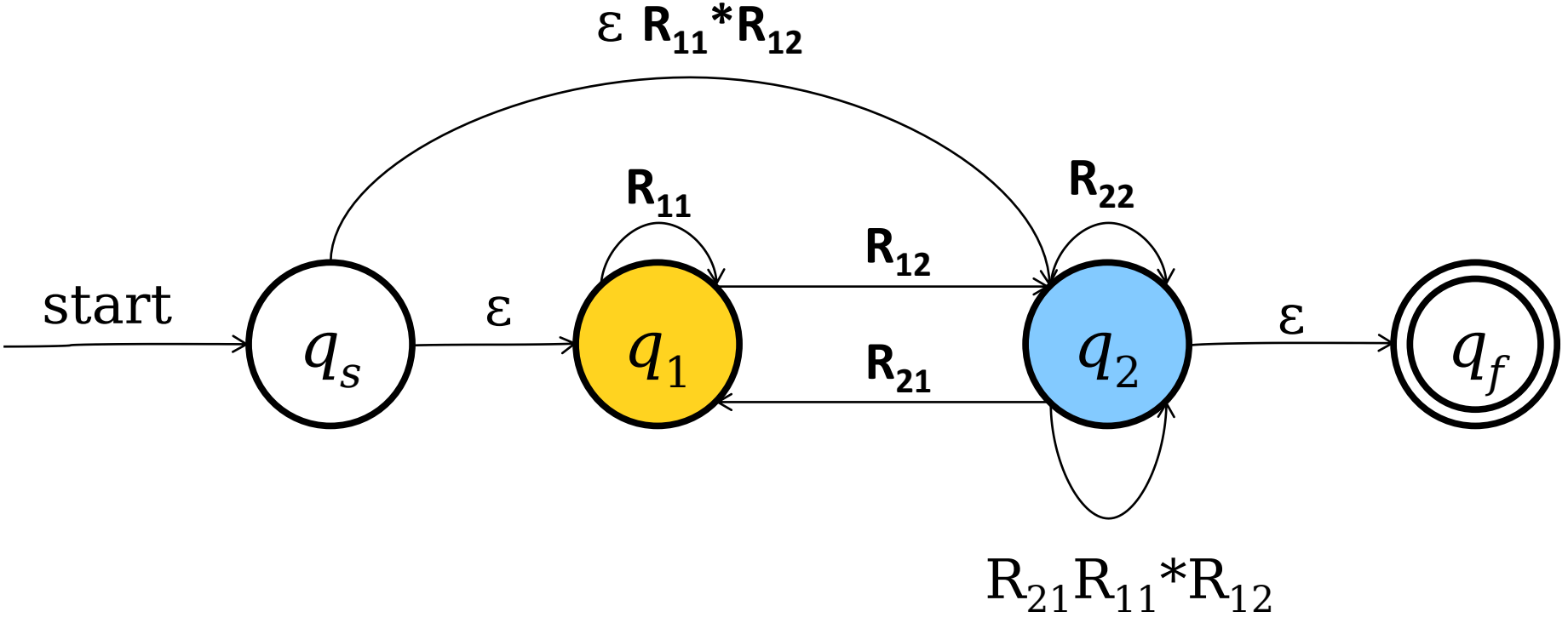
From NFAs to Regular Expressions



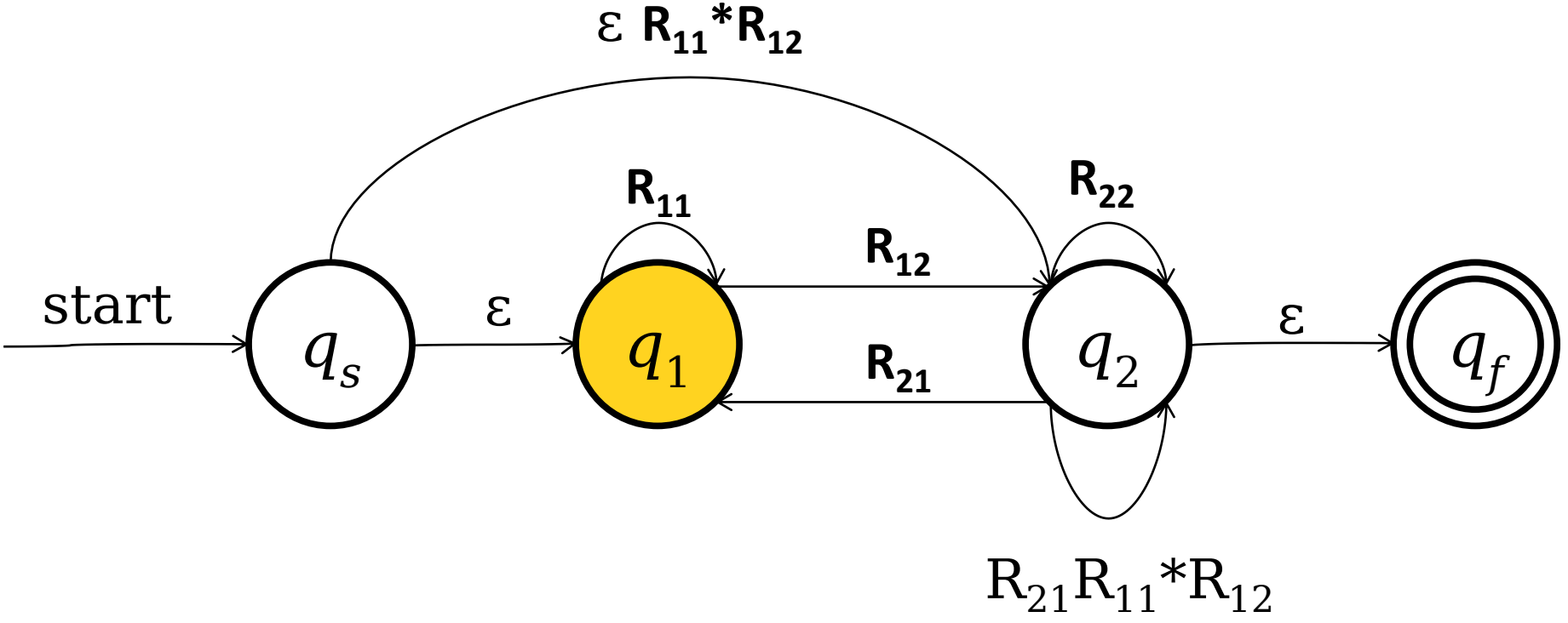
From NFAs to Regular Expressions



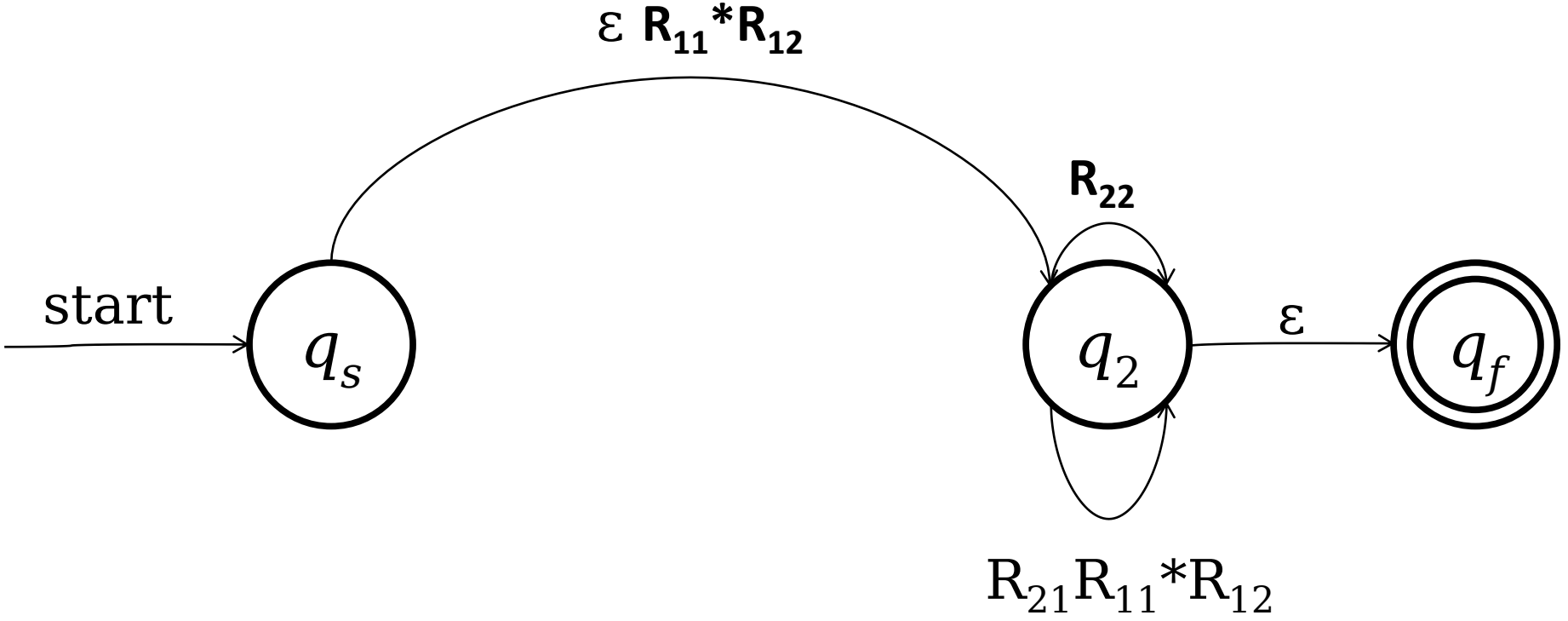
From NFAs to Regular Expressions



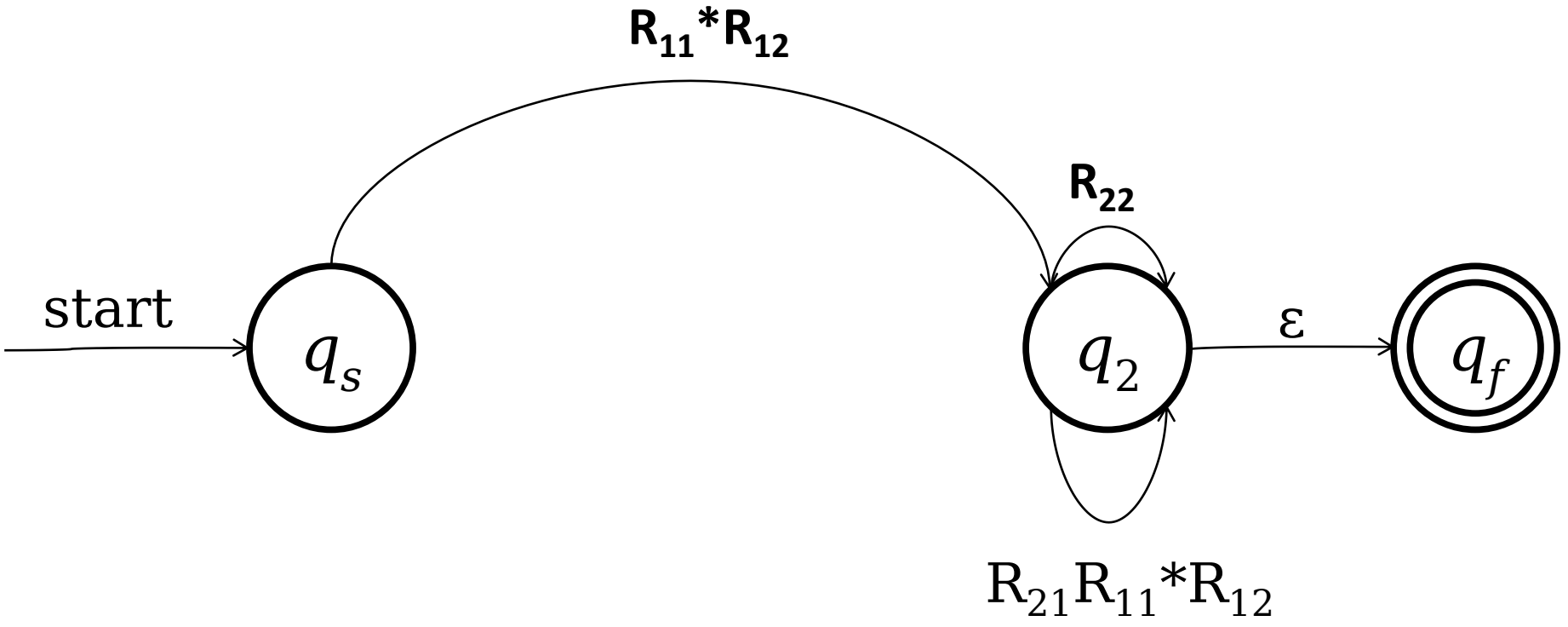
From NFAs to Regular Expressions



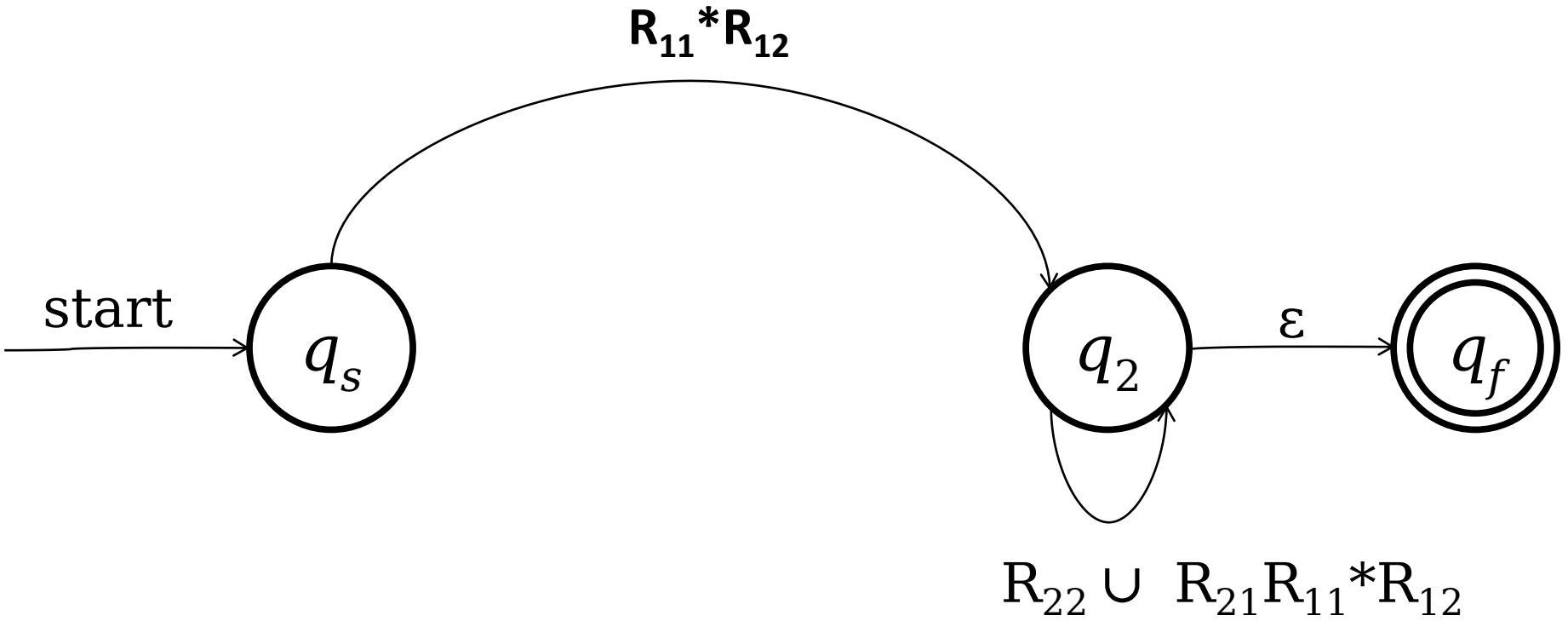
From NFAs to Regular Expressions



From NFAs to Regular Expressions

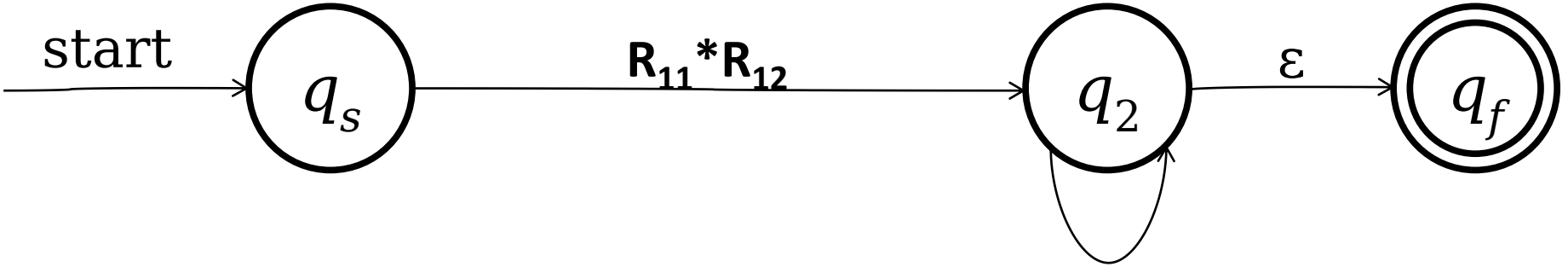


From NFAs to Regular Expressions

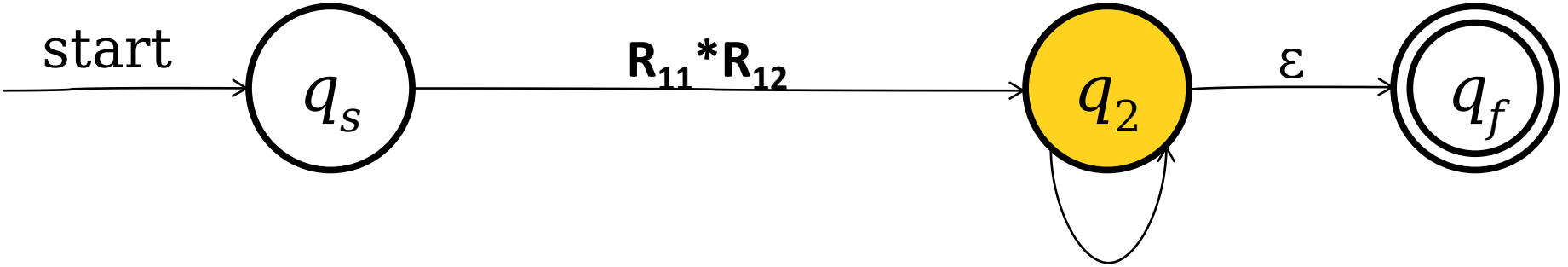


Note: We're using **union** to combine these transitions together.

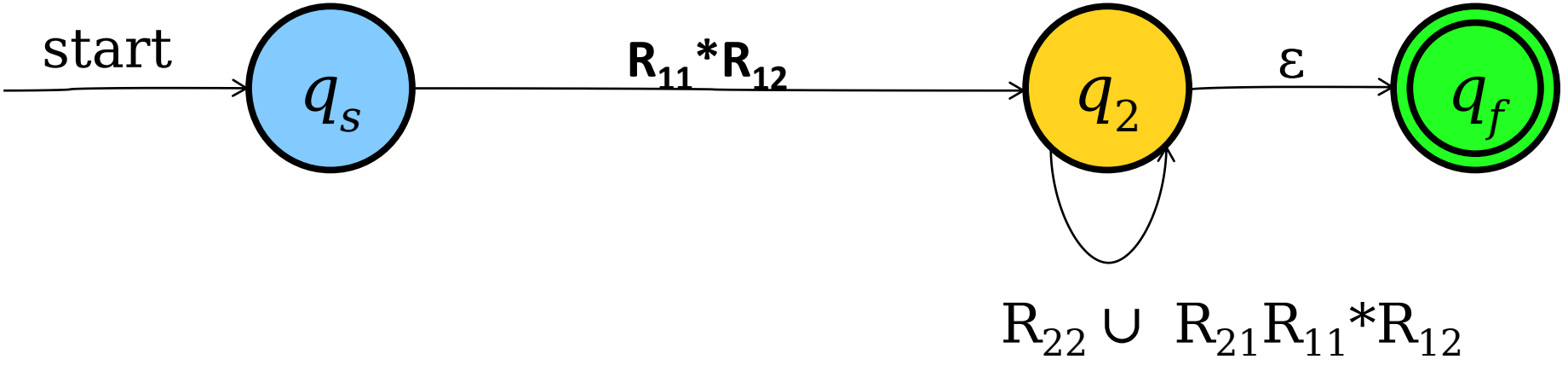
From NFAs to Regular Expressions



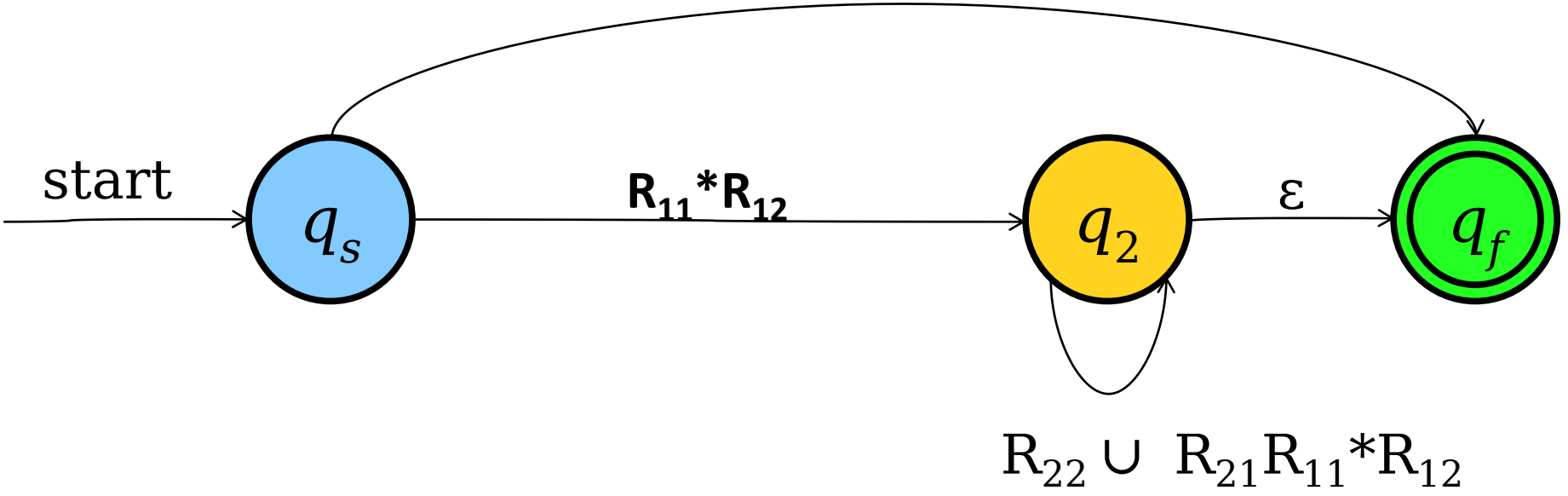
From NFAs to Regular Expressions



From NFAs to Regular Expressions

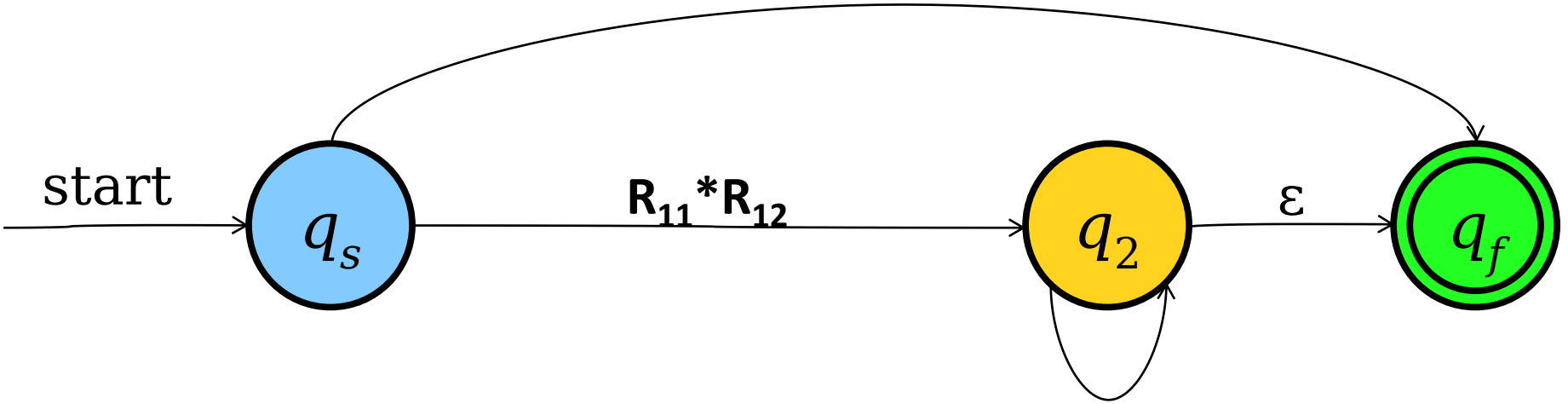


From NFAs to Regular Expressions

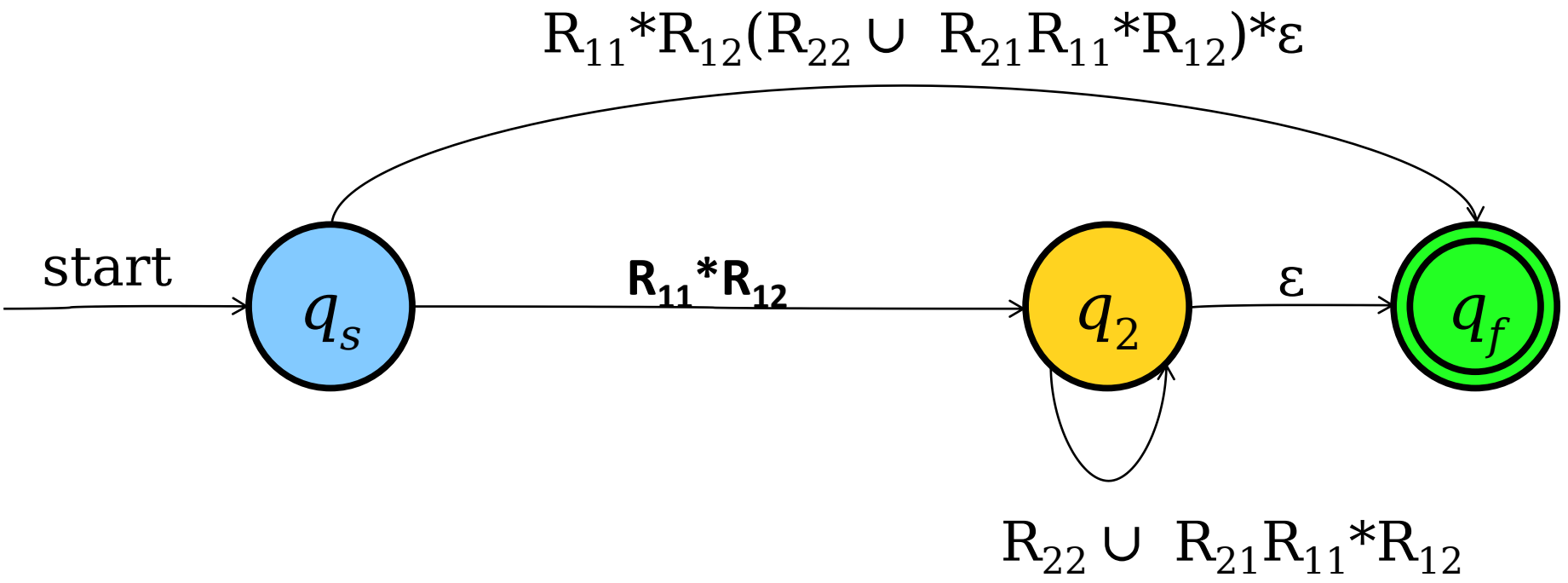


From NFAs to Regular Expressions

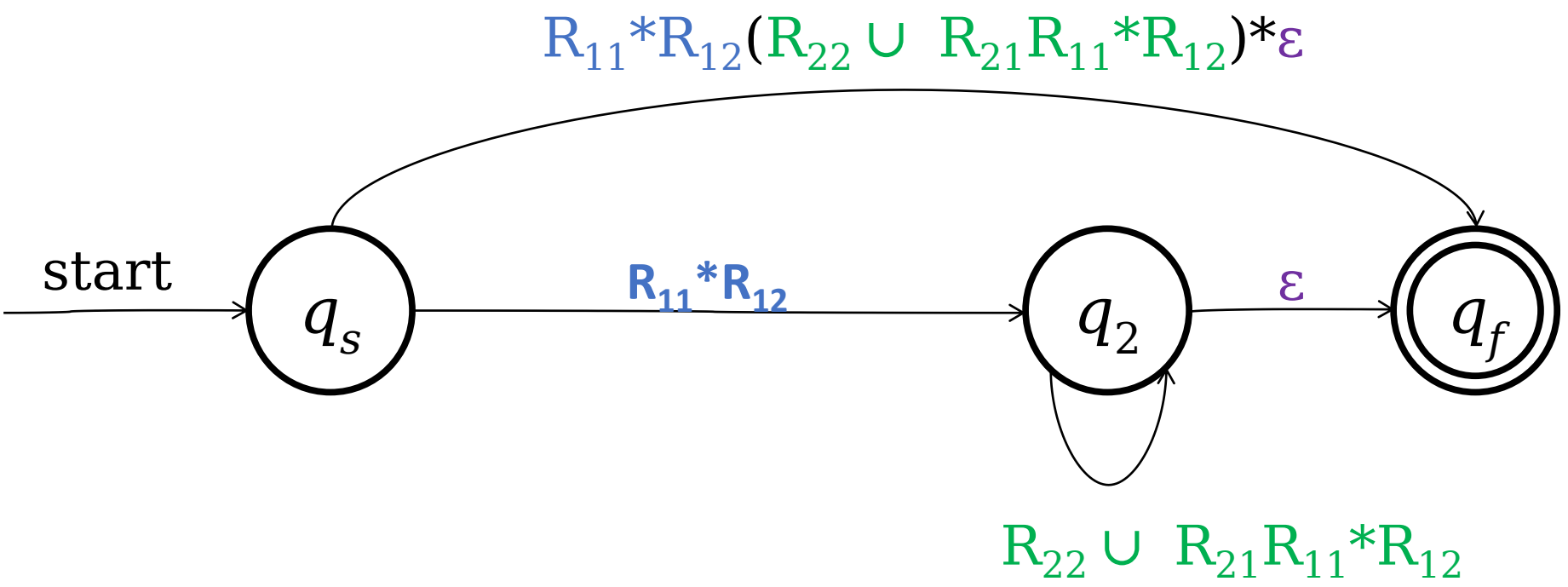
What should we put on this transition?



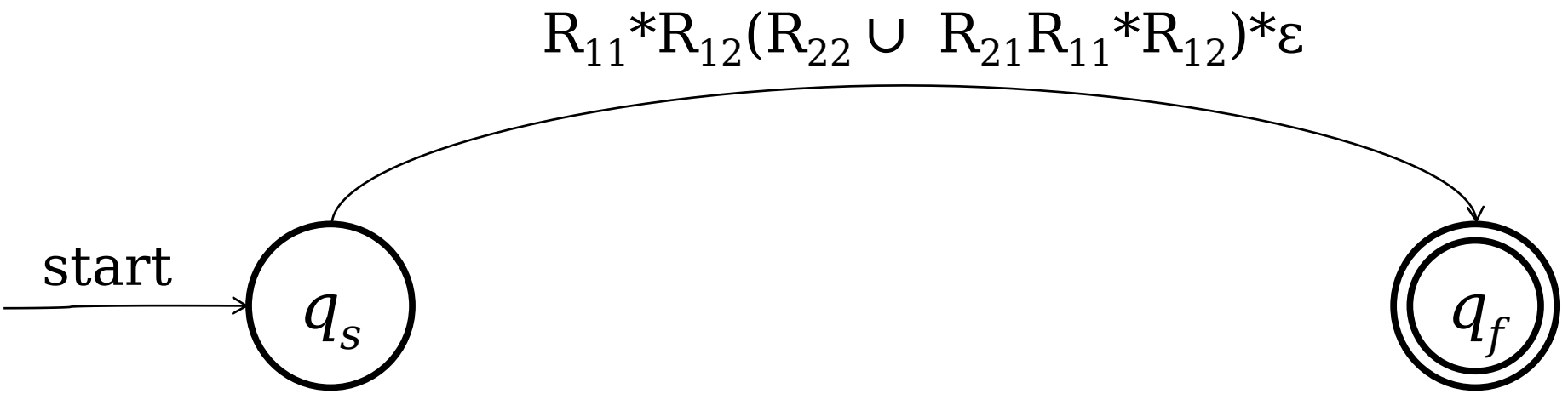
From NFAs to Regular Expressions



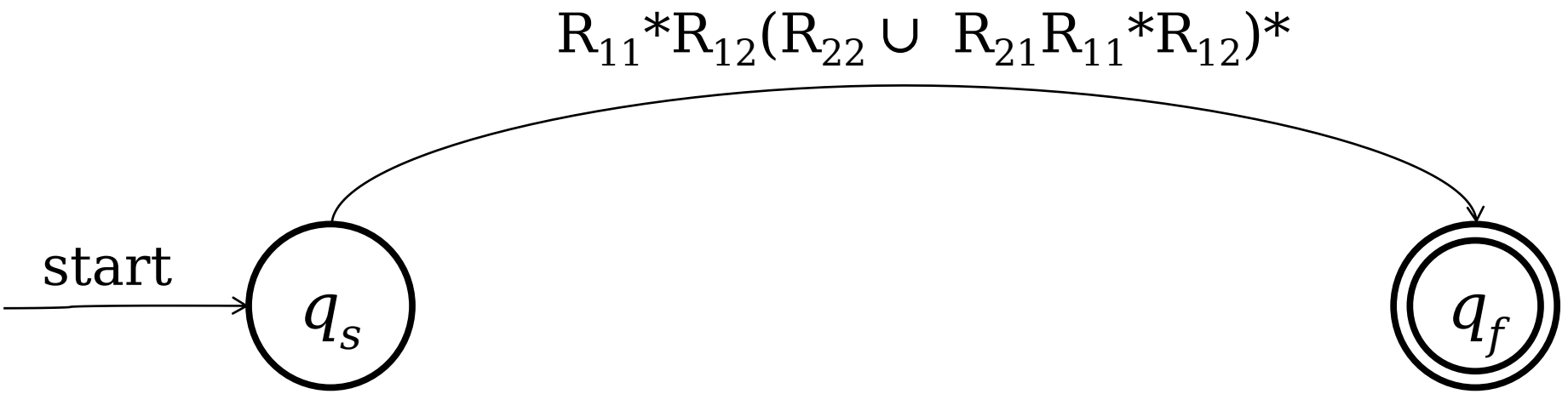
From NFAs to Regular Expressions



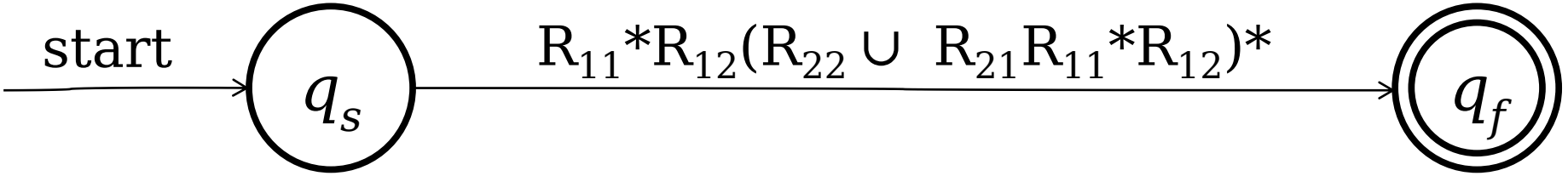
From NFAs to Regular Expressions



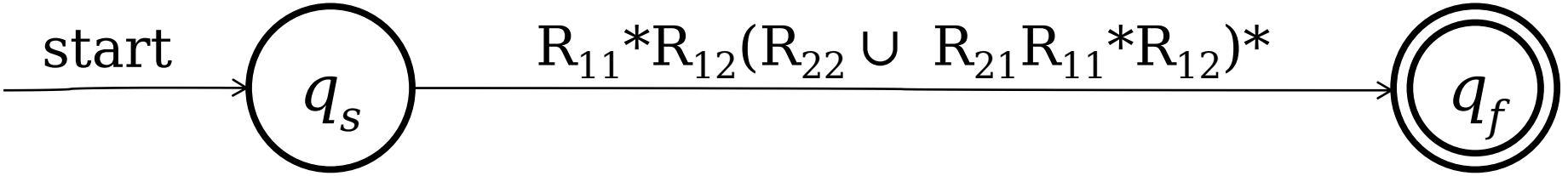
From NFAs to Regular Expressions

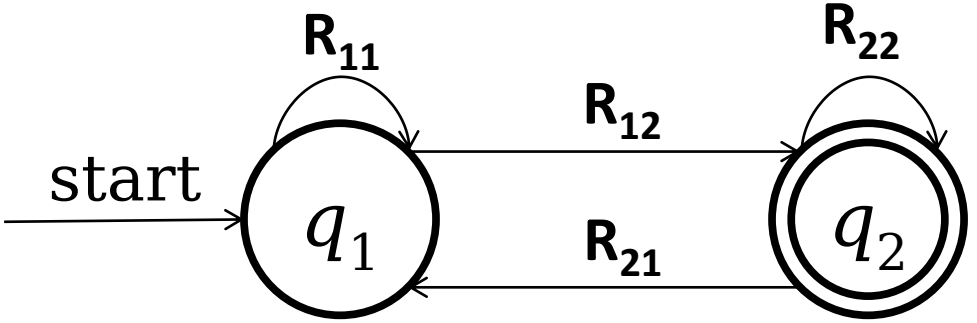


From NFAs to Regular Expressions



From NFAs to Regular Expressions





The Construction at a Glance

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.

Eliminating a State

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.

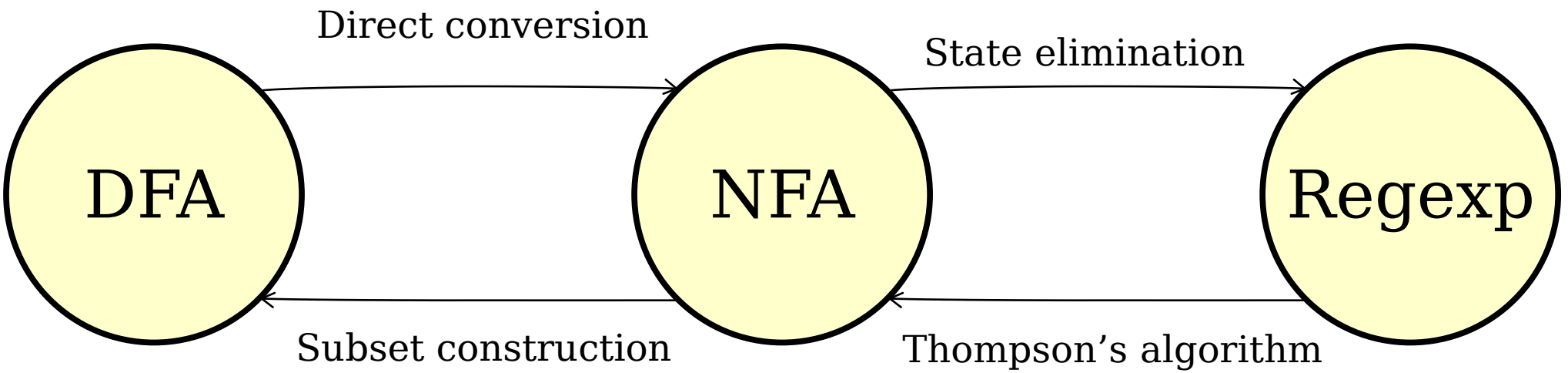
Let R_{out} be the regex on the transition from q to q_1.

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

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.

Tools like `grep` and `flex` that use regular expressions capture all the power available via DFAs and NFAs.

This also is hugely theoretically significant: the regular languages can be assembled “from scratch” using a small number of operations!

Let’s take a five minute break!

Art by 白砂糖
OREO
O&REO
O&O
OREOREO
RERERERERE
OOOOO
OREOO
OREORERERERE ORE
OREOREORE
REREO
REORE
ORERERERERERERERERE O
OOORERERERERERERE OOO
OREREREREOOOOOOOOO
Typeset by Ree
Translated by Mysterious Uploader (danbooru)

Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

For simplicity, let's just use a single character for the "cream" part of the Oreo :)

Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

Design a DFA for the language

$$L = \{ w \in \Sigma^* \mid w \neq \varepsilon \text{ and the first and last character of } w \text{ are the same } \}$$

Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

Design a DFA for the language

$$L = \{\ w \in \Sigma^* \mid w \neq \varepsilon \text{ and the first and last character of } w \text{ are the same } \}$$

ORO $\in L$

ROOOR $\in L$

OROORORRO $\in L$

OR $\notin L$

OOOOOR $\notin L$

RORORORO $\notin L$

Designing DFAs

States – pieces of information

What do I have to keep track of in the course of figuring out whether a string is in this language?

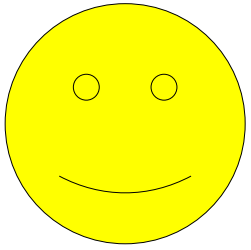
Transitions – updating state

From the state I'm currently in, what do I know about my string? How would reading this character change what I know?

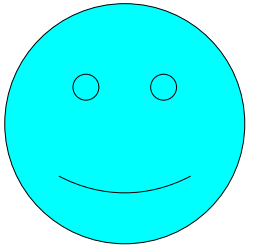
An Analogy

Imagine a scenario where Bob is thinking of a string and Alice has to figure out whether that string is in a particular language

$L = \{ w \text{ is divisible by } 5 \}$



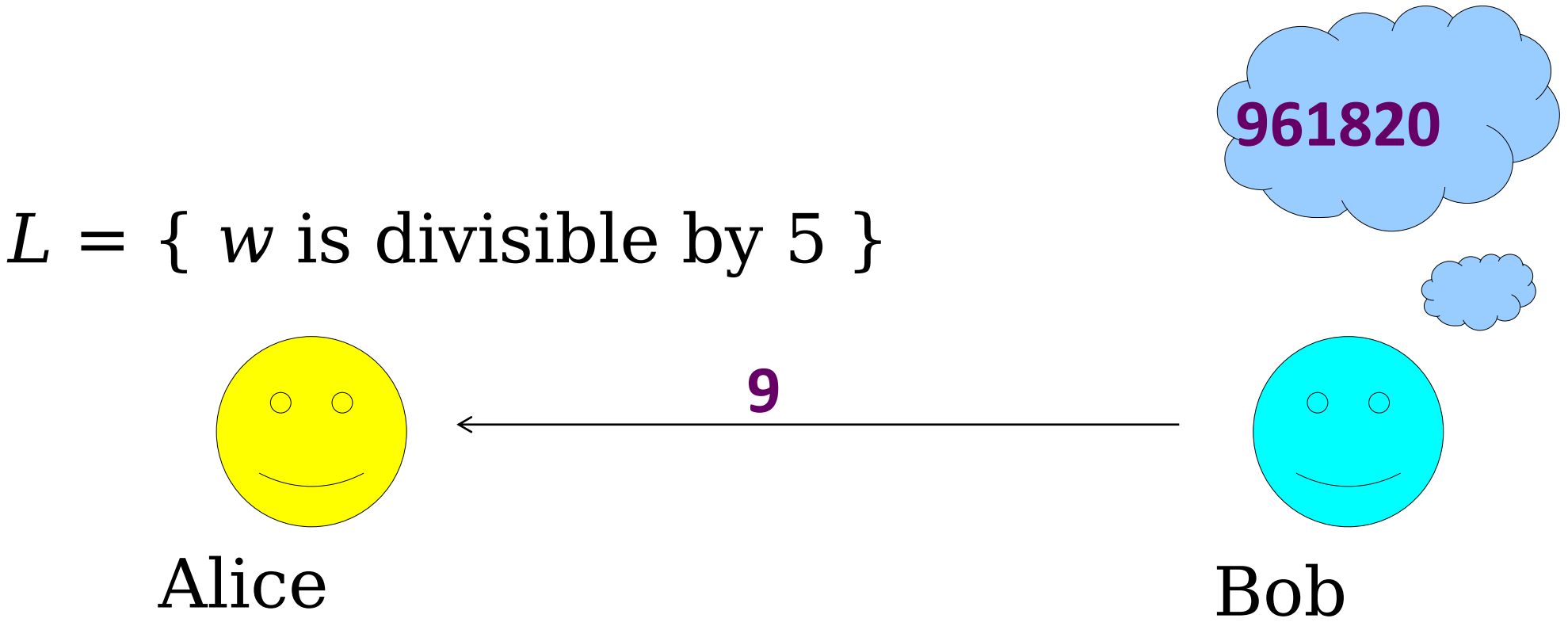
Alice



Bob

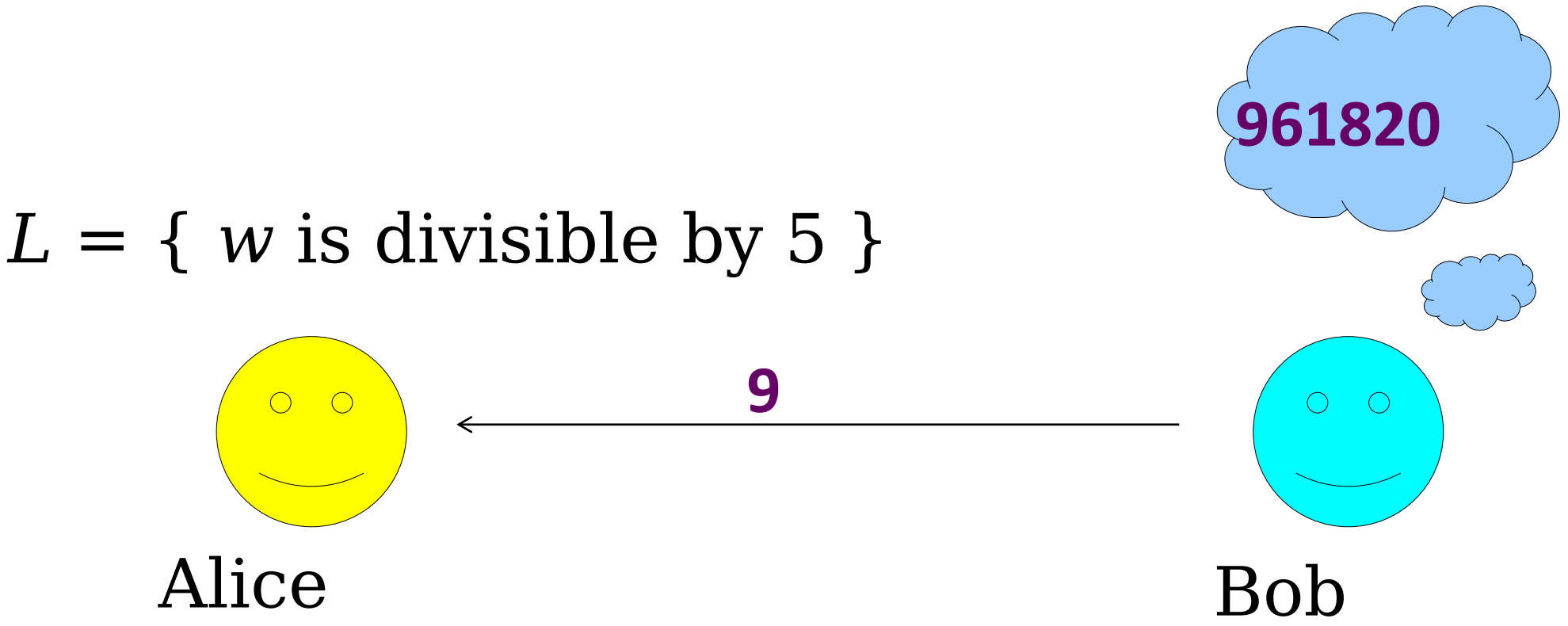
An Analogy

The catch: Bob can only send Alice one character at a time, and Alice doesn't know how long the string is until Bob tells her that he's done sending input



An Analogy

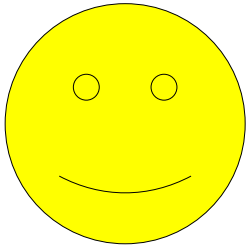
What does Alice need to remember about the characters she's receiving from Bob?



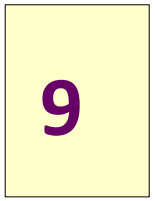
An Analogy

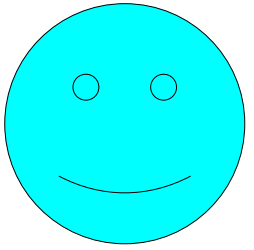
Key insight: Alice only needs to remember the last character she received from Bob

$L = \{ w \text{ is divisible by } 5 \}$



Alice

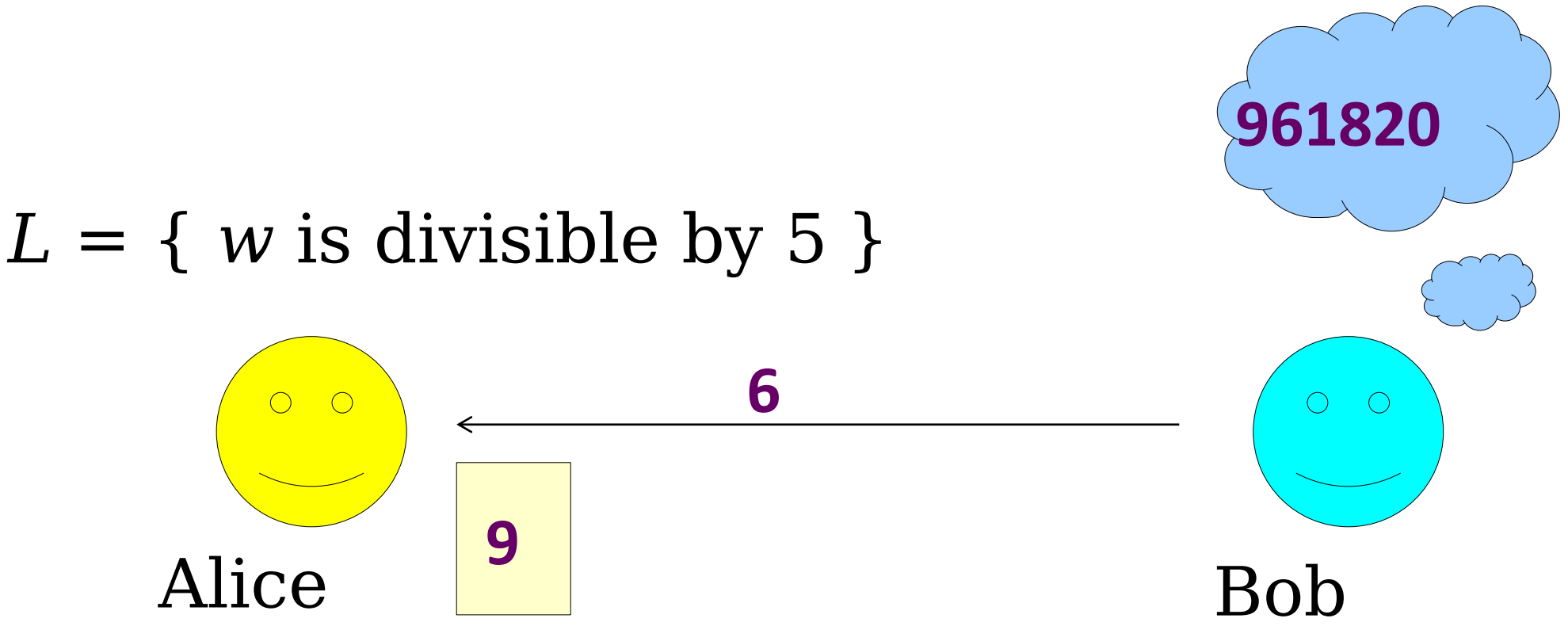




Bob

An Analogy

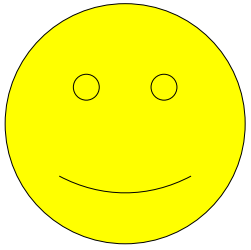
Key insight: Alice only needs to remember the last character she received from Bob



An Analogy

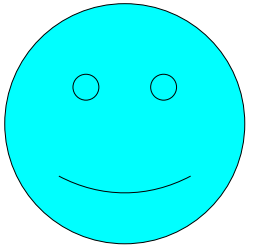
Key insight: Alice only needs to remember the last character she received from Bob

$L = \{ w \text{ is divisible by } 5 \}$



Alice

6

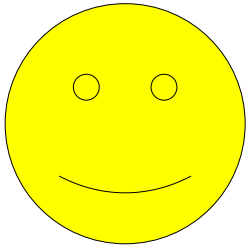


Bob

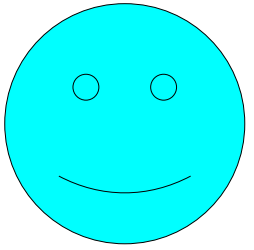
An Analogy

Key insight: Alice only needs to remember the last character she received from Bob

$L = \{\ w$ is divisible by 5 $\}$



. . .



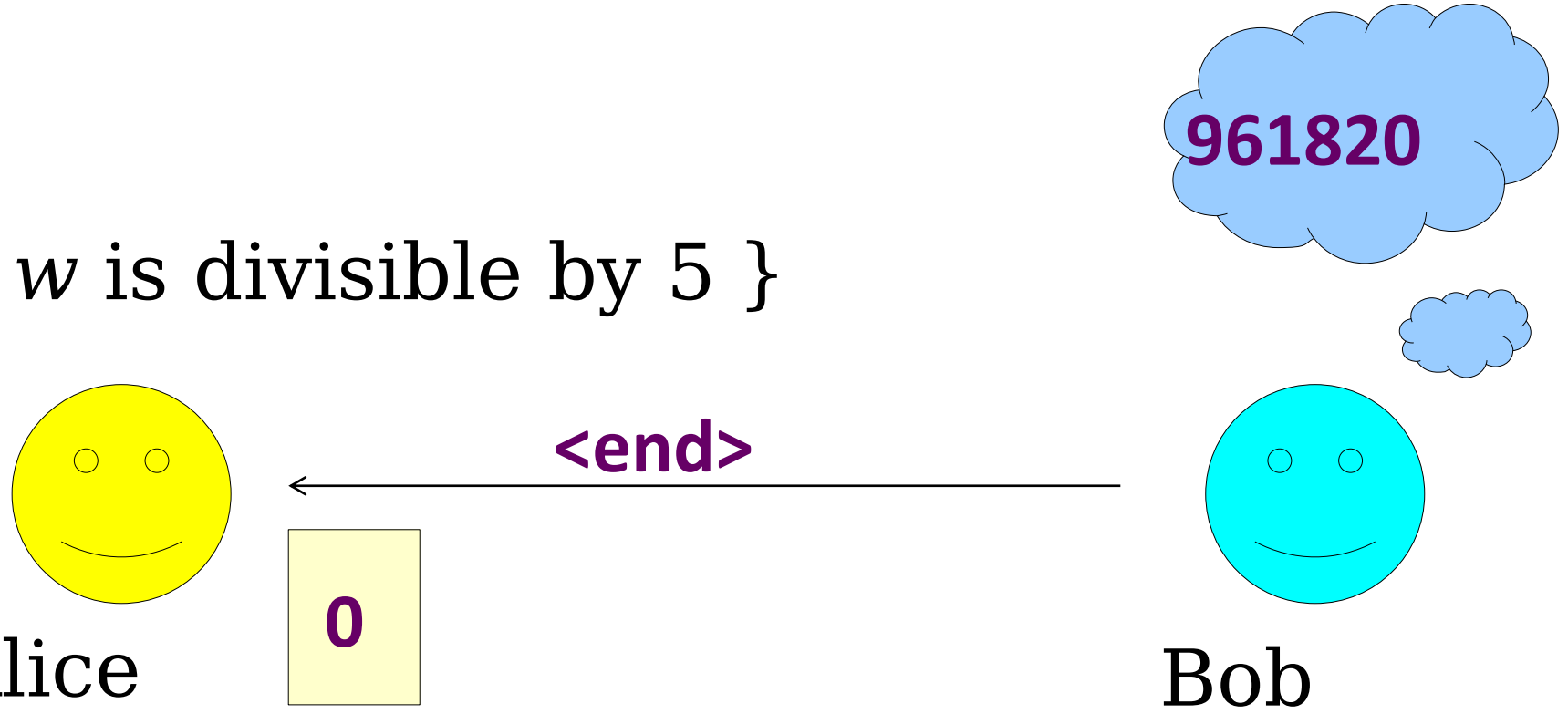
Alice

Bob

An Analogy

Eventually Bob gets to the end of his string and sends Alice a signal that he's done sending input

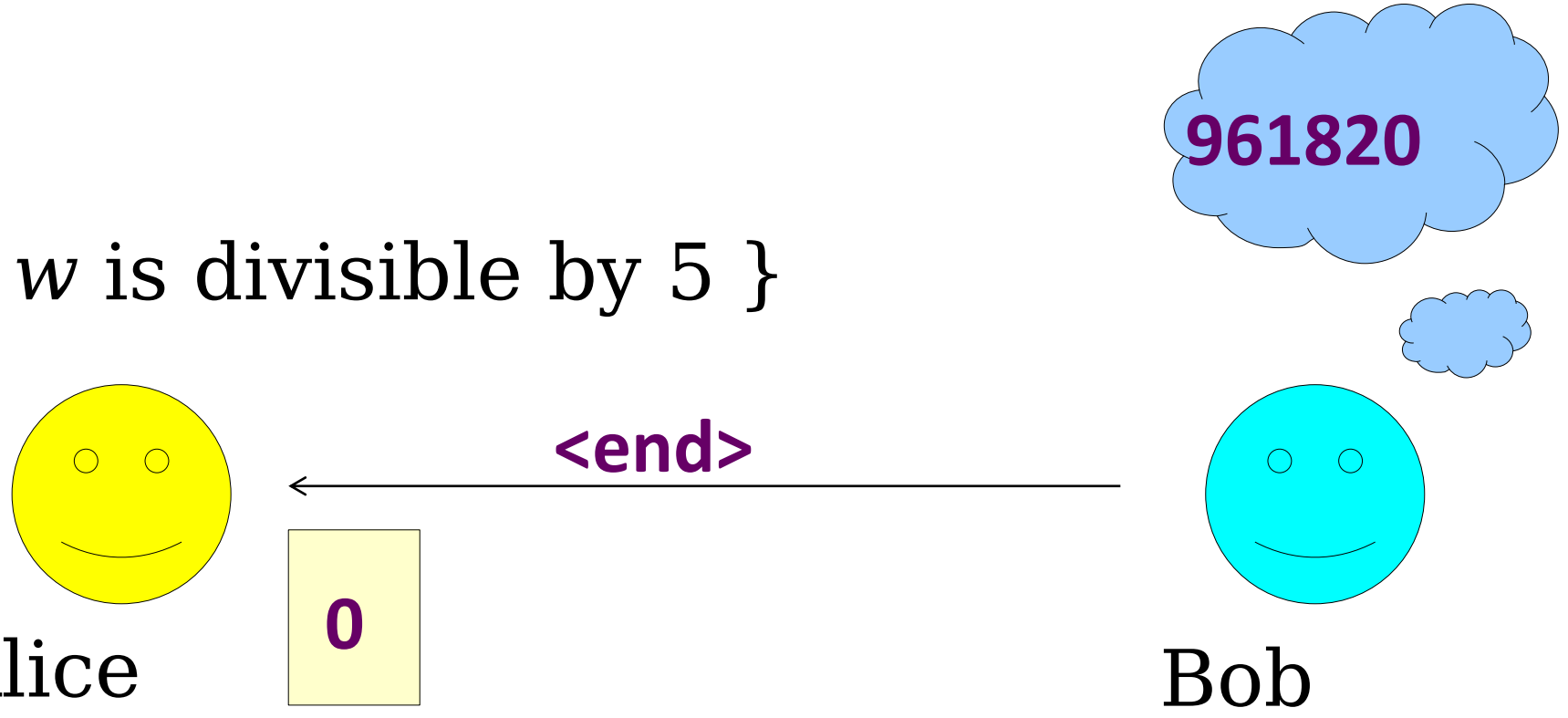
$L = \{ w \text{ is divisible by } 5 \}$



An Analogy

At this point, Alice just has to look at the last digit she wrote down and if it's a 5 or 0, Bob's string belongs in the language

$L = \{ w \text{ is divisible by } 5 \}$



DFA Design Strategy

1. Answer the question “What do I have to keep track of in the course of figuring out whether a string is in this language?”

2. Create a state that represents each possible answer to that question.

3. From each state, go through all of the characters and answer the question “How would reading this character change what I know about my string?” and draw transitions to the appropriate states.

DFA Design Strategy

$$L = \{ w \text{ is divisible by } 5 \}$$

1. Answer the question “What do I have to keep track of in the course of figuring out whether a string is in this language?”

We need to keep track of the last character.

2. Create a state that represents each possible answer to that question.

The last character could be any digit 0-9. The states for 0 and 5 are accepting states.

3. From each state, go through all of the characters and answer the question “How would reading this character change what I know about my string?” and draw transitions to the appropriate states.

Reading a character *d* should transition to the state representing “the last character of the string is *d*”.

Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

Design a DFA for the language

$$L = \{ w \in \Sigma^* \mid w \neq \varepsilon \text{ and the first and last character of } w \text{ are the same } \}$$

What do I have to keep track of in the course of figuring out whether a string is in this language?

Oreo Sandwiches

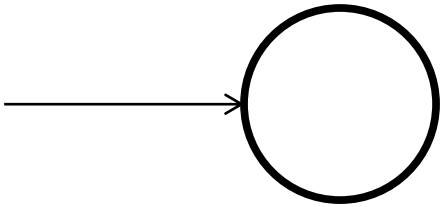
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$

We need to keep track of the very first character

And we need to keep track of the last character we've read so that when we reach the end, we can check whether the first and last characters were the same

Oreo Sandwiches

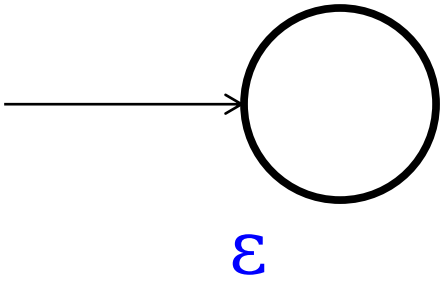
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Remember that each state should represent a piece of information. We'll annotate what each state represents in blue.

Oreo Sandwiches

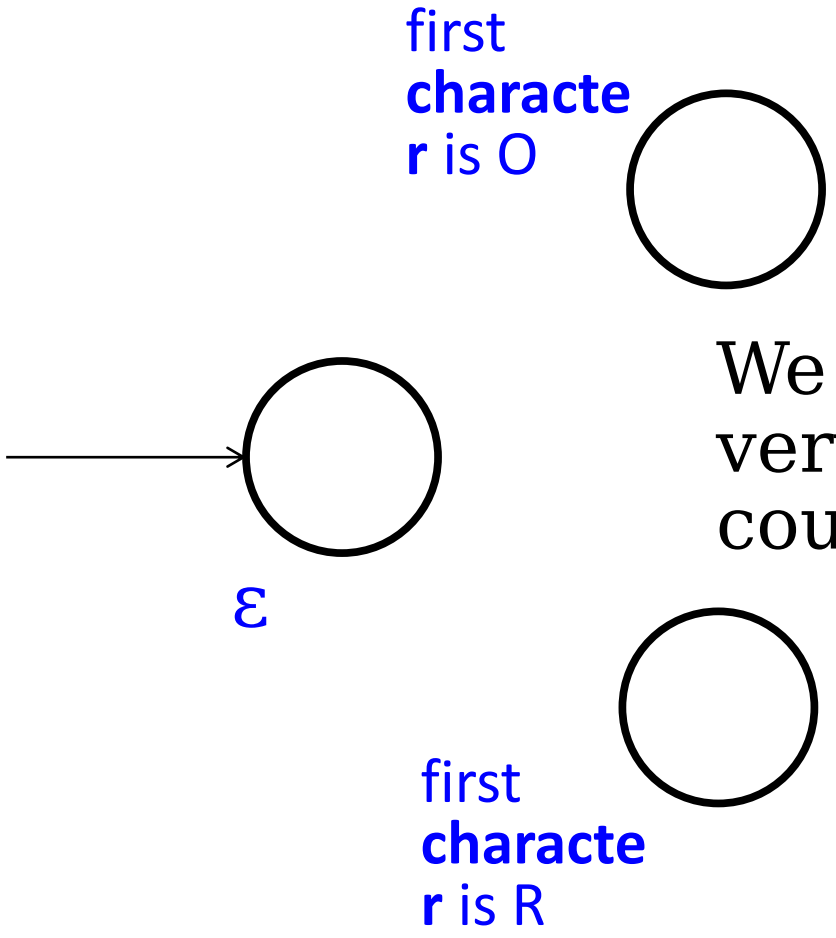
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



We need to keep track of the very first character, which could either be an **O** or an **R**

Oreo Sandwiches

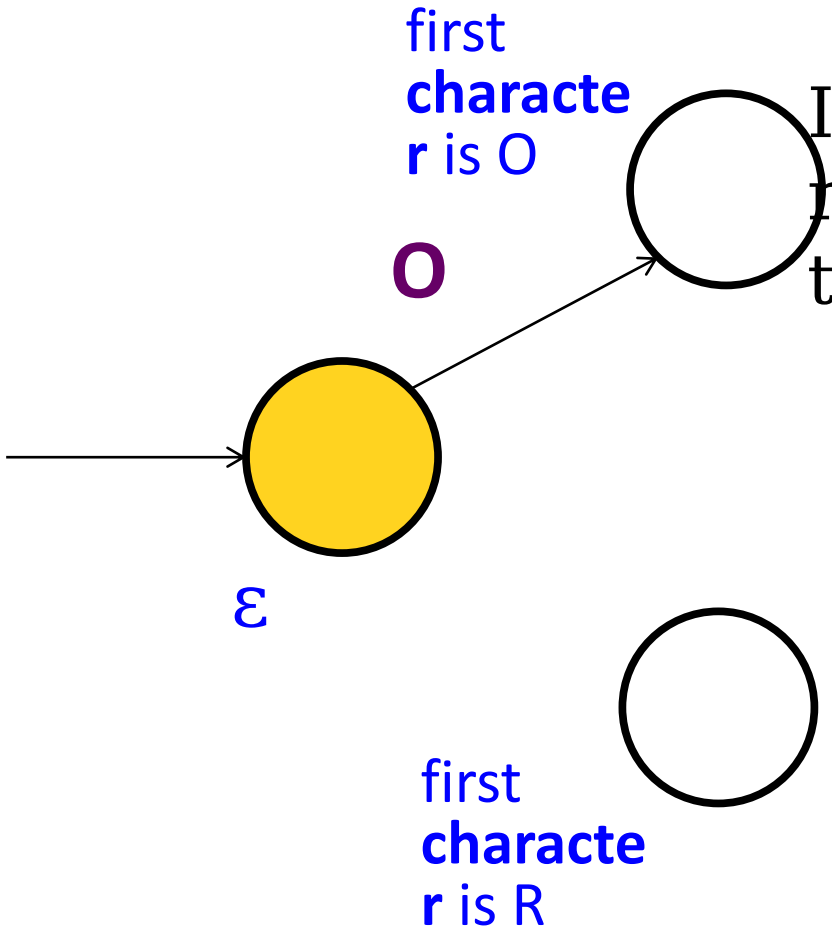
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



We need to keep track of the very first character, which could either be an **O** or an **R**

Oreo Sandwiches

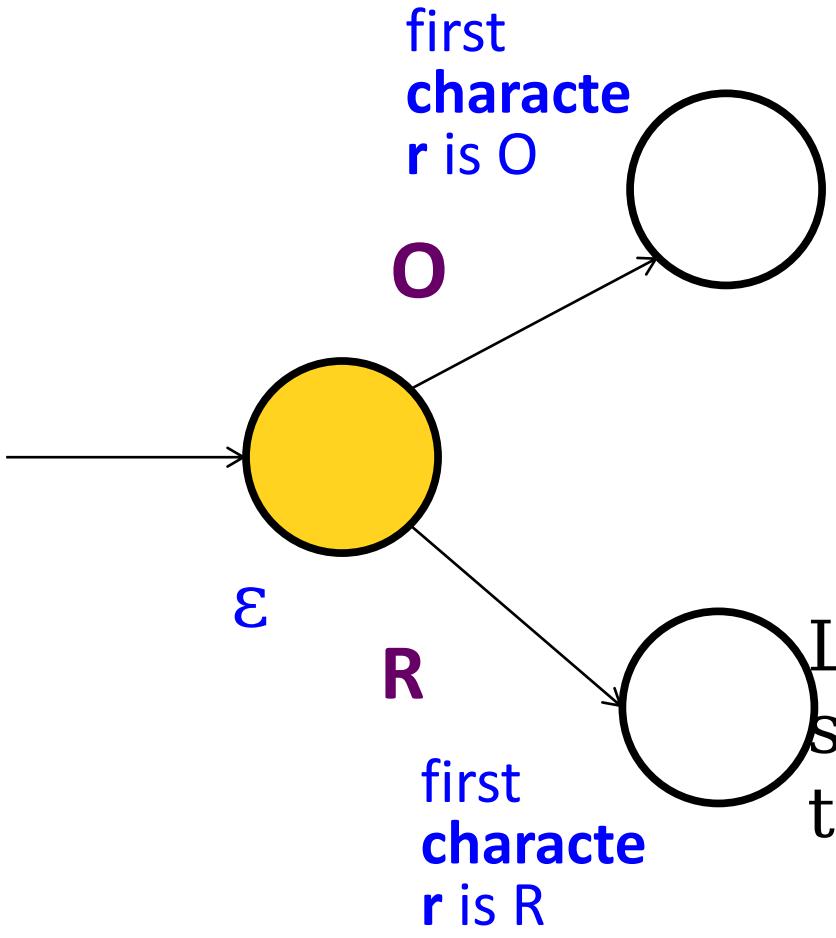
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



If I'm in the start state and I read an **O**, I should transition to this state

Oreo Sandwiches

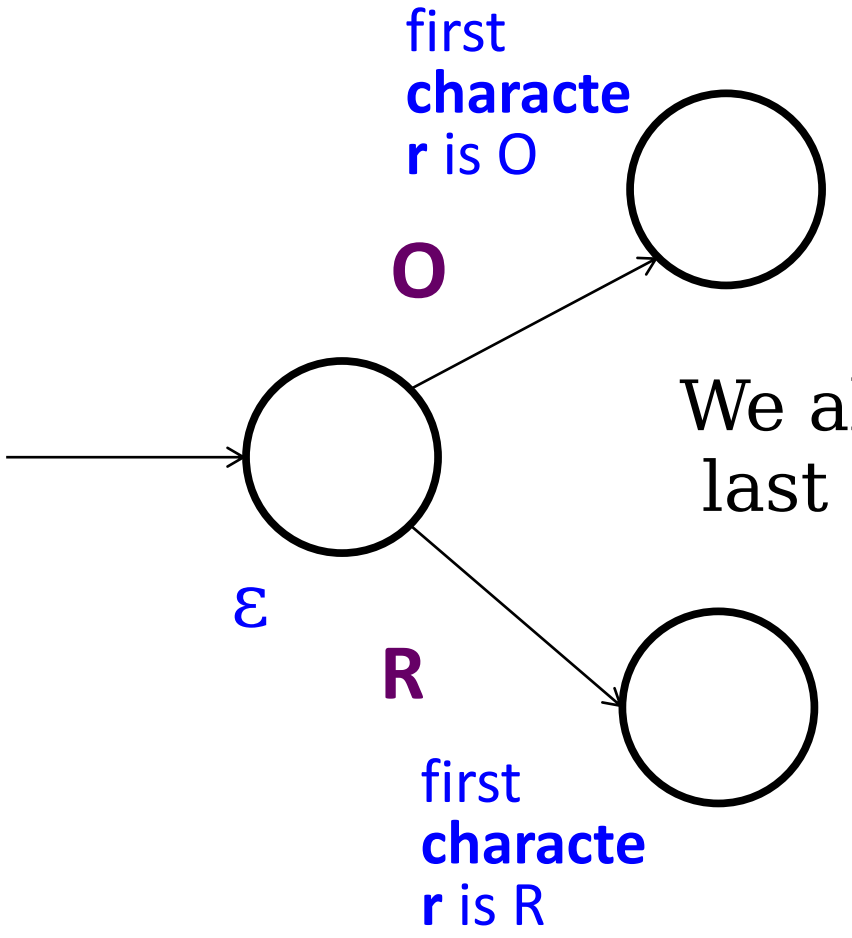
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Likewise if I'm in the start state and I read an **R**, I should transition to this state

Oreo Sandwiches

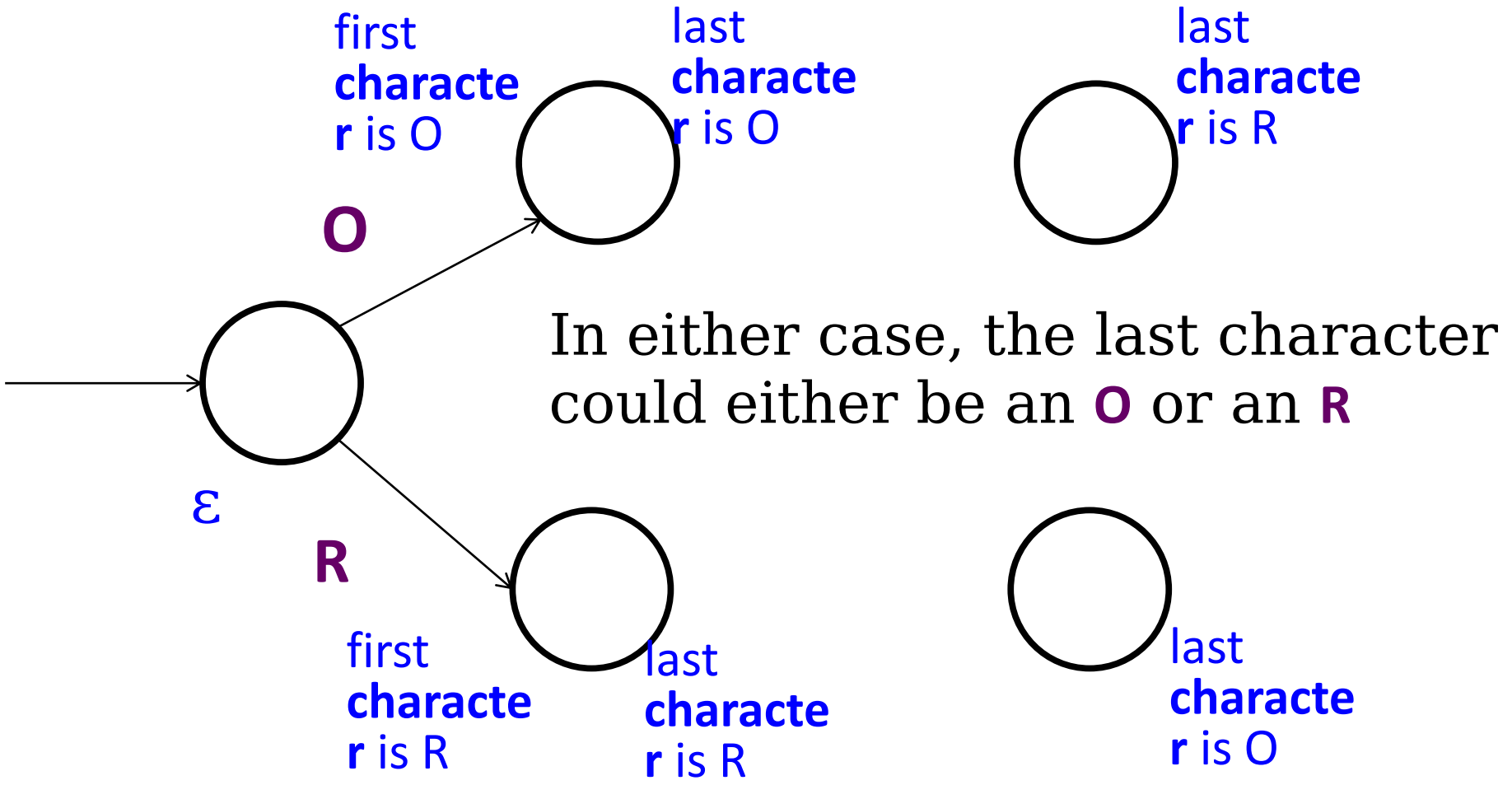
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



We also need to keep track of the last character we've read

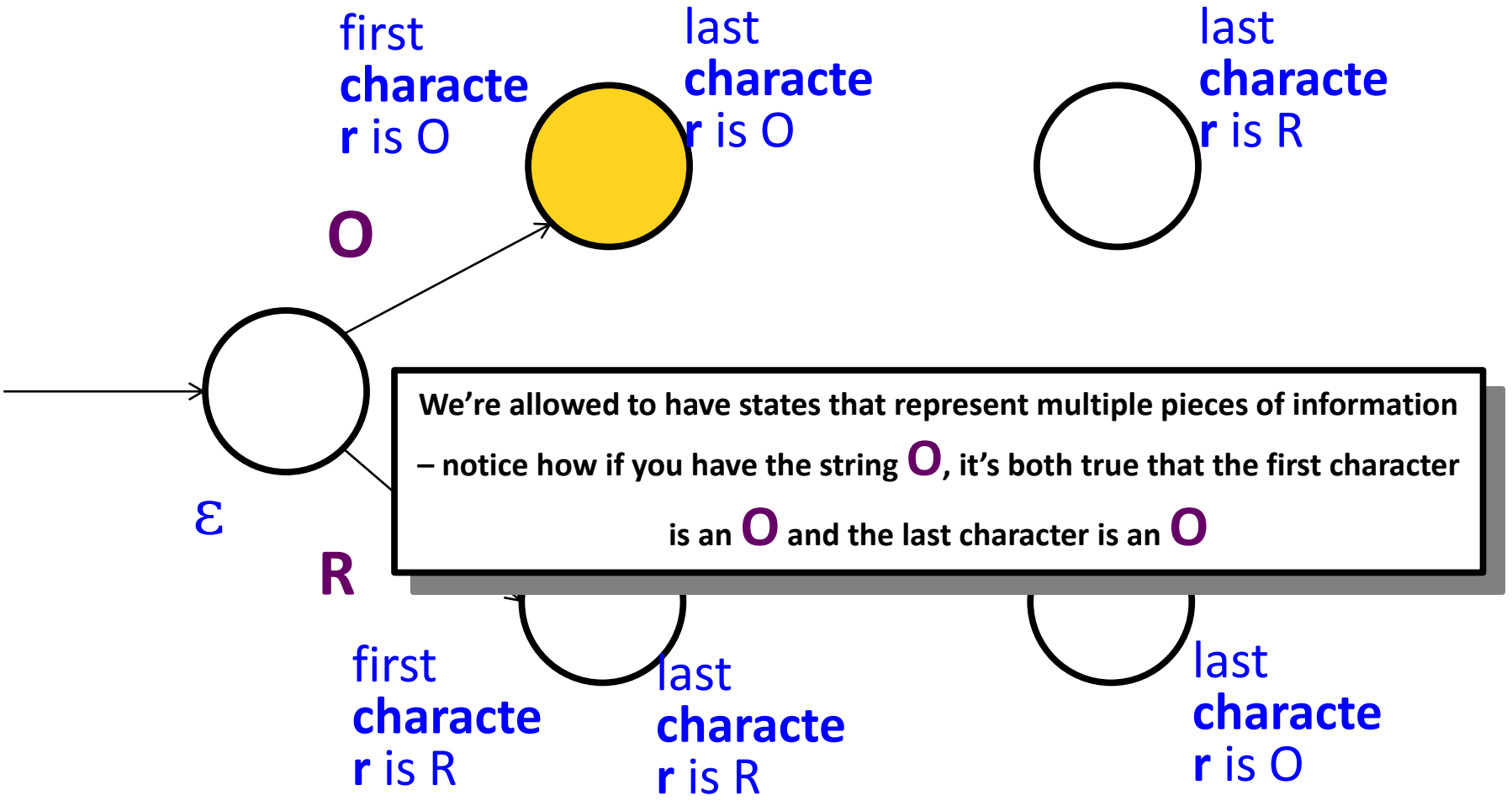
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



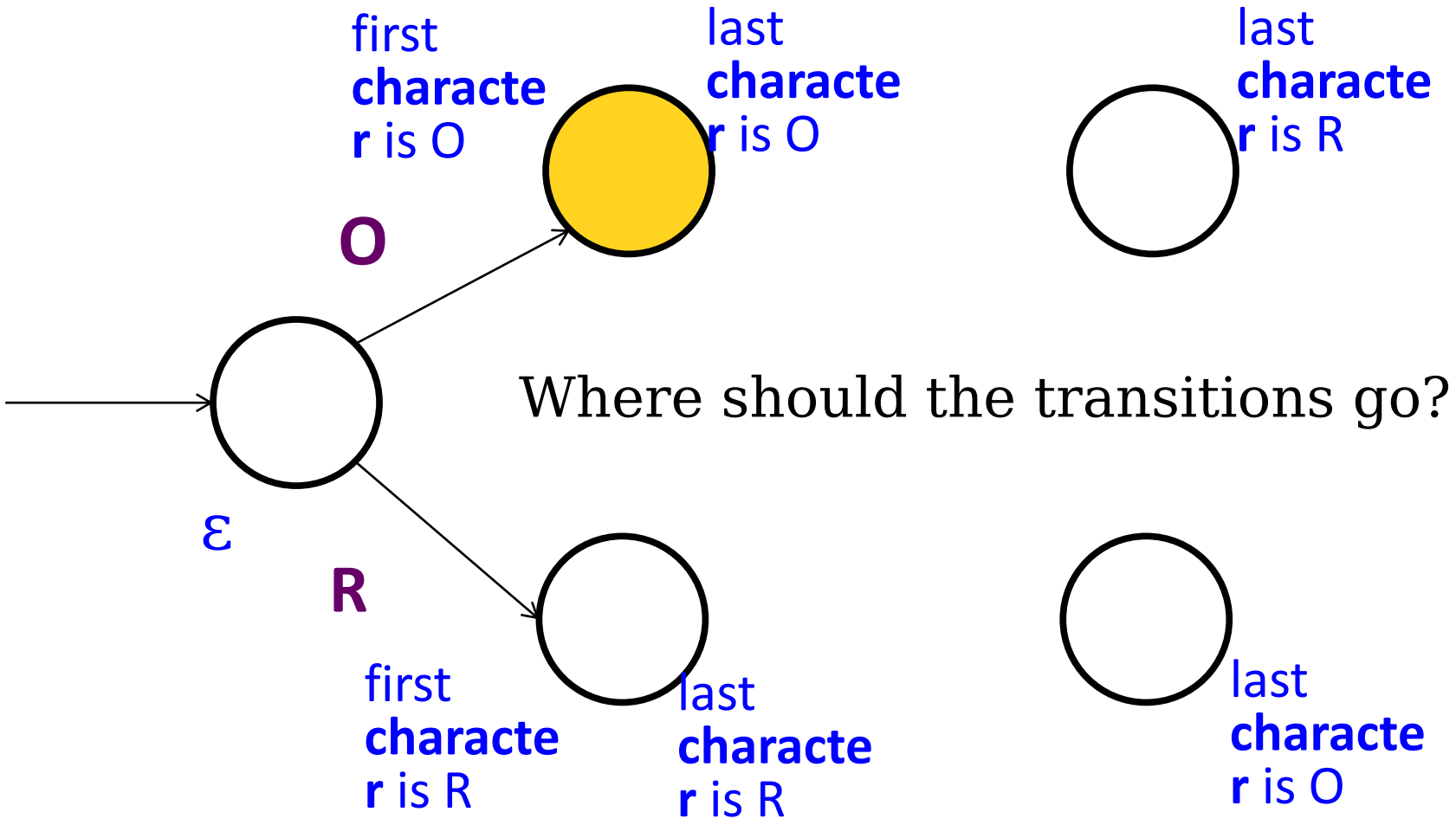
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



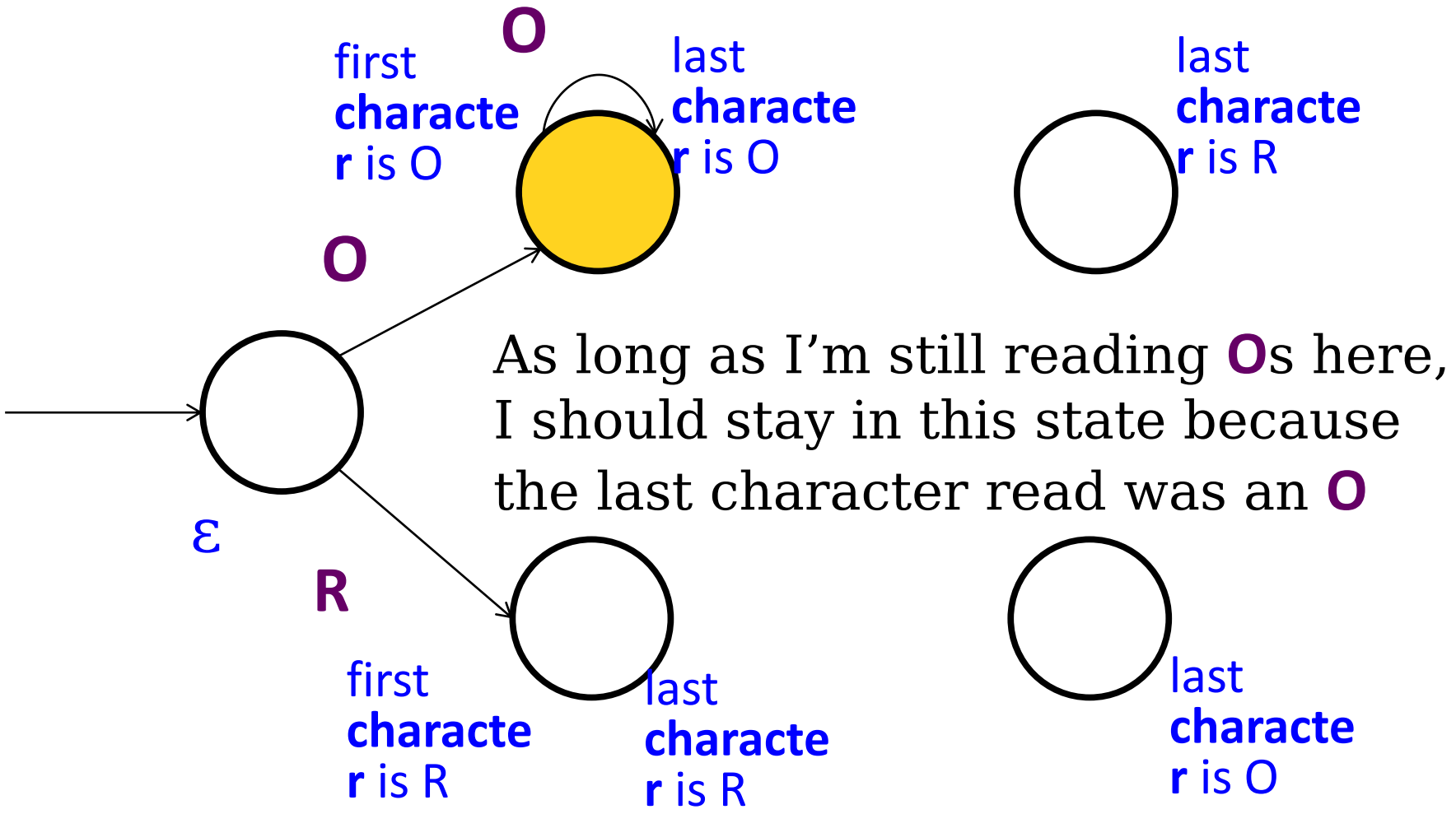
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



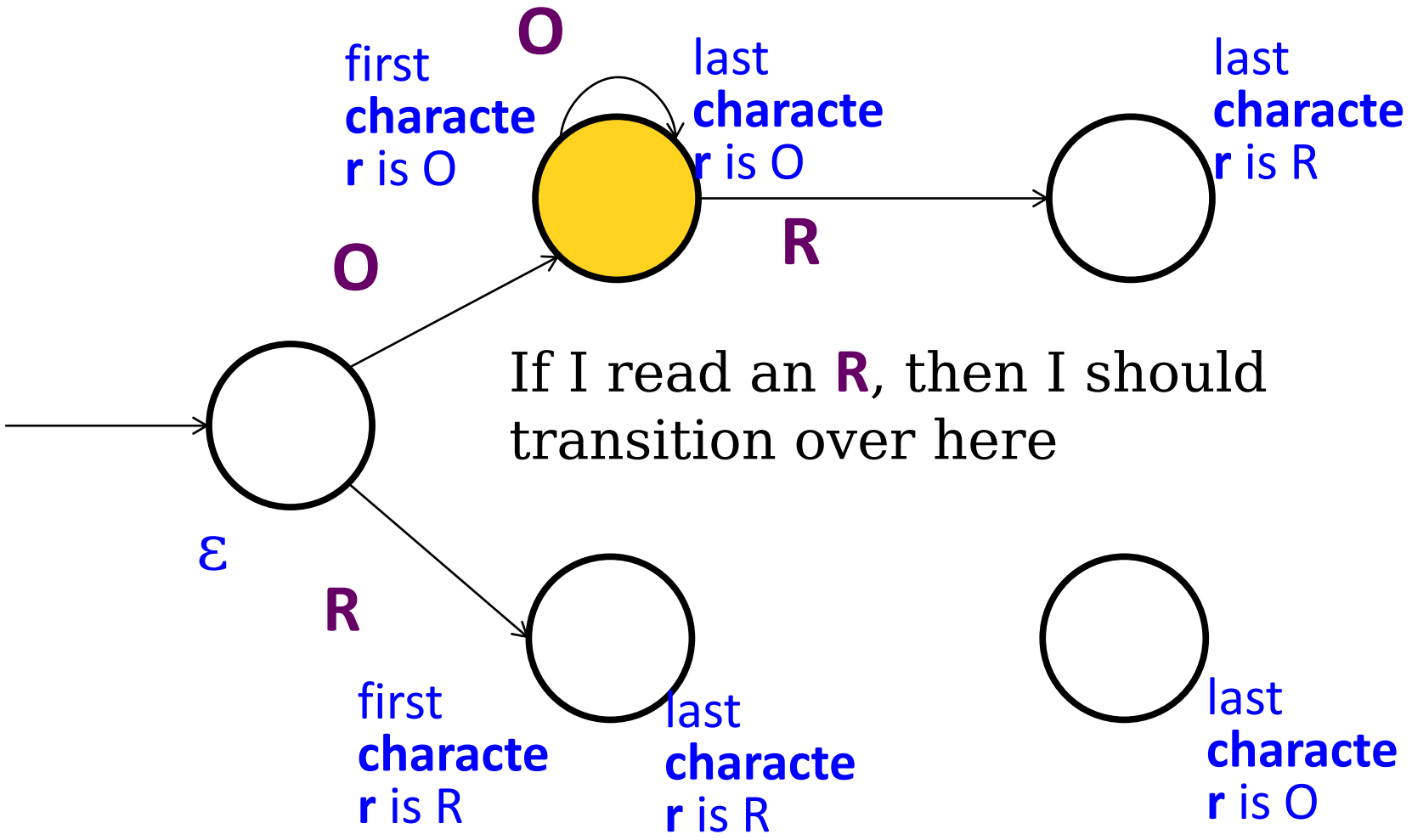
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



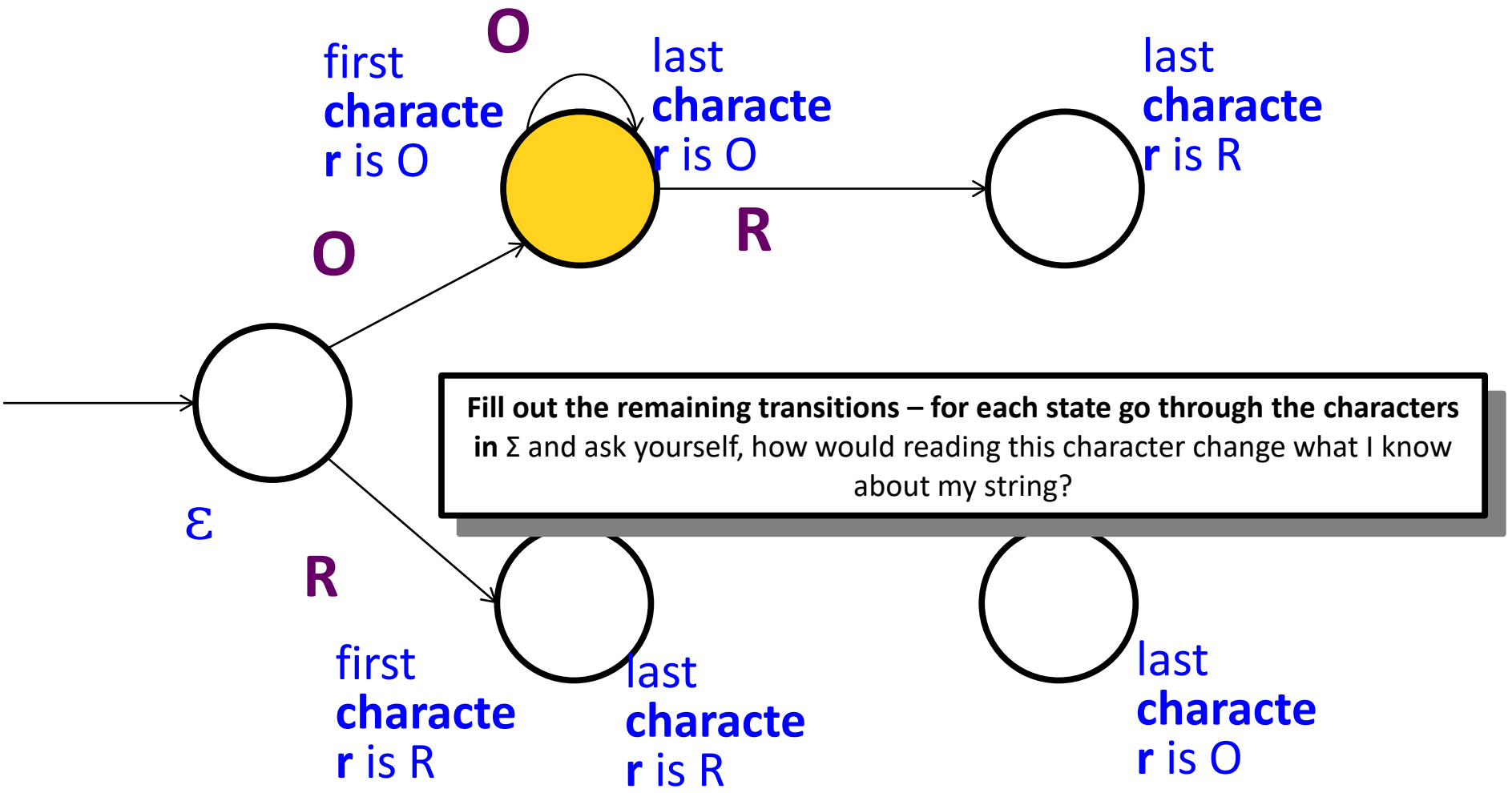
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



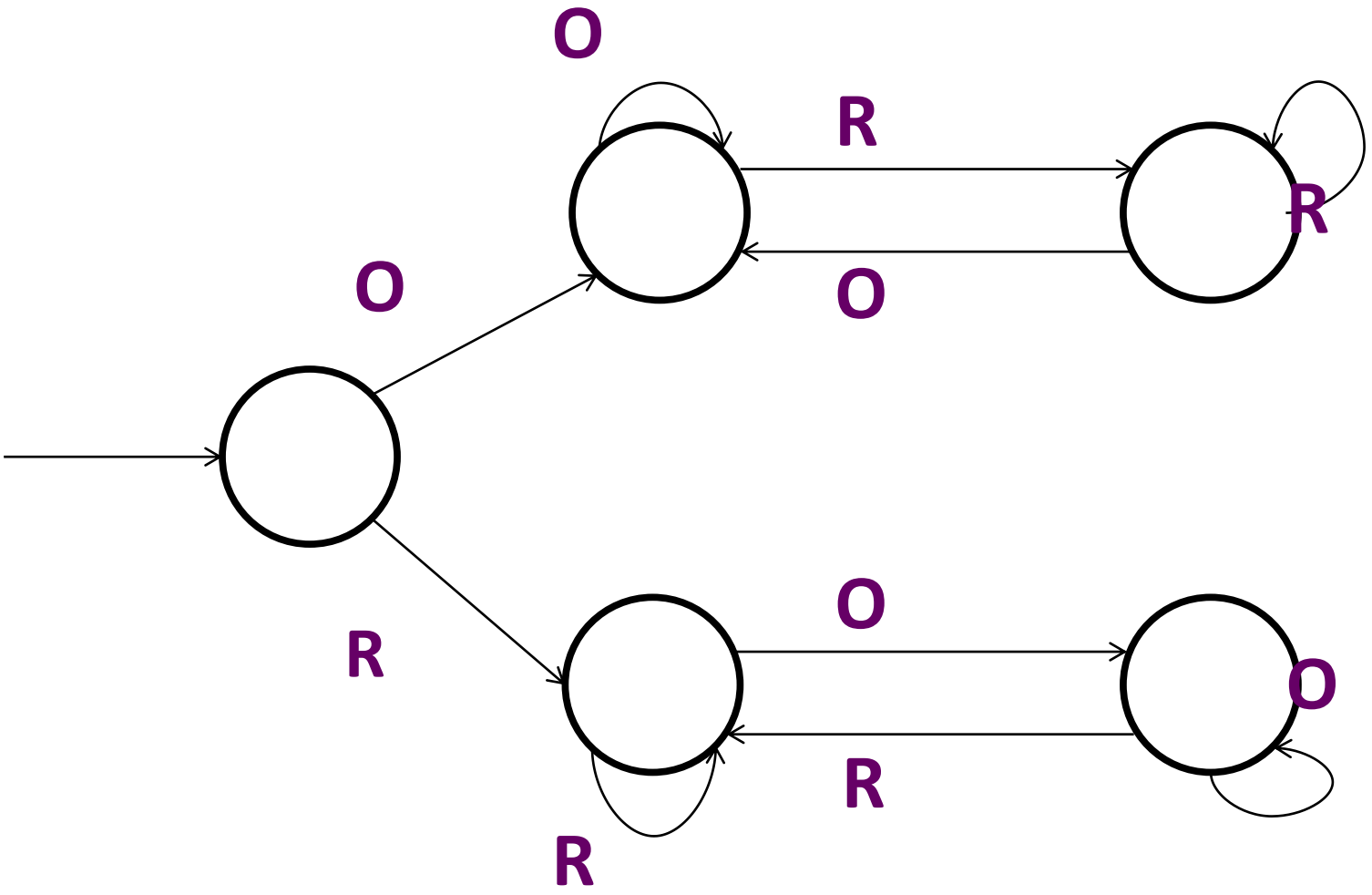
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



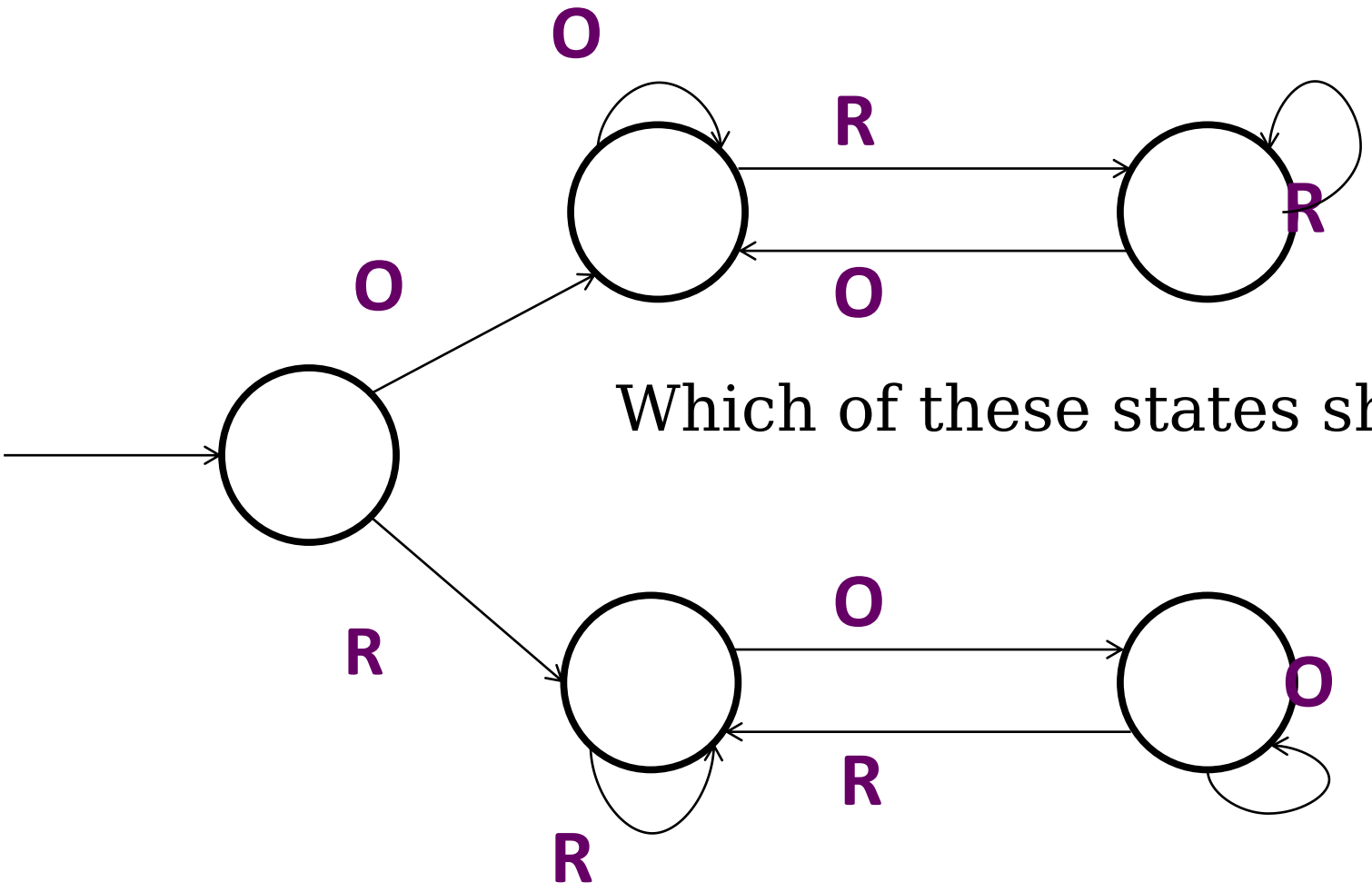
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Oreo Sandwiches

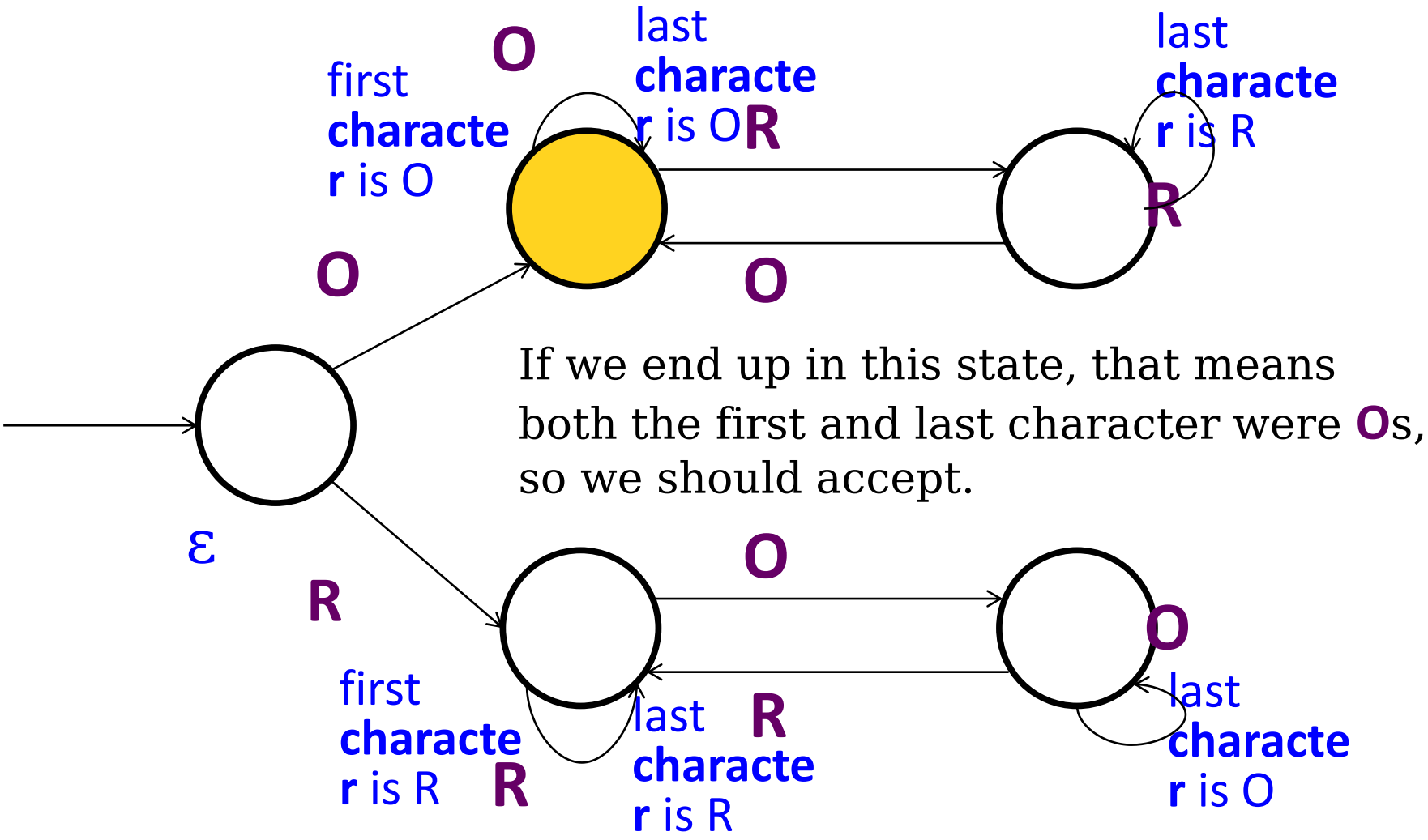
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Which of these states should be acc

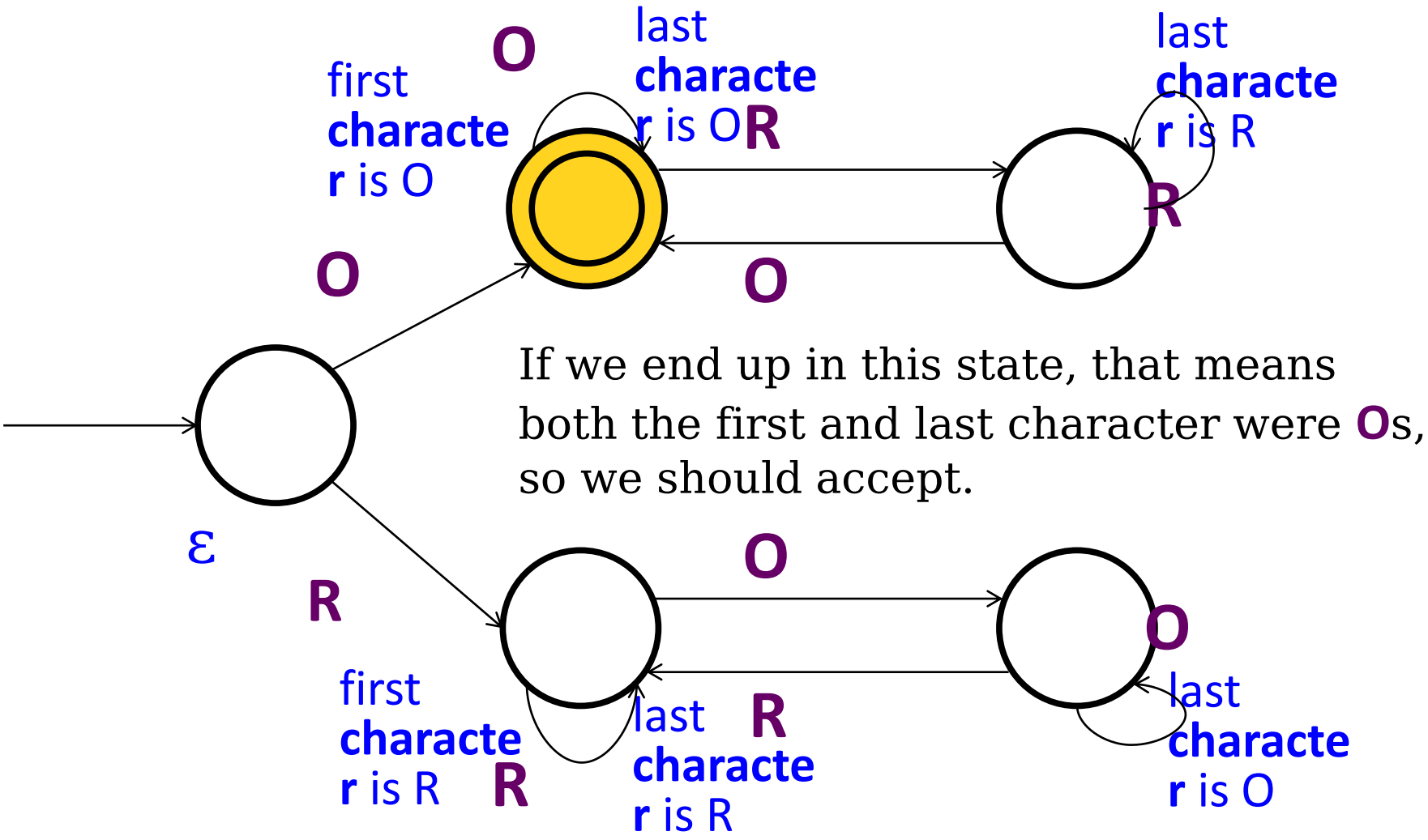
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



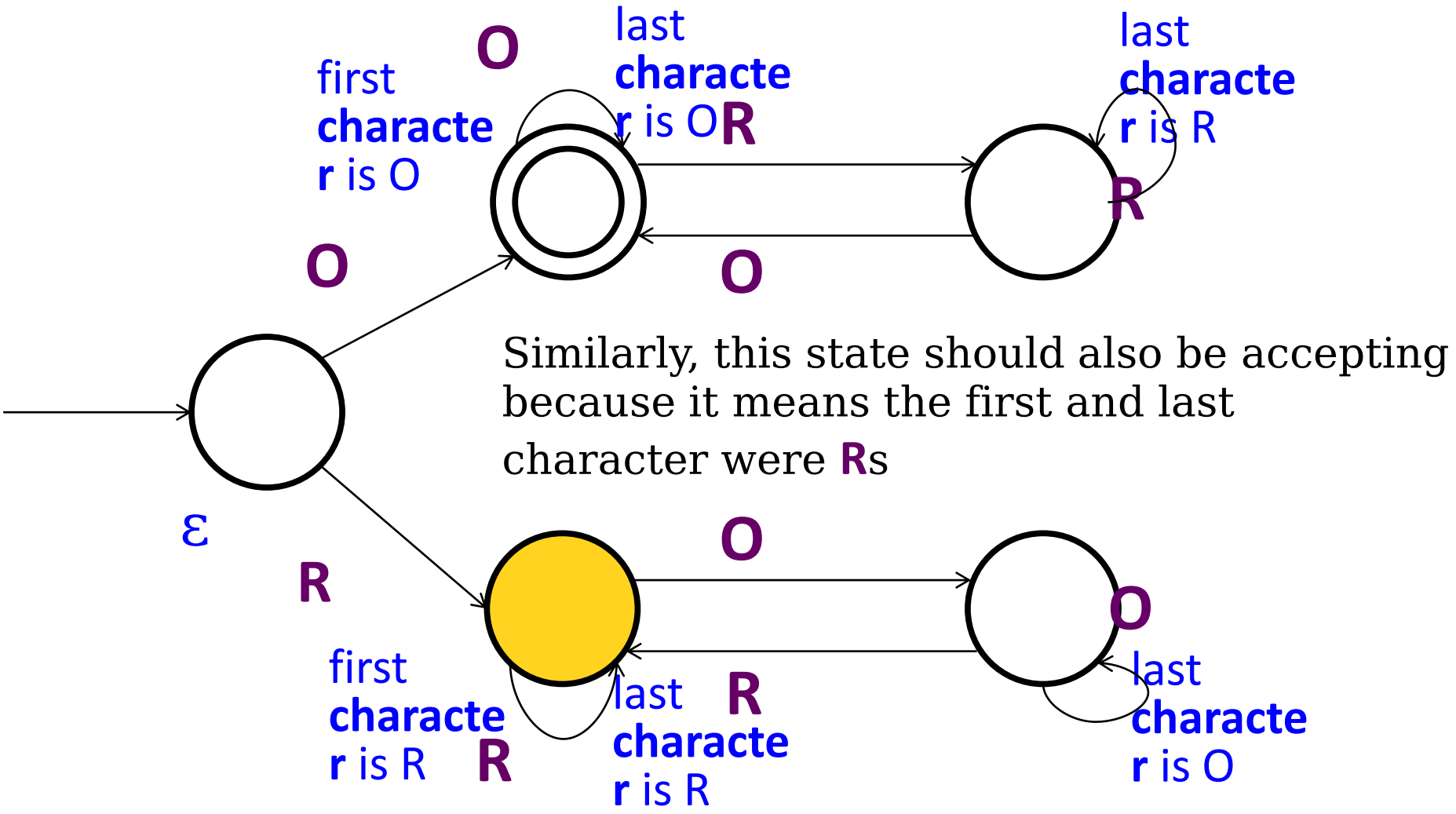
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



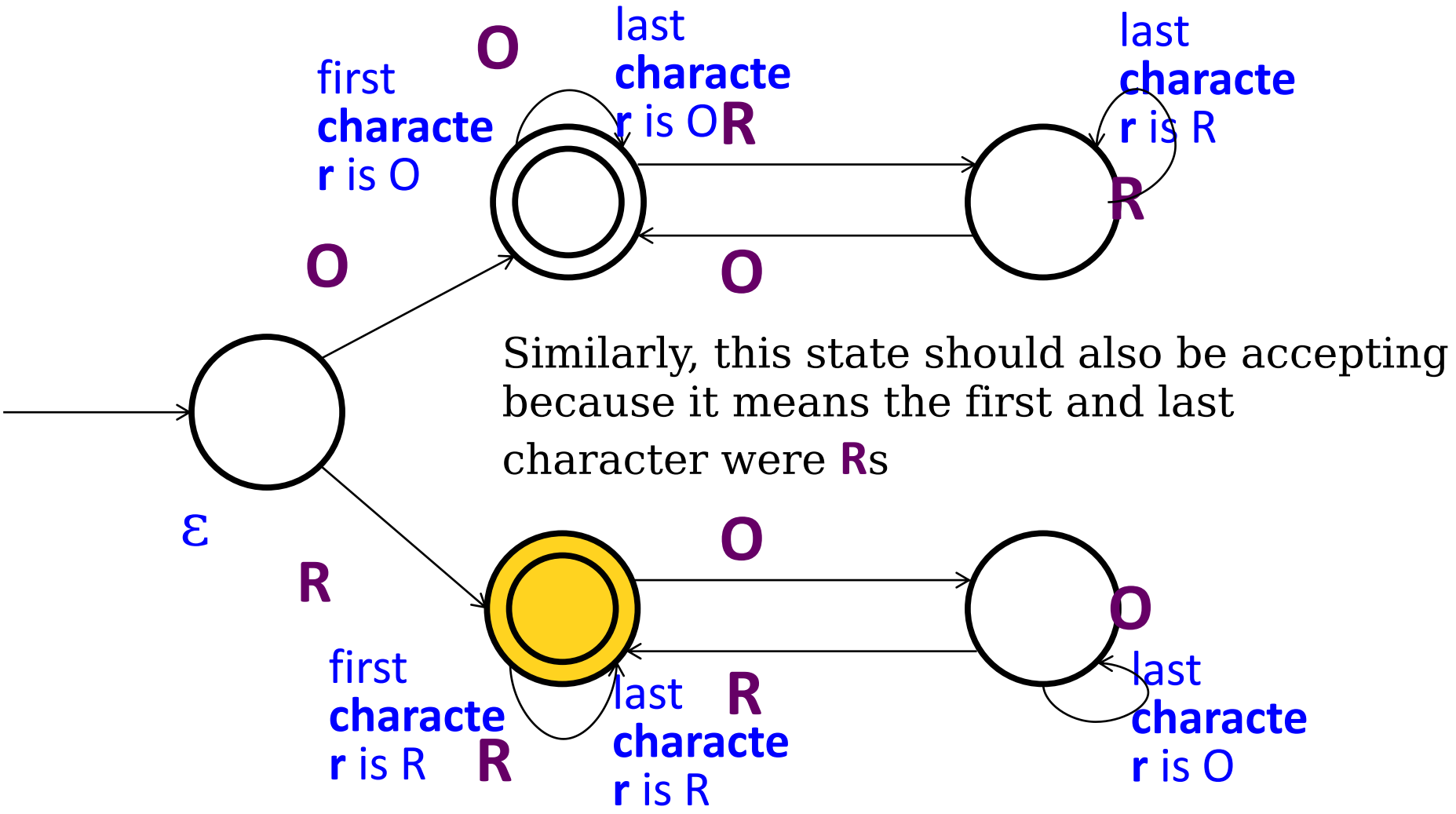
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



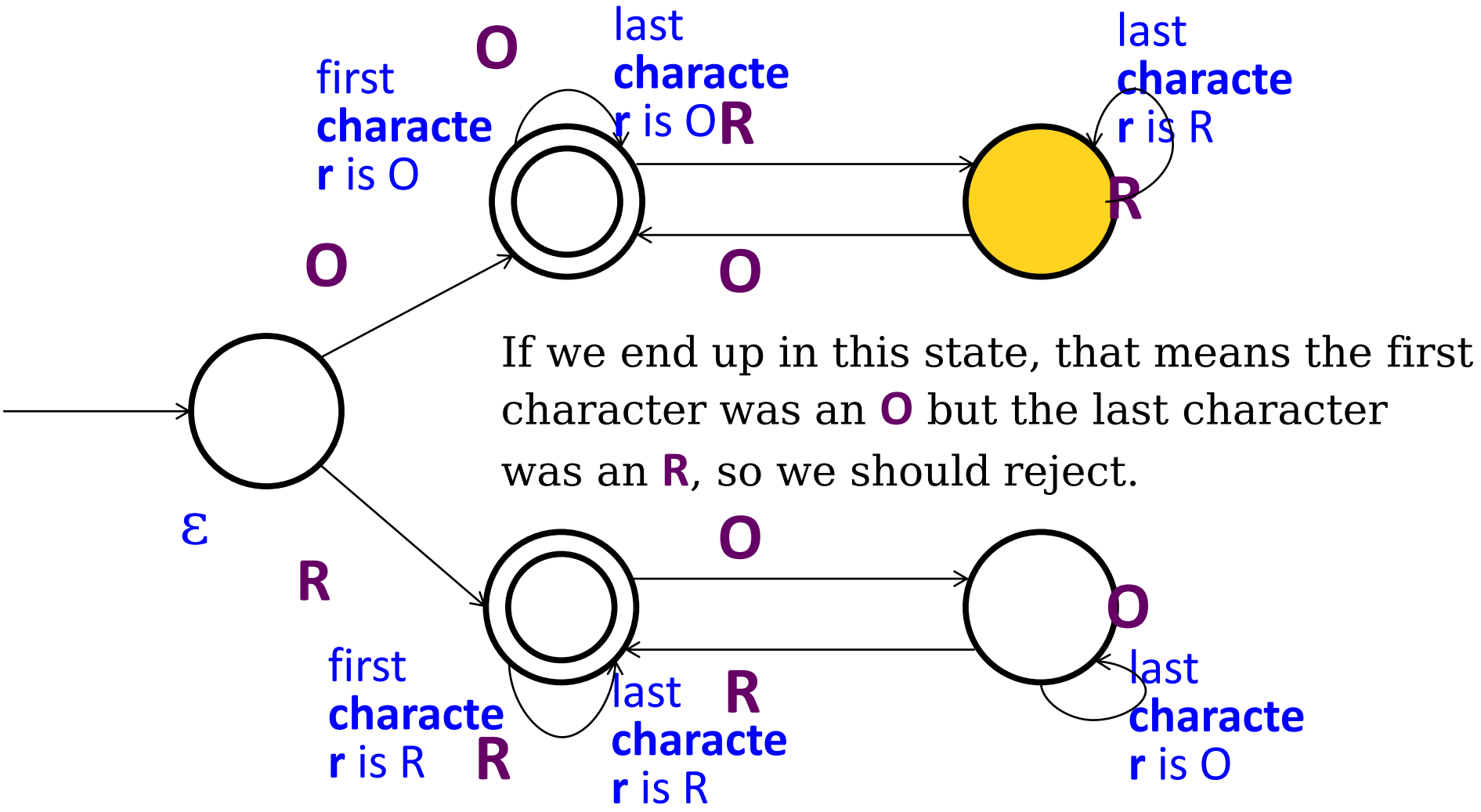
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



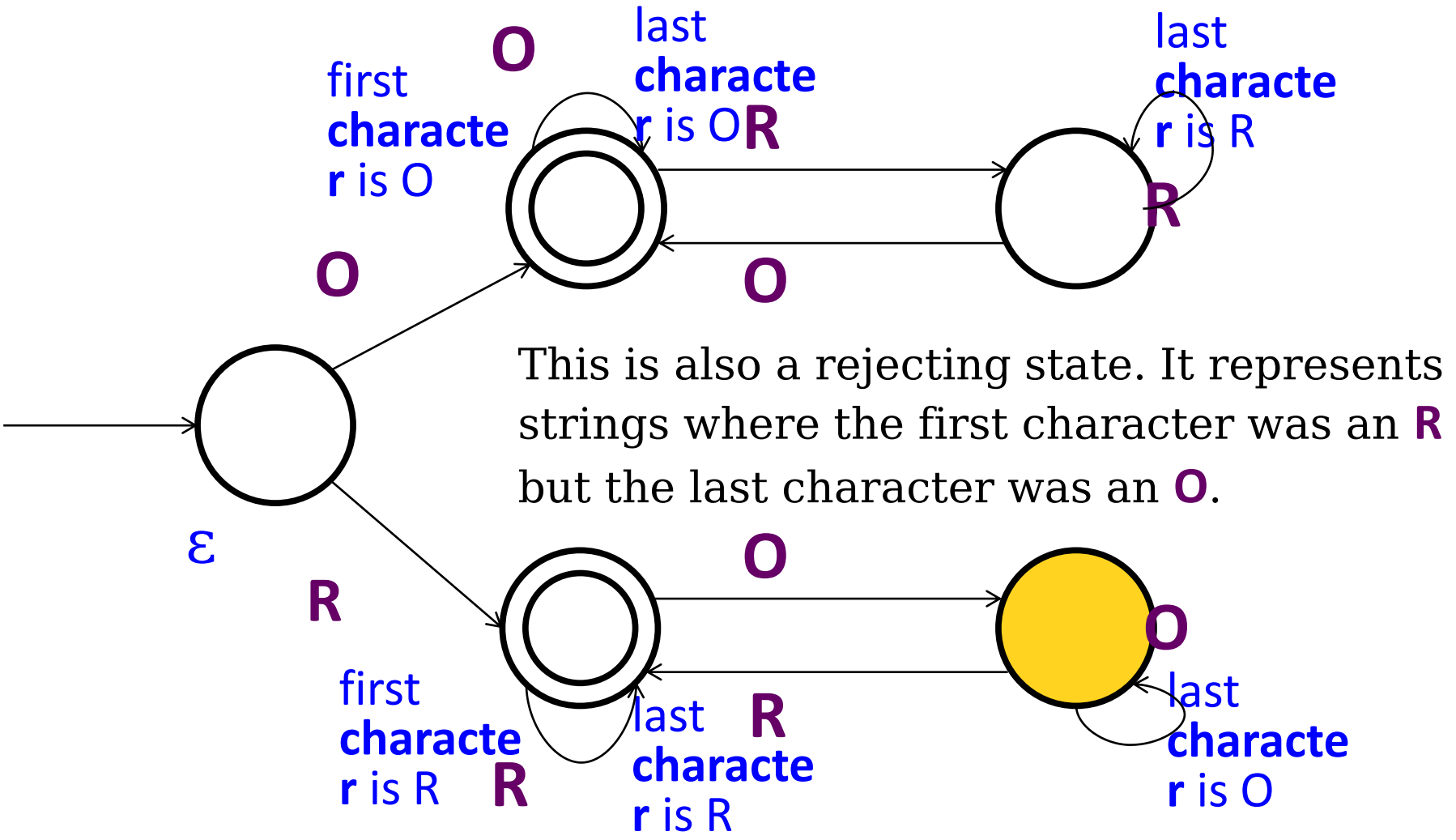
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



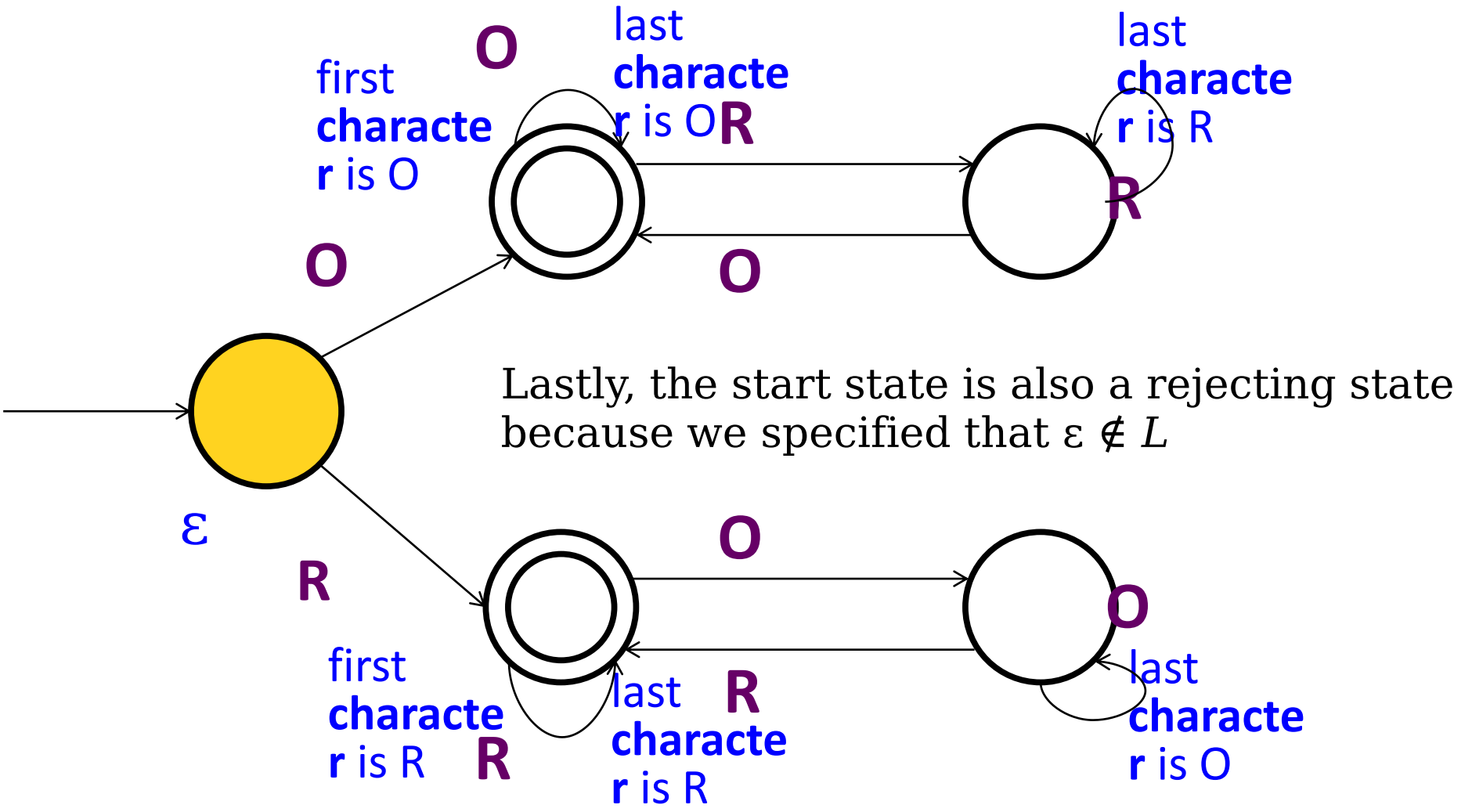
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



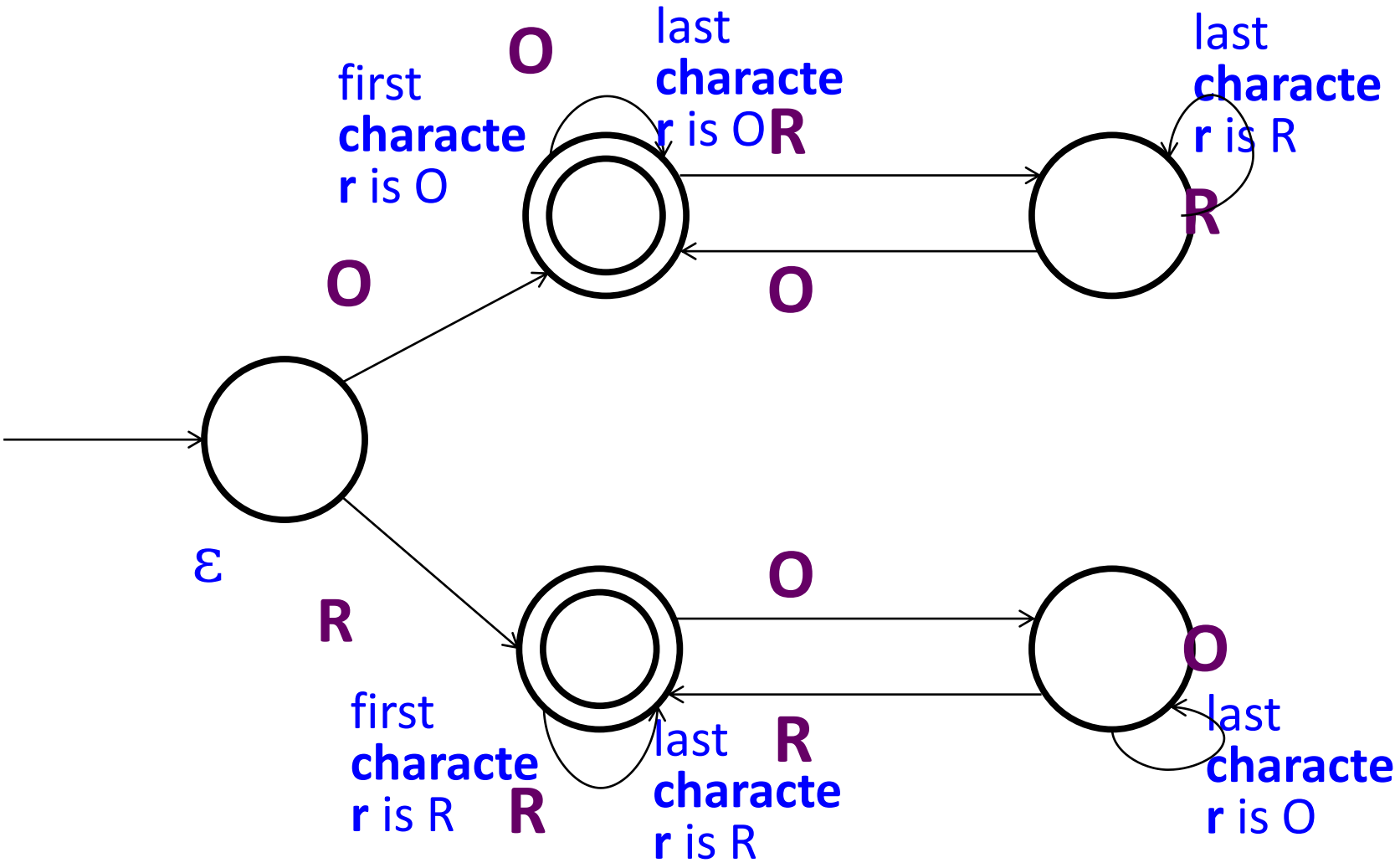
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



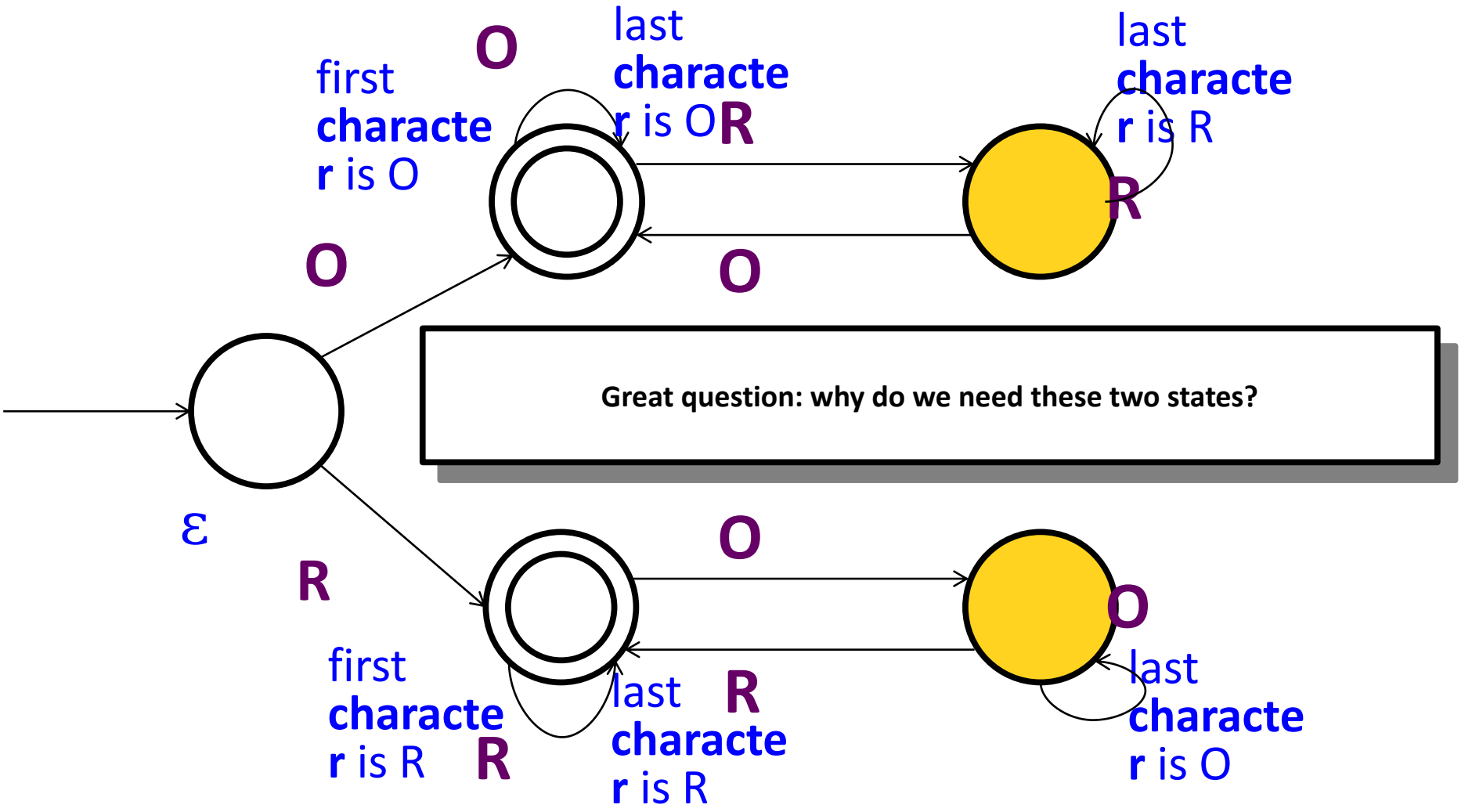
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



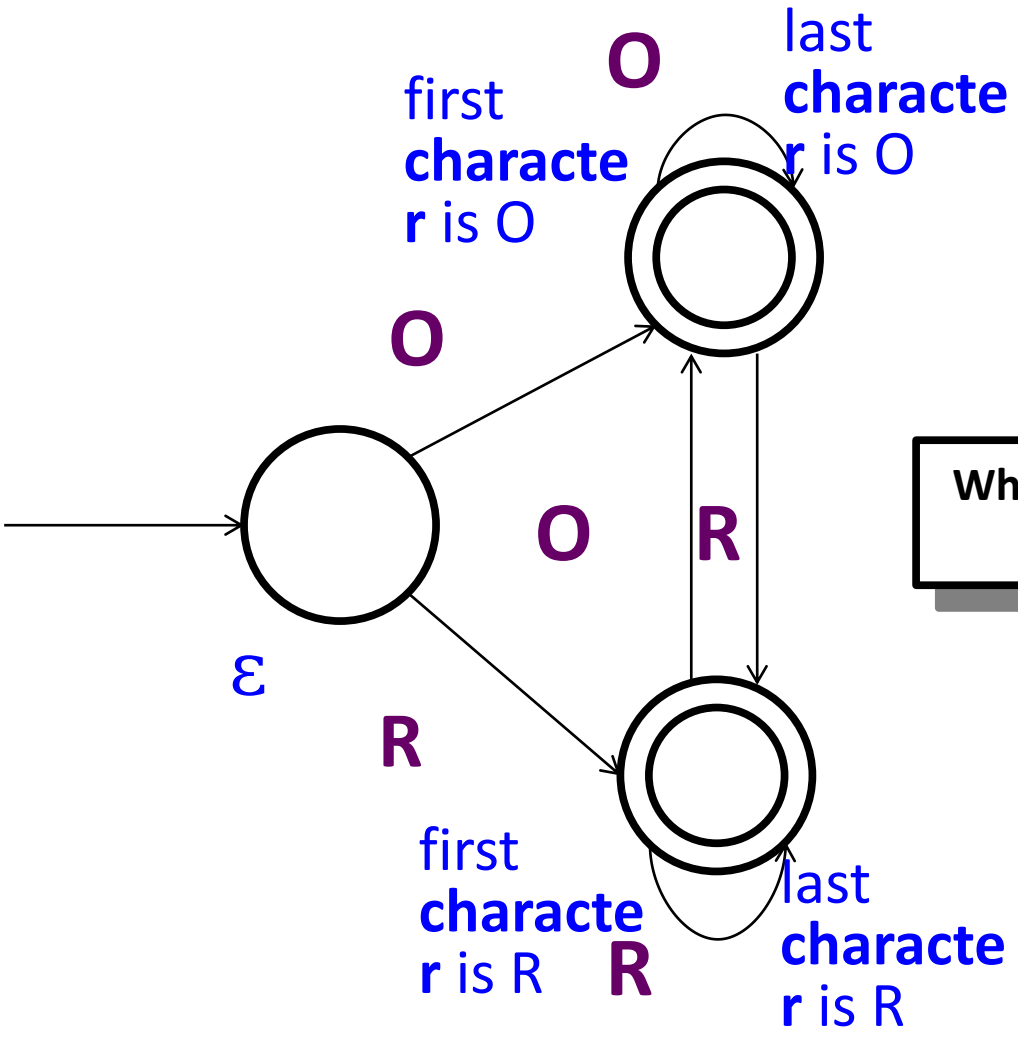
Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Oreo Sandwiches

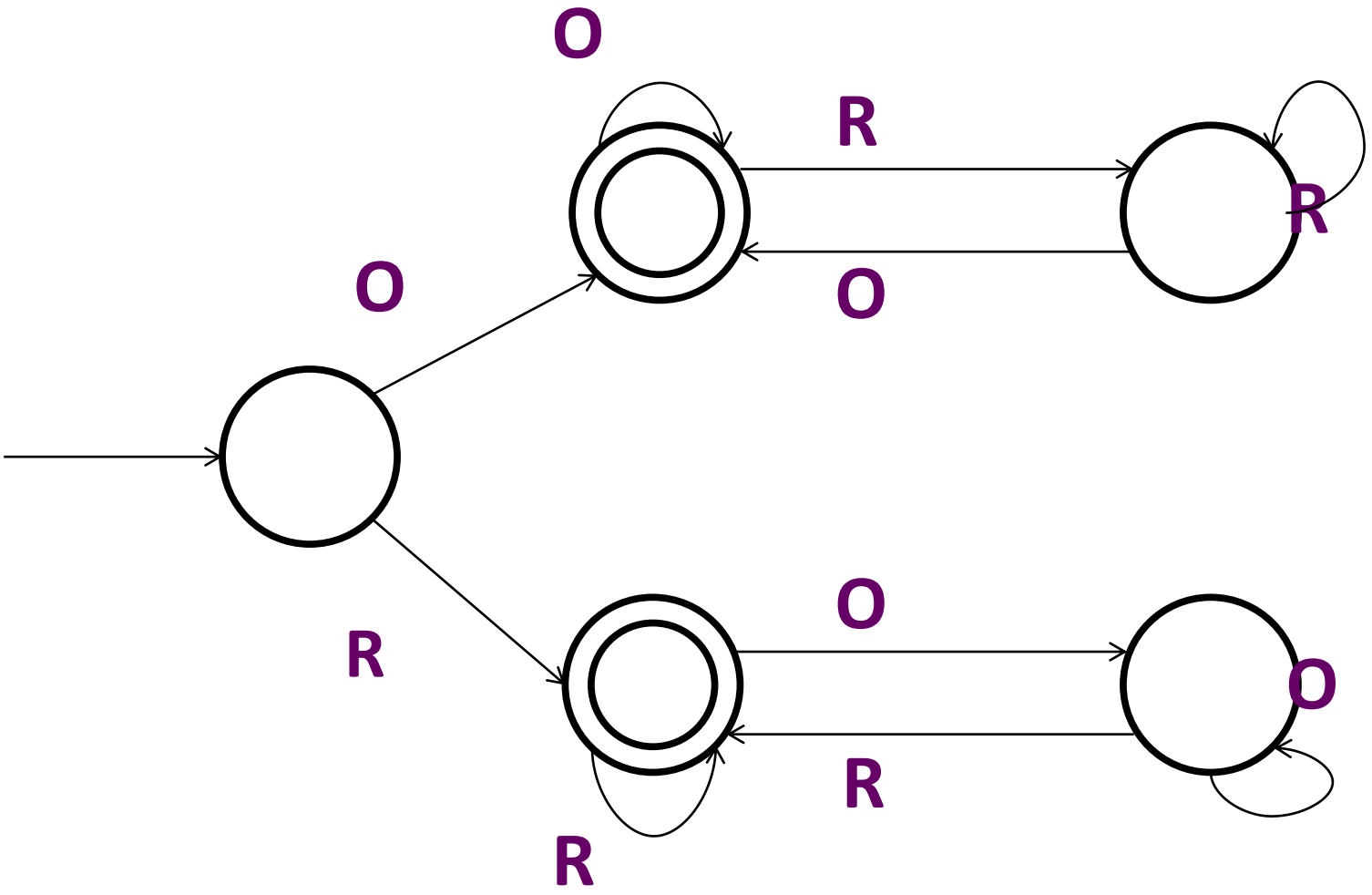
$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



Why can't we have a DFA that looks like this for this language?

Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the first and last character of w are the same $\}$



More Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

Design a regex for the language

$$L = \{ w \in \Sigma^* \mid w \neq \varepsilon \text{ and the characters of } w \text{ alternate between } \mathbf{O} \text{ and } \mathbf{R} \}$$

More Oreo Sandwiches

Let $\Sigma = \{$ **O**, **R** $\}$

Design a regex for the language

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

ORO $\in L$

ROROR $\in L$

ORORORORO $\in L$

OOR $\notin L$

RRRRR $\notin L$

ROROOROR $\notin L$

Designing Regexes

Write out some sample strings in the language and look for patterns:

Can I separate out the strings into two (or more) categories?

Union – find the pattern for each category, then union together

Can I break this problem down into solving some smaller subproblems?

Concatenation - find the pattern for each piece/subproblem, then concatenate together

Is there some sort of repeating structure?

Kleene star – find smallest repeating unit, then star that pattern

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

O

R

OR

RO

ORO

ROR

OROR

RORO

ORORO

ROROR

…

Here's one way we could design this regex

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

O

R

OR

RO

ORO

ROR

OROR

RORO

ORORO

ROROR

…

Can I separate out the strings into two (or more) categories?

Union – find the pattern for each category, then union together

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	**…**

Can I separate out the strings into two (or more) categories?

Union – find the pattern for each category, then union together

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	**…**

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	**…**

Can I break this problem down into solving some smaller subproblems?

Concatenation – find the pattern for each piece/subproblem, then concatenate together

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	**…**

Can I break this problem down into solving some smaller subproblems?

Concatenation – find the pattern for each piece/subproblem, then concatenate together

O(sequence of **RO**s)(possibly another **R**)

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	**…**

O(**RO**)*__R__?

Is there some sort of repeating structure?

Kleene star – find smallest repeating unit, then star that pattern

More Oreo Sandwiches

$L = \{ w \in \Sigma^* \mid w \neq \varepsilon$ and the characters of w alternate between **O** and **R** $\}$

Starts with **O**	Starts with **R**
O	**R**
OR	**RO**
ORO	**ROR**
OROR	**RORO**
ORORO	**ROROR**
…	…

O(**RO**)***R**? ∪ **R**(**OR**)***O**?

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.