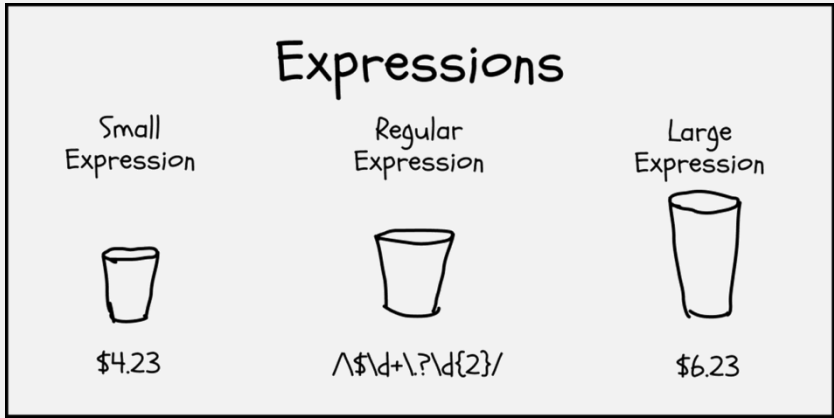
CS 420

Regular Expressions

Tuesday, October 13, 2026

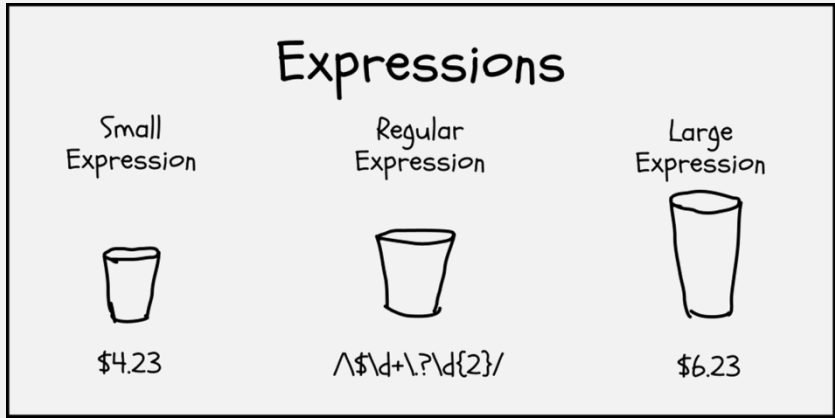
UMass Boston Computer Science



Announcements

- HW 4
  - ~~Due: Tue 10/13~~
  - Due (extended): **Thu 10/15 12:30pm**

- HW 5
  - Out: **Thu 10/15**
  - Due: **Thu 10/22 12:30pm**



Last Time

List of Closed Ops for Reg Langs (so far)

- ☑ Union
- ☑ Concatentation
- Kleene Star (repetition) ?

Star: $A^* = \{x_1 x_2 \ldots x_k | \; k \geq 0 \text{ and each } x_i \in A\}$

Kleene Star Example

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$.

If $A = \{\mathtt{good}, \mathtt{bad}\}$

"repeat" zero

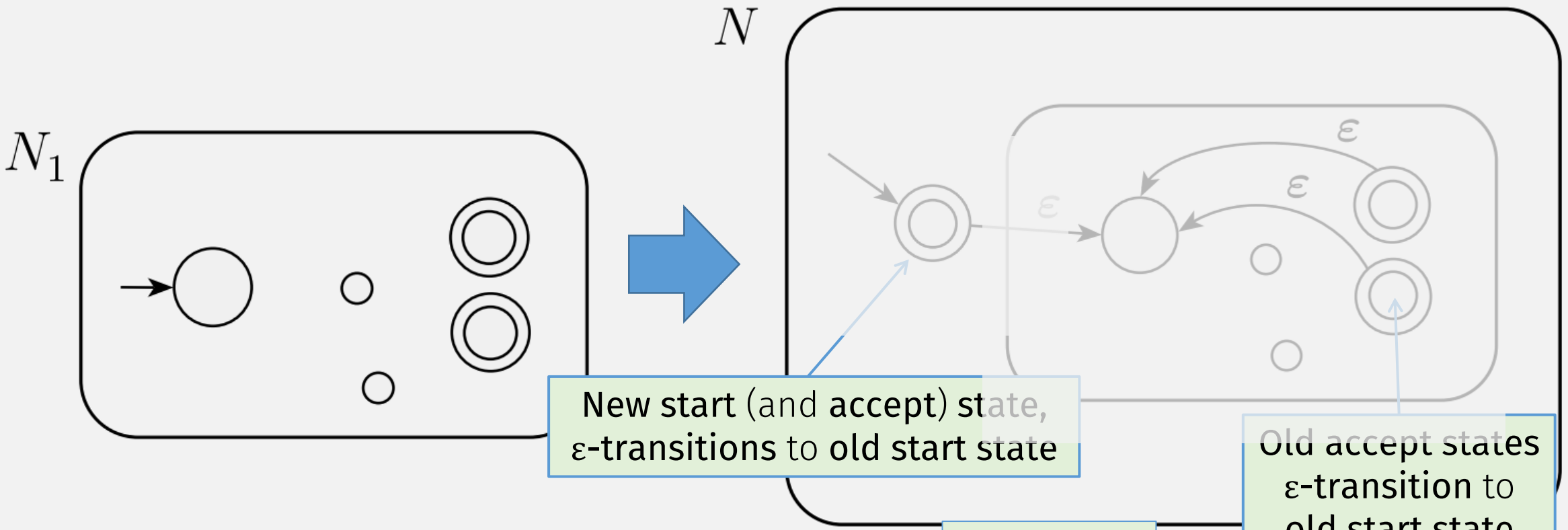
$$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$$

Note: **repeat strings** in A zero or more times

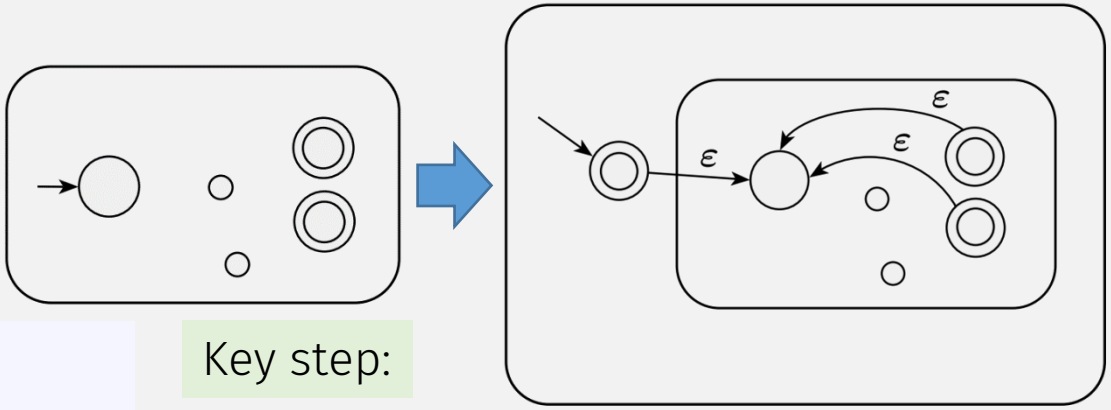
(this is an infinite language!)

Star: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Regular Langs Closed Under Kleene Star?



Regular Langs Closed Under Kleene Star?



THEOREM

The class of regular languages is closed under the star operation.

Key step:

$\text{STAR}_{\text{NFA}} : \text{NFA} \rightarrow \text{NFA}$

Where:

$N = \text{STAR}_{\text{NFA}}(N_1)$

$\text{LANGOF}(N) = \text{LANGOF}(N_1)^*$

Define function $\text{STAR}_{\text{NFA}} : \textbf{NFA} \rightarrow \textbf{NFA}$

$\text{STAR}_{\text{NFA}}(N_1) = N$, where

$(\ Q_1, \Sigma, \delta_1, q_1, F_1\ ) =$ **`get-NFA-parts`** N_1

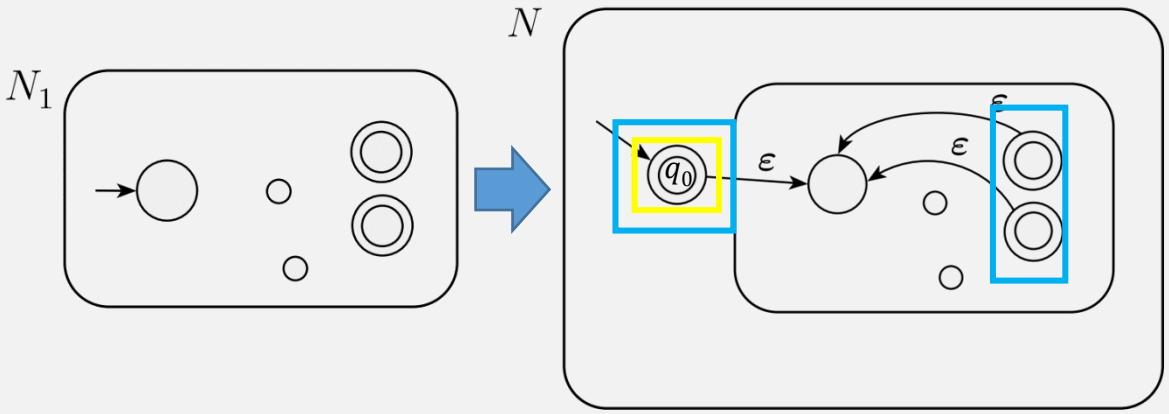
$N =$ **`mk-NFA`**$(\ Q, \Sigma, \delta, q_0, F\ )$ where

$Q = \{q_0\} \cup Q_1$

The state q_0 is ... a new start state!

$F = \{q_0\} \cup F_1$

Kleene star of a language must accept the empty string!



Define function $\text{STAR}_{\text{NFA}} : \text{NFA} \to \text{NFA}$

$\text{STAR}_{\text{NFA}}(N_1) = N$, where

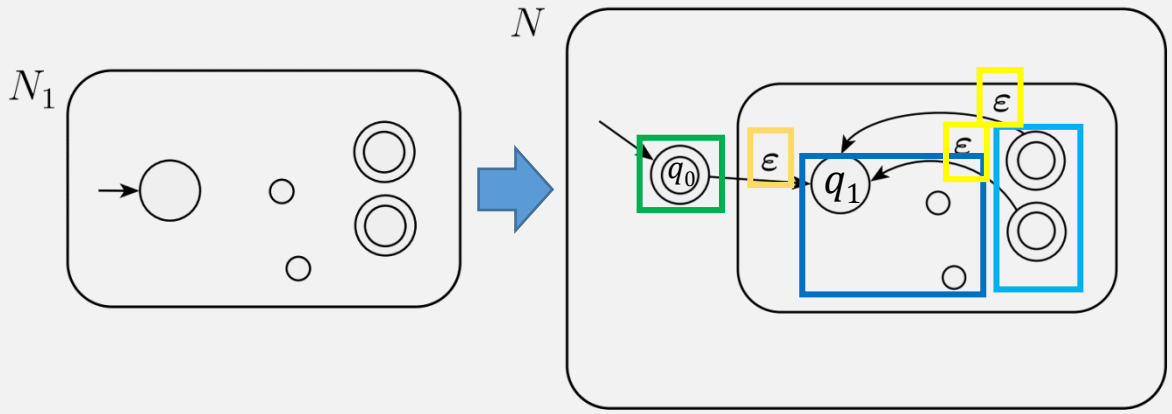
$(\ Q_1, \Sigma, \delta_1, q_1, F_1\ ) =$ **get-NFA-parts** N_1

$N =$ **mk-NFA**$(\ Q, \Sigma, \delta, q_0, F\ )$ where

$Q = \{q_0\} \cup Q_1$

The state q_0 is ... a new start state!

$F = \{q_0\} \cup F_1$



$\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ where: for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

"N does what N_1 does"

F_1 states might already have empty transitions!

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & ? \\ \delta_1(q,a) & ? \\ \delta_1(q,a) \cup \{q_1\} & ? \\ \{q_1\} & ? \\ \emptyset & ? \end{cases}$$

$q \in Q_1$ and $q \notin F_1$	
$q \in F_1$ and $a \neq \varepsilon$	
$q \in F_1$ and $a = \varepsilon$	Old accept states ε-transition to old start state
$q = q_0$ and $a = \varepsilon$	New start state ε-transitions to old start state
$q = q_0$ and $a \neq \varepsilon$.	New start state has only ε-transitions

Why These (Closed) Operations?

- **Union** $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$
- **Concatenation** $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$
- **Kleene star** (repetition) $A^* = \{x_1 x_2 \ldots x_k | \, k \geq 0 \text{ and each } x_i \in A\}$

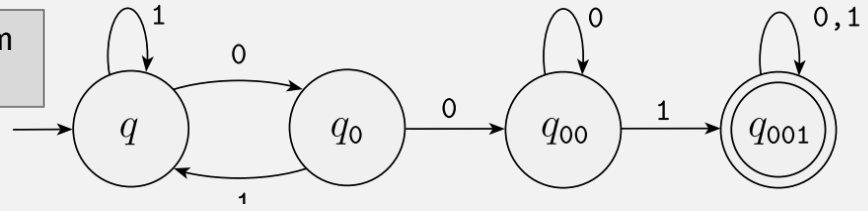
All regular languages can be **constructed from:**

- (language of) **single-char strings** (from some alphabet), …
- And these **three closed operations!**

e.g., {"a"} , {"b"} , …

So Far: Regular Language Representations

1. State diagram (NFA/DFA)



Formal description

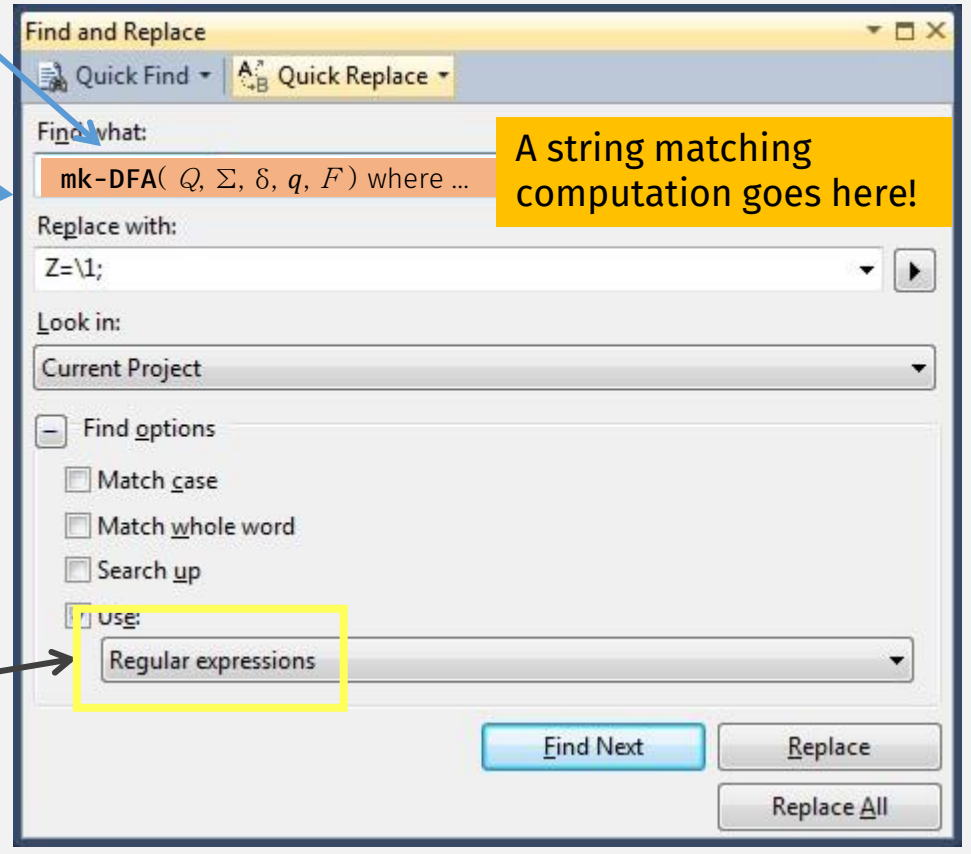
`mk-DFA`(Q, Σ, δ, q, F) where

2.

$$Q = \{q, q_0, q_{00}, q_{001}\}$$
$$\Sigma = \{0,1\}$$
$$\delta : Q \times \Sigma \rightarrow Q = \cdots$$
$$F = \{q_{001}\}$$

(hard to write)

Actually, it's a <u>real</u> programming language, for **text search** / **string matching** computations



<u>Our Running Analogy:</u>
- <u>Set</u> of all **regular languages** ~ a "programming language"
- <u>One</u> **regular language** (or any equiv representation) ~ a "**program**"

? 3. $\Sigma^*001\Sigma^*$

Need a more concise (textual) notation??

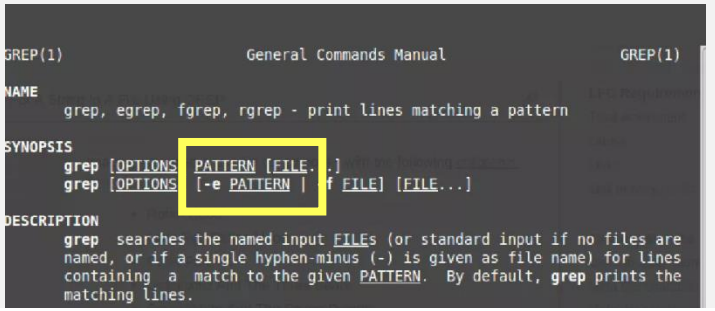
Regular Expressions:
A Widely Used Programming Language
(usually within tools / languages)

- Unix / Linux
- Java
- Python
- Web APIs

java.util.regex

Class Pattern

java.lang.Object
    java.util.regex.Pattern



Python » English 3.8.6rc1 Documentation » The Python Standard Library » Text Processing Services »

About regular expressions (regex)

Analytics supports regular expressions so you can create more flexible definitions for things like view filters, goals, segments, audiences, content groups, and channel groupings.

This article covers regular expressions in both Universal Analytics and Google Analytics 4.

In the context of Analytics, regular expressions are specific sequences of characters that broadly or narrowly match patterns in your Analytics data.

For example, if you wanted to create a view filter to exclude site data generated by your own employees, you could use a regular expression to exclude any data from the entire range of IP addresses that serve your employees. Let's say those IP addresses range from 198.51.100.1 - 198.51.100.25. Rather than enter 25 different IP addresses, you could create a regular expression like **198\.51\.100\.\d*** that matches the entire range of addresses.

— Regular expression operations

ce code: Lib/re.py

module provides regular expression matching operations similar to those found in Perl.

Why These (Closed) Operations?

- **Union** $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$
- **Concatenation** $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$
- **Kleene star** (repetition) $A^* = \{x_1 x_2 \ldots x_k | \, k \geq 0 \text{ and each } x_i \in A\}$

All **regular languages** can be constructed from:
- (language of) **single-char strings** (from some alphabet), and
- these **three closed operations!**

They are used to define **regular expressions!**

Regular Expressions: Formal Definition

"interface" type

DefRegEx

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

6 different constructors!

6 different ways to make something that IS-A **Regular** Expression

"itemized" data definition

i.e., this **definition** defines the following facts:
DefRegEx1: a IS-A **Regular** Expression
DefRegEx2: ε IS-A **Regular** Expression
DefRegEx3: ∅ IS-A **Regular** Expression
DefRegEx4: R_1 IS-A **Regular** Expression AND R_2 IS-A **Regular** Expression IFF $R_1 \cup R_2$ IS-A **Regular** Expression
DefRegEx5: R_1 IS-A **Regular** Expression AND R_2 IS-A **Regular** Expression IFF $R_1 \circ R_2$ IS-A **Regular** Expression
DefRegEx6: R_1 IS-A **Regular** Expression IFF R_1^* IS-A **Regular** Expression

Regular Expressions: Formal Definition

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

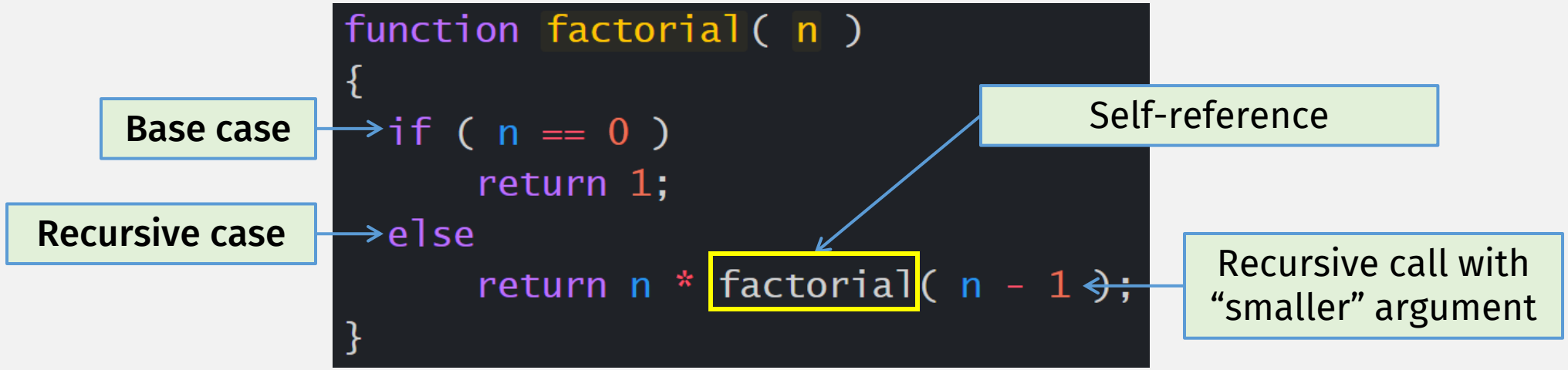
This is a recursive definition

Flashback: # Recursive Definitions

Recursive definitions are definitions with a self-reference

A valid recursive definition must have:
- **base case** and
- **recursive case** (with a **"smaller" self-reference**)

Flashback: # Recursive Definitions



Flashback: # Recursive Definitions

A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

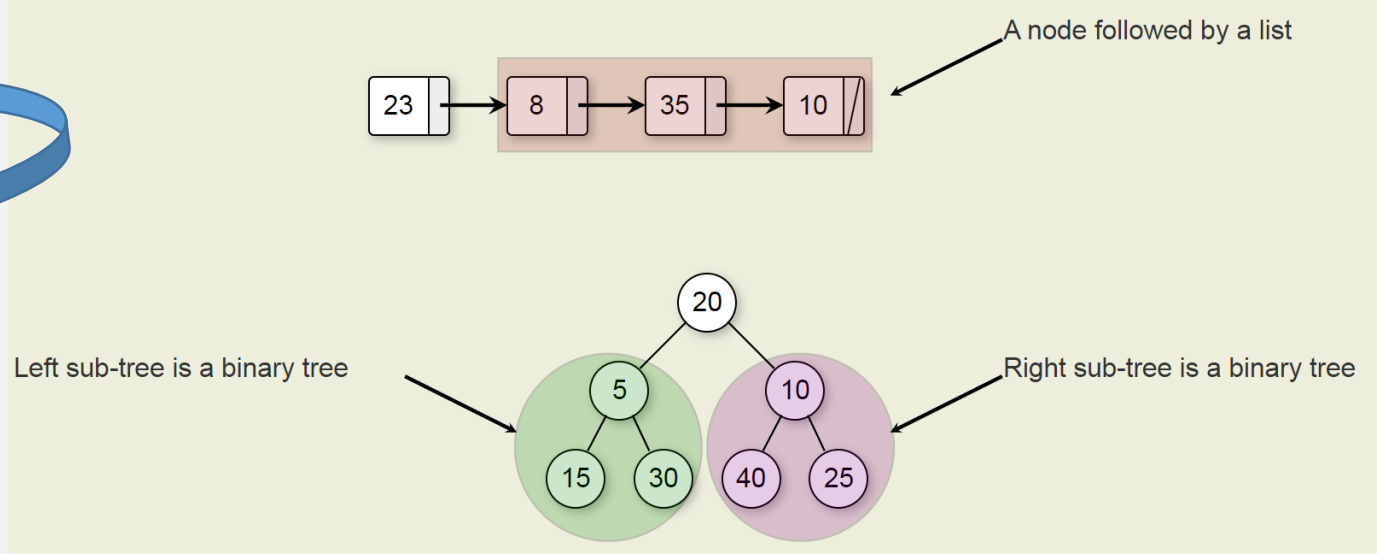
Self-reference

"smaller" argument

Flashback: Recursive Definitions

```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

Smaller self-reference



Q: Where's the base case??

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965.
>
> — Tony Hoare —

Data structures are commonly defined recursively

Regular Expressions: Formal Definition

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

3 Base Cases

(A lang containing a) **length-1 string**

(A lang containing) **the empty string**

(This is the 3rd **use** of the ε **symbol!**)

The empty set (i.e., a lang containing no strings)

union → 4.

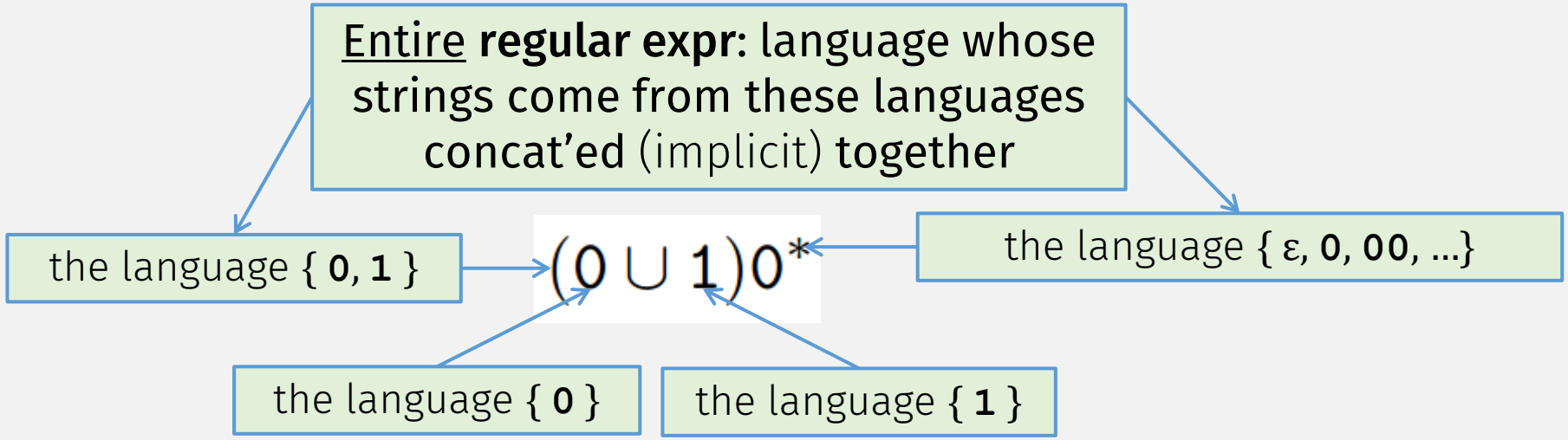
concat → 5.

star → 6.

3 Recursive Cases

Note:
- A **regular expression** describes a **language**
- The *set of all* **regular expressions** describes a *set of* **languages**

Regular Expression: Concrete Example



- Operator Precedence:
  - Parentheses
  - Kleene Star
  - Concat (sometimes use ∘, sometimes implicit)
  - Union

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

alphabet Σ is $\{0,1\}$

Regular Expression: More Examples

$0^*10^* = \{w | \ w \text{ contains a single } 1\}$

$\Sigma^*1\Sigma^* = \{w | \ w \text{ has at least one } 1\}$ Σ in regular expression = "any char"

$1^*(01^+)^* = \{w | \ \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$ let R^+ be shorthand for RR^*

$(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$ $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$

$1^*\emptyset = \emptyset$ $A \circ B = \{xy | \ x \in A \text{ and } y \in B\}$

nothing in B = nothing in $A \circ B$

$\emptyset^* = \{\varepsilon\}$ Star of any lang has ε

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

Regular Expressions: "Compute" Definition

A **regular expression** "describing" a **language** = computation!

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

A **regular expression** R describing a language L means LANGOF $R = L$

i.e., this **definition defines the following facts:**

DefRegLangCompute1: LANGOF a = { a }
DefRegLangCompute2: LANGOF ε = { ε }
DefRegLangCompute3: LANGOF ∅ = { }
DefRegLangCompute4: LANGOF $R_1 \cup R_2$ = LANGOF R_1 ∪ LANGOF R_2
DefRegLangCompute5: LANGOF $R_1 \circ R_2$ = LANGOF R_1 ∘ LANGOF R_2
DefRegLangCompute6: LANGOF (R_1^*) = (LANGOF R_1)*

Regular Expressions = Regular Langs?

R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

A **regular expression** describes a language

(But we have to prove it)

We would like to say:

- A **regular expression** describes a **regular** language
- The *set of all* **regular expressions** describes the *set of all* **regular languages**

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

(some) R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

(You should be able to sketch out the other steps of the full proof on your own)

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

Key step: convert regular expression → equivalent NFA!

How to show that a language is regular?

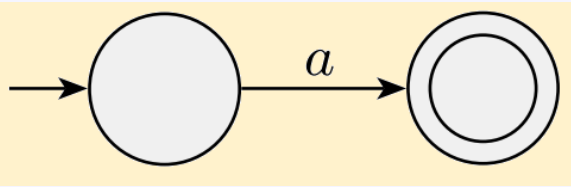
Construct a **DFA** *or* **NFA!**

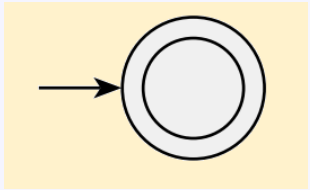
RegExpr→NFA (sketch)

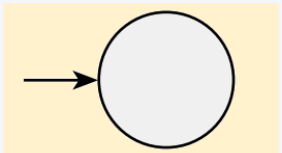
(this function is recursive!)

R is a ***regular expression*** if R is

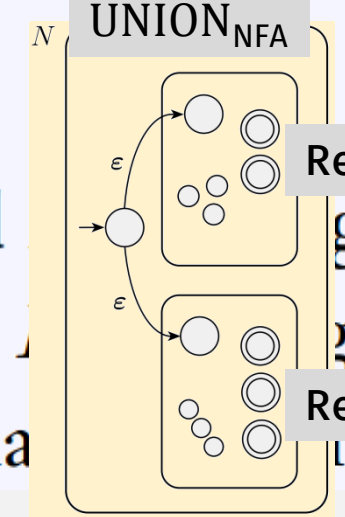
1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,
6. (R_1^*), where R_1 is a regular expression.





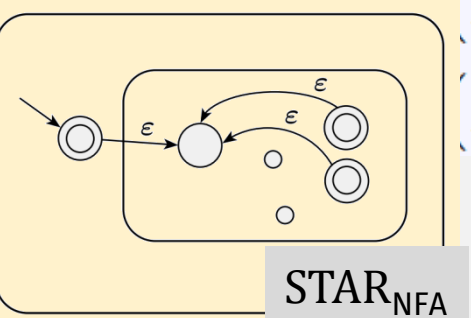


UNION$_{NFA}$

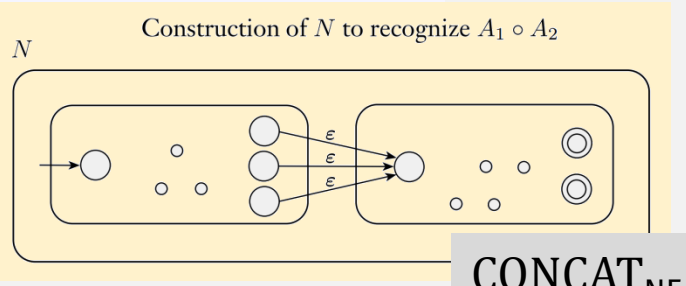


RegExpr→NFA(R_1)

RegExpr→NFA



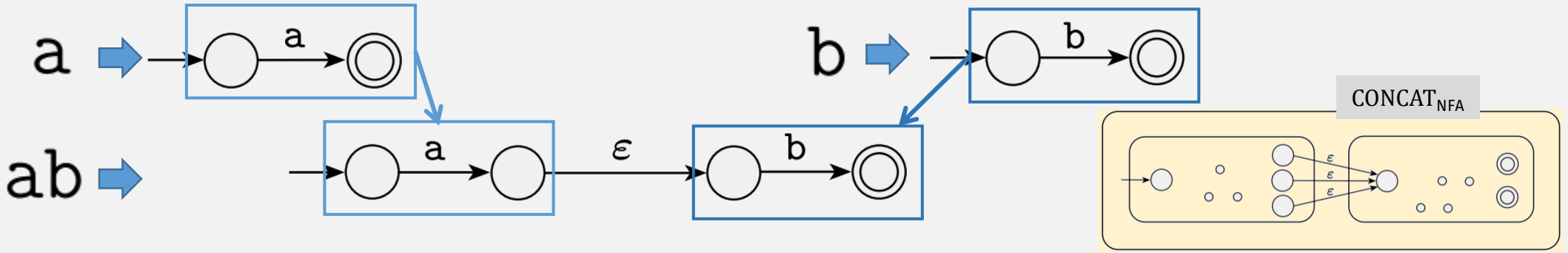
STAR$_{NFA}$



CONCAT$_{NFA}$

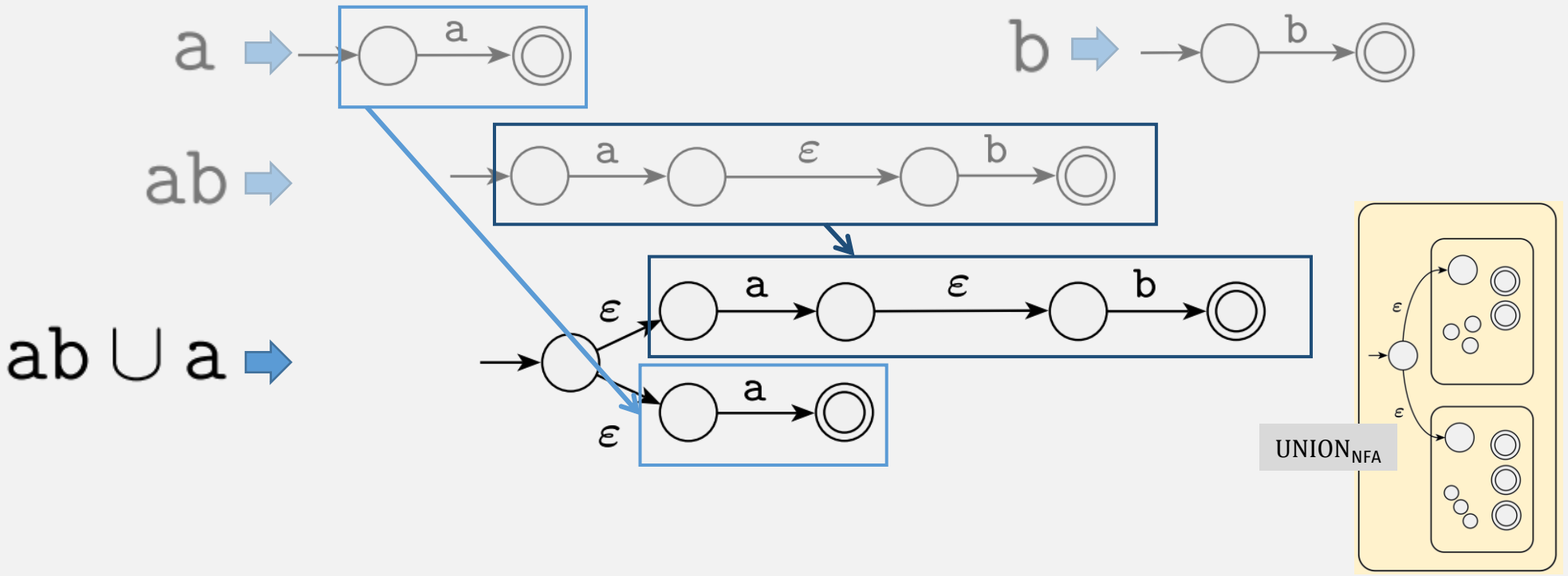
RegExpr→NFA: Example

convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA ... step by step



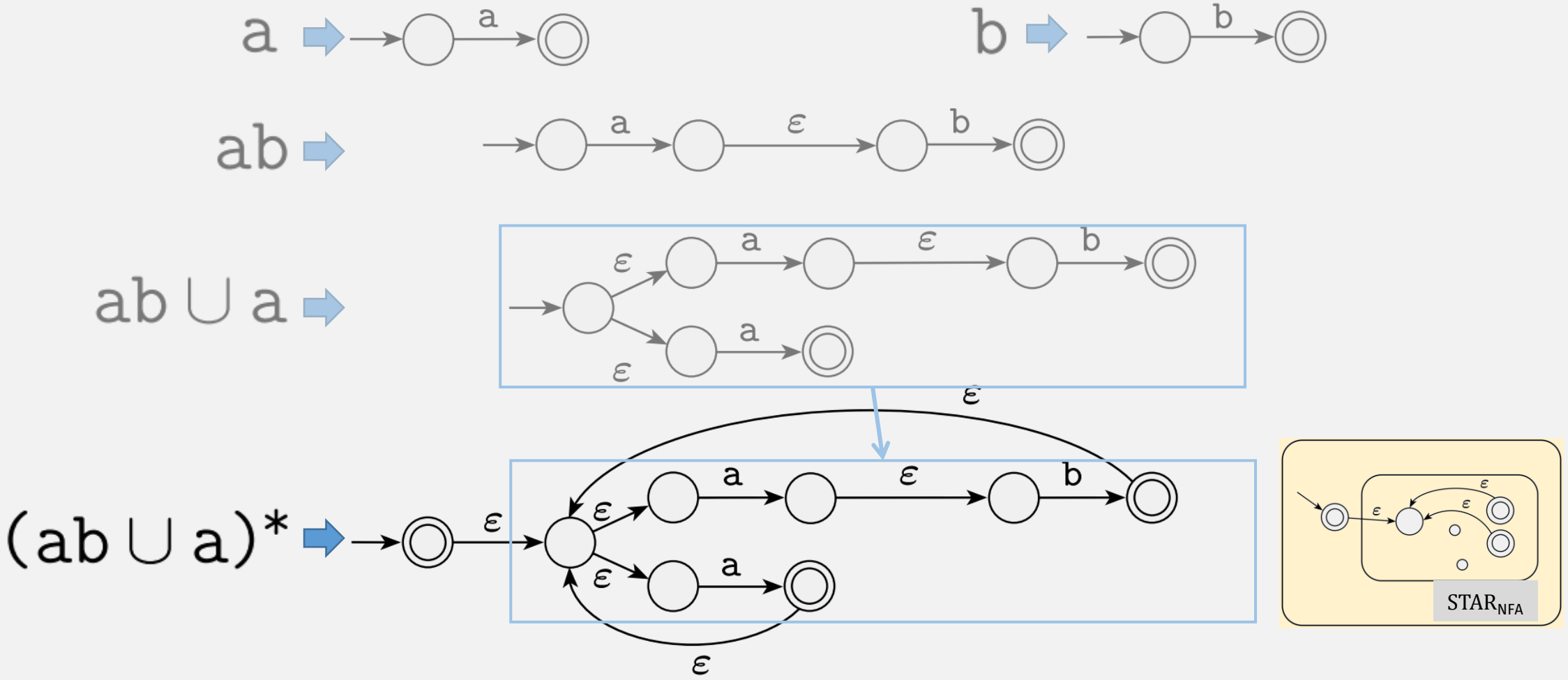
RegExpr→NFA: Example

convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA



RegExpr→NFA: Example

convert the regular expression $(ab \cup a)^*$ to an NFA



L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

Key step: Convert an ~~DFA~~ or NFA → equivalent Regular Expression

GNFA

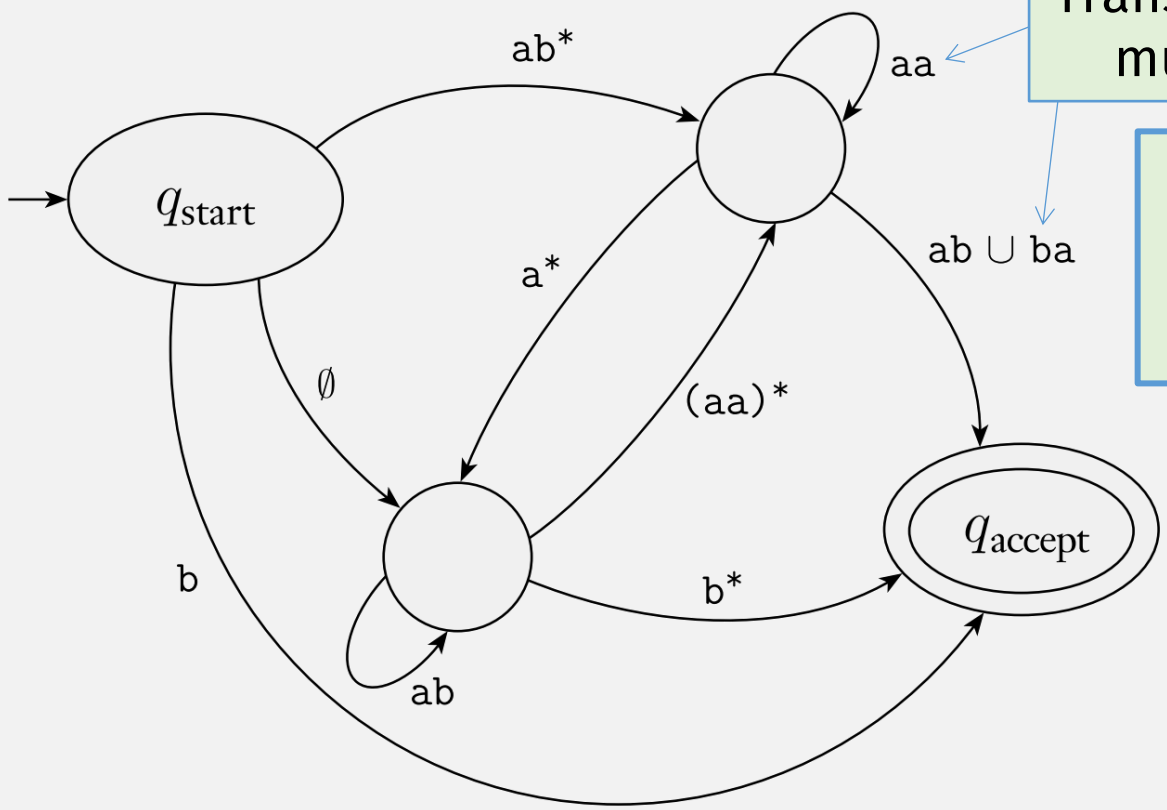
First, we need another kind of finite automata: a **GNFA**

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

☑ Key step: convert regular expression → equivalent NFA!

(You should be able to sketch out the other steps of the full proof on your own)

Generalized NFAs (GNFAs)



Transition can read multiple chars

plain NFA
= GNFA with single char regular expr transitions

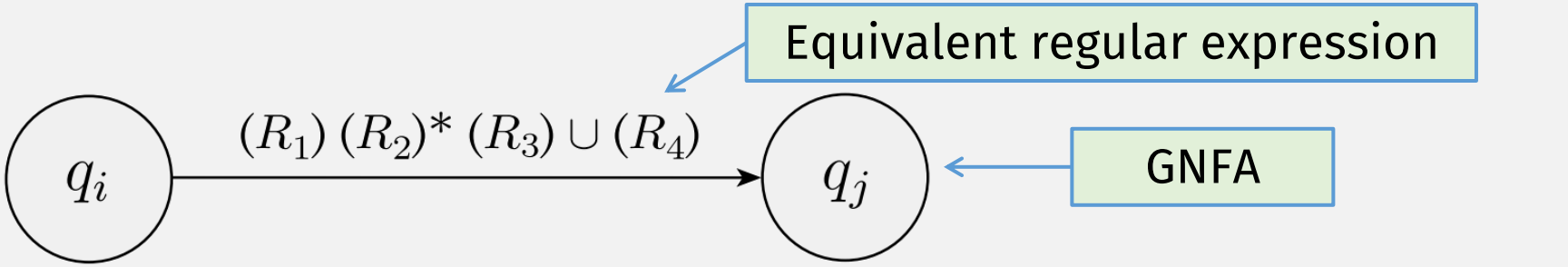
Goal: convert **GNFAs** to equivalent **Regular Exprs**

- GNFA = NFA with regular expression transitions

GNFA→RegExpr function : (sketch)

On **GNFA** input G:

- If G **has 2 states,** **`return`** **the regular expression** (on the transition), e.g.:



Could there be less than 2 states?

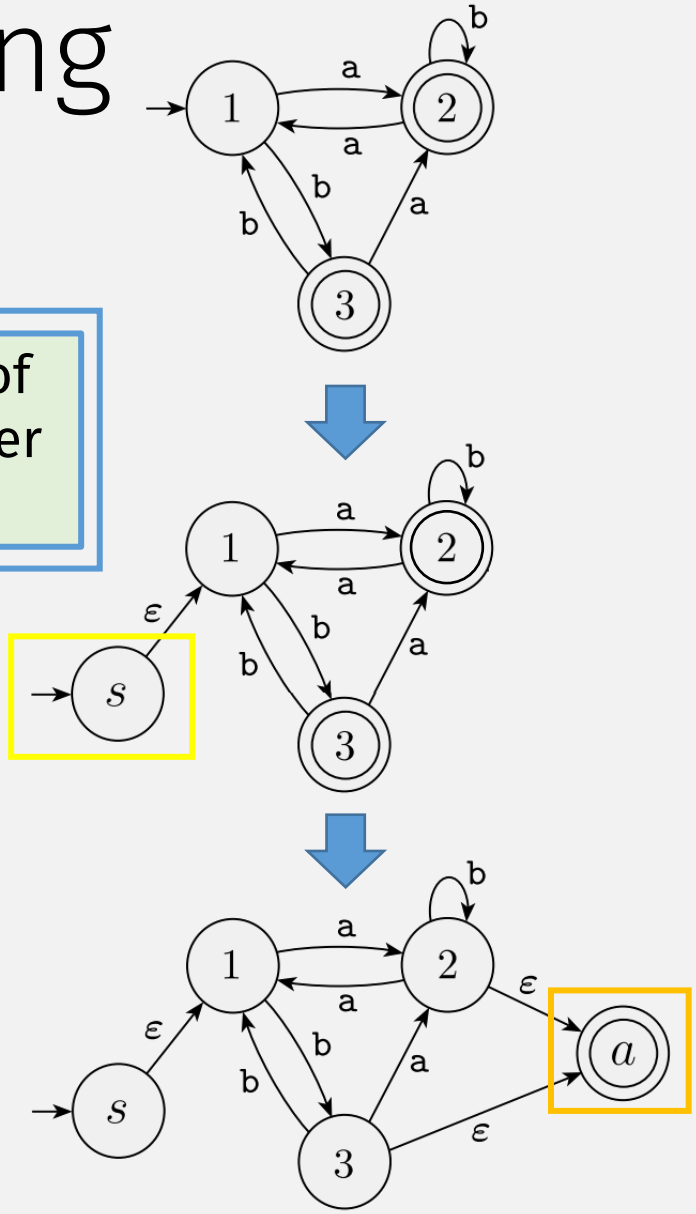
GNFA→RegExpr Preprocessing

- Modify input machine to have:

Does this change the language of the machine? i.e., are before/after machines equivalent?

- New start state:
  - No incoming transitions
  - ε transition to old start state
- New, single accept state:
  - With ε transitions from old accept states

Modified machine always has 2+ states:
- New start state
- New accept state

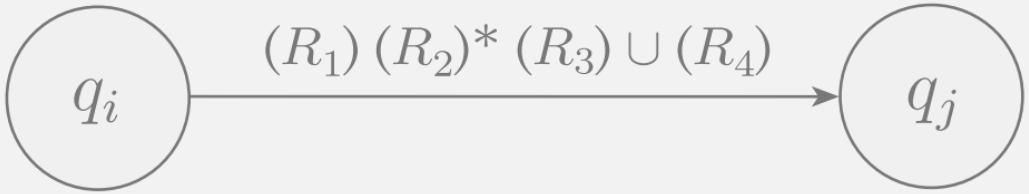


GNFA→RegExpr function (recursive)

On **GNFA** input G:

Base Case

- If G has 2 states, **`return`** the regular expression (from transition), e.g.:

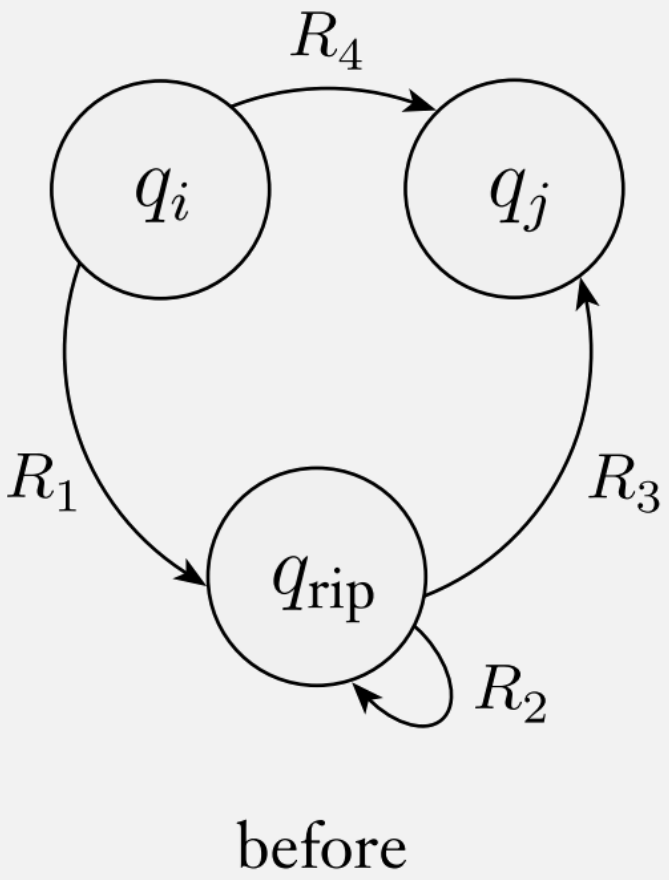


Recursive Case

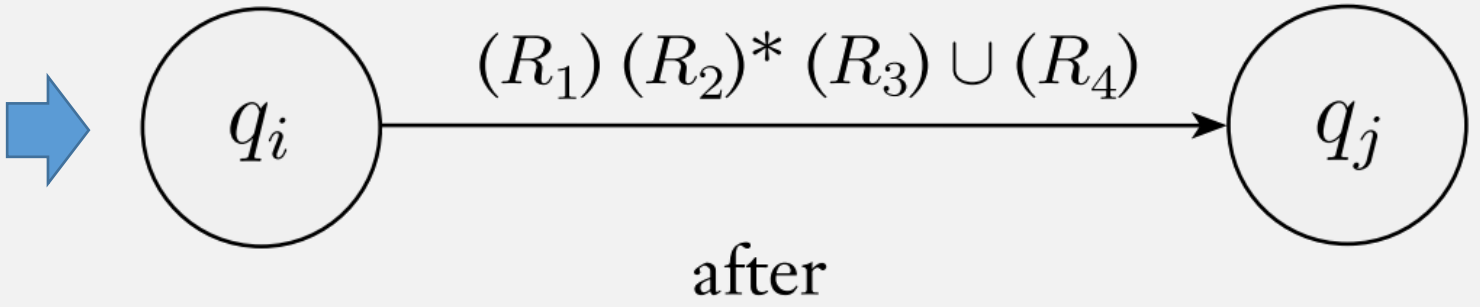
- Else:
  - "Rip out" one state
  - "Repair" the machine to get an equivalent GNFA G'
  - Recursively call **GNFA→RegExpr**(G')

Recursive definitions have:
- base case and
- recursive case
  (with "smaller" self-reference)

GNFA→RegExpr: “Rip/Repair” step



before

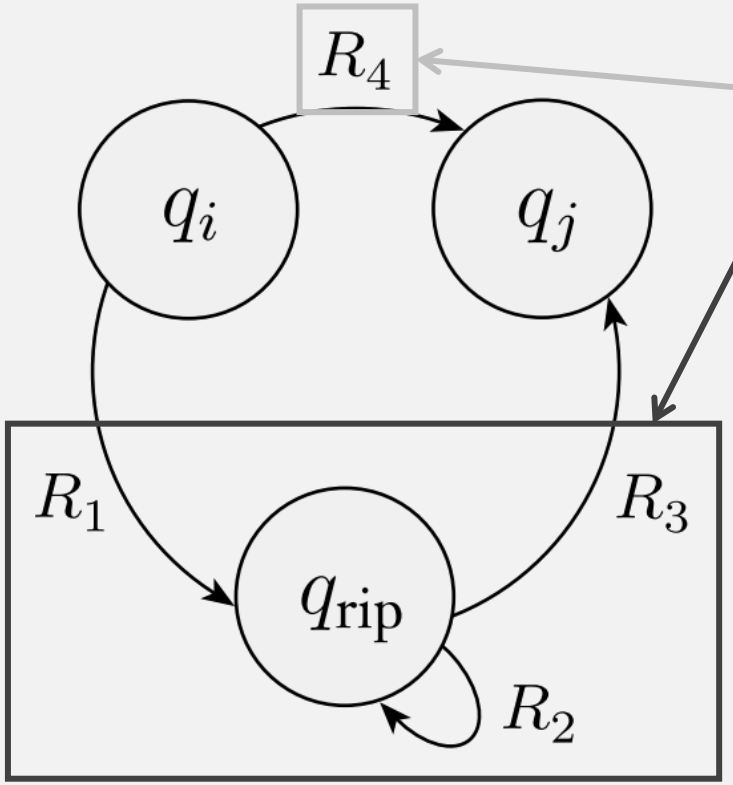


after

To convert GNFA -> regular expression:

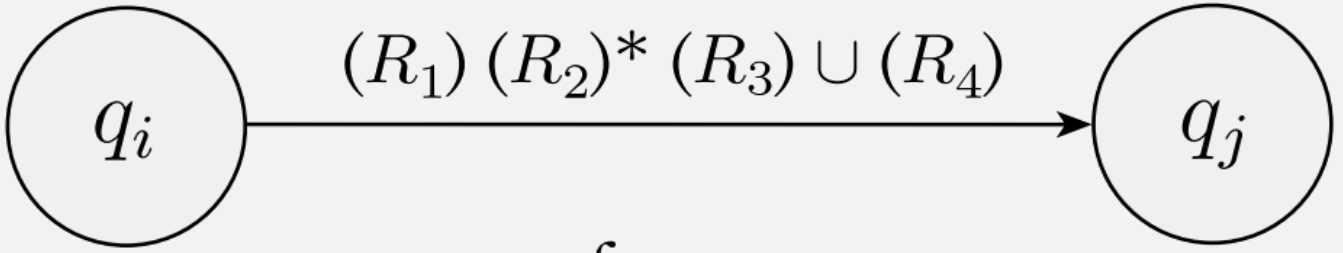
1. “rip out” a state,
2. “repair” so machine is equivalent
3. repeat until only 2 states remain

GNFA→RegExpr: "Rip/Repair" step



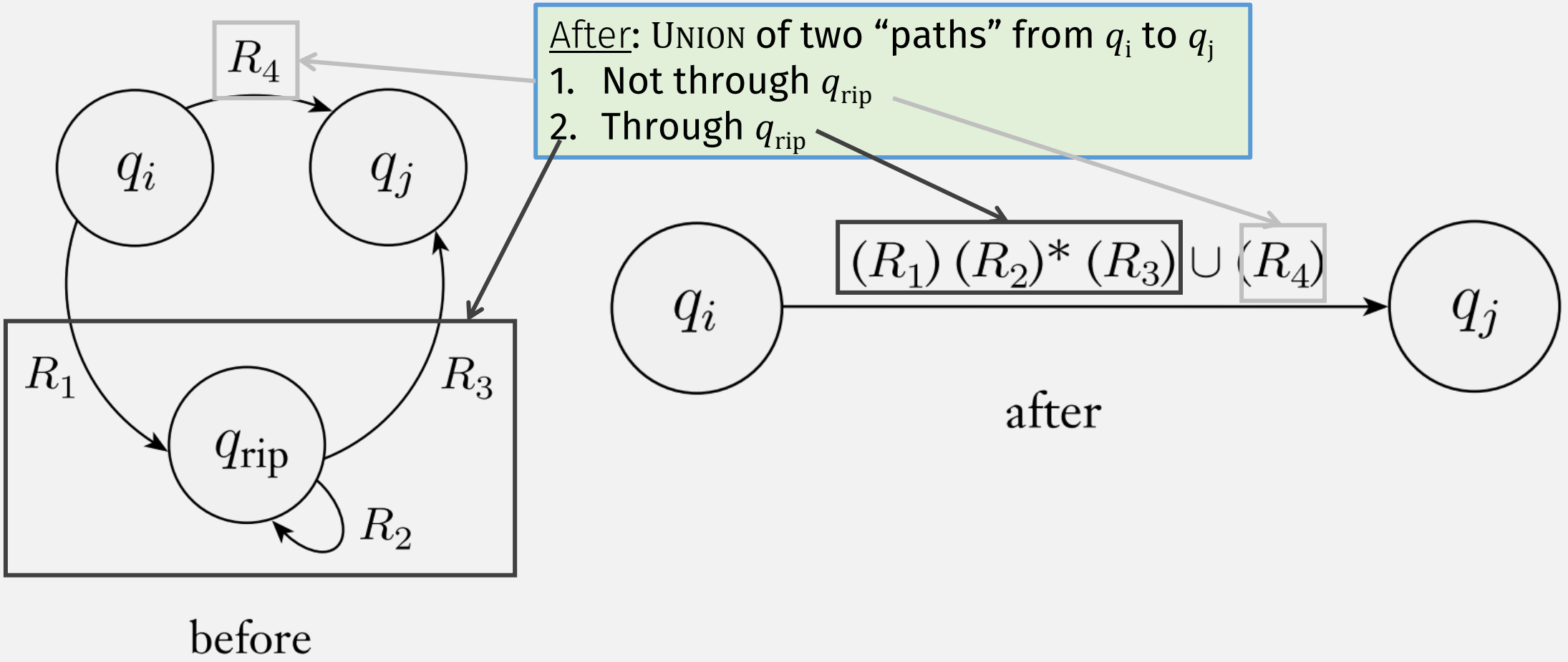
before

Before: two paths from q_i to q_j:
1. Not through q_{rip}
2. Through q_{rip}

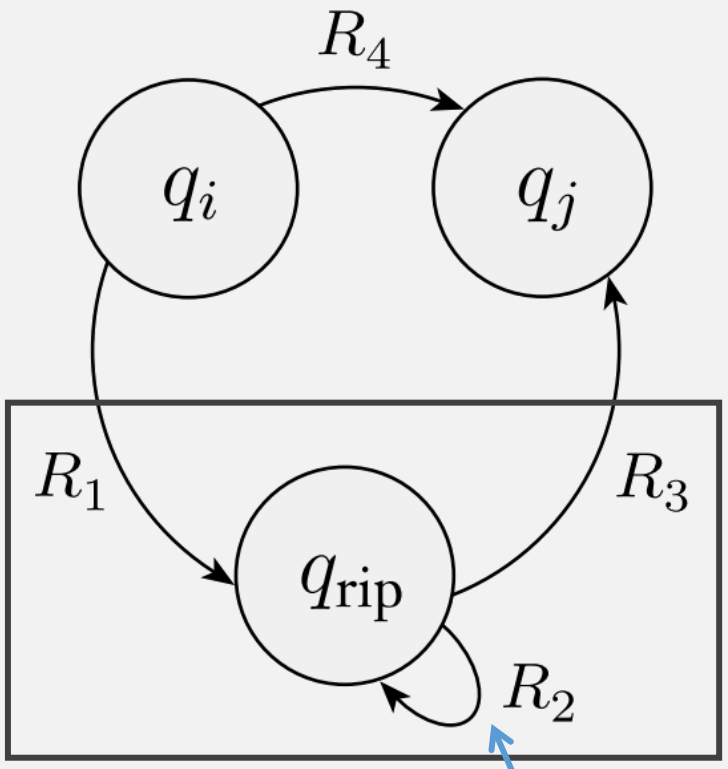


after

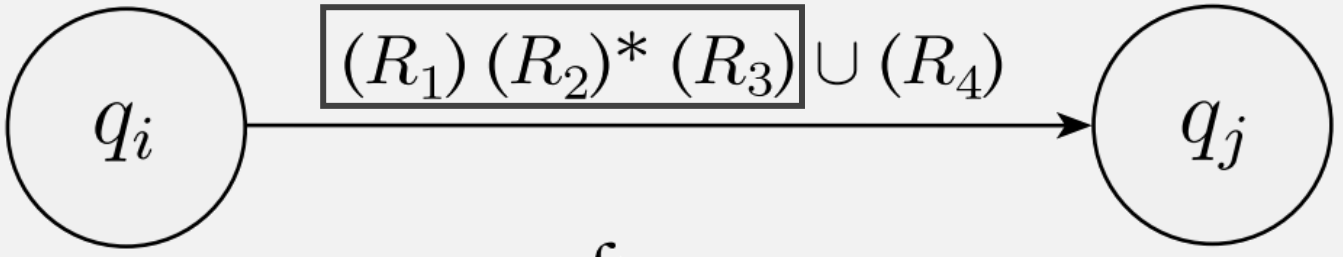
GNFA→RegExpr: "Rip/Repair" step



GNFA→RegExpr: “Rip/Repair” step



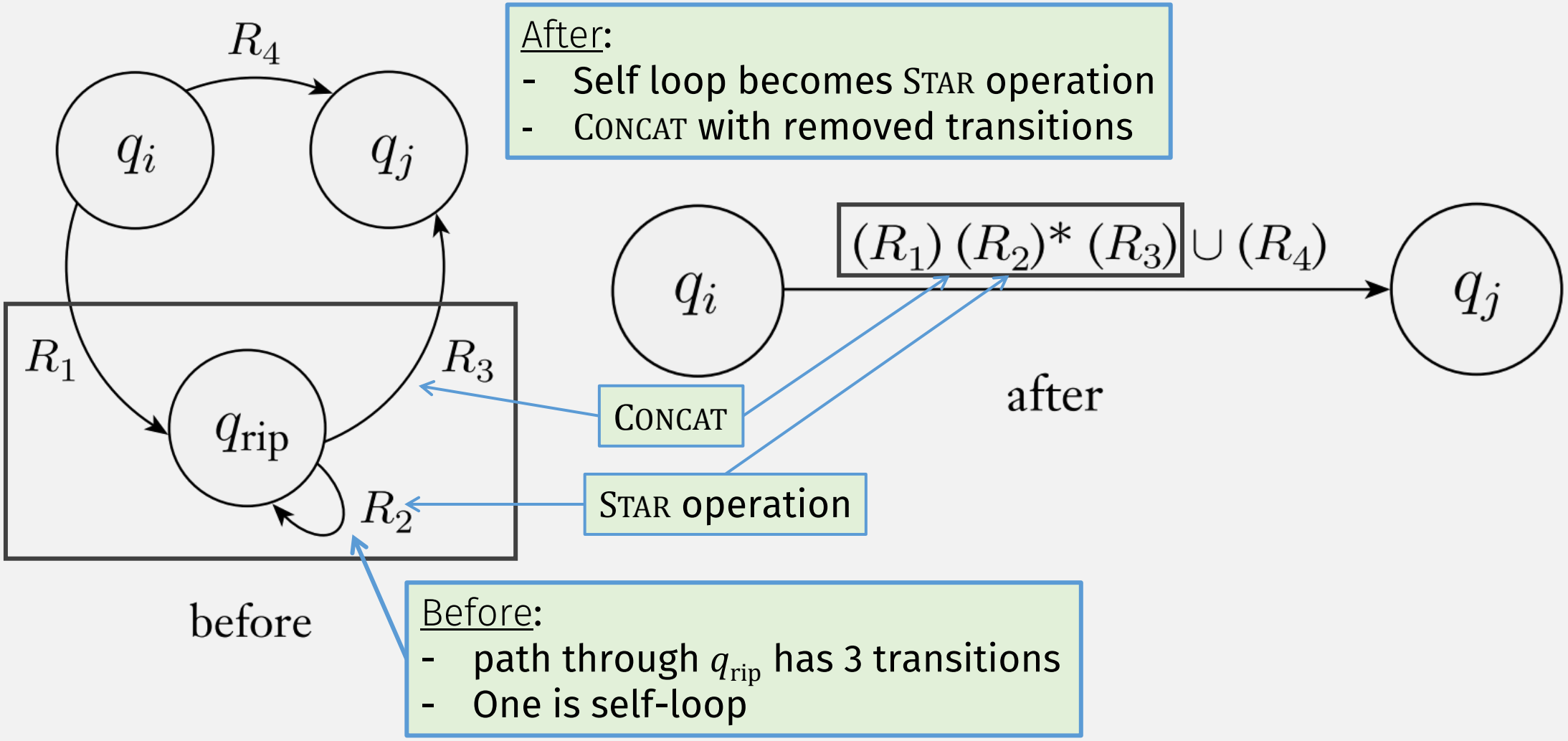
before



after

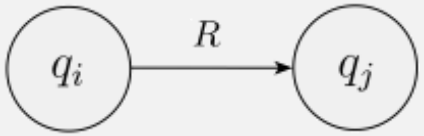
Before:
- path through q_{rip} has 3 transitions
- One is self-loop

GNFA→RegExpr: "Rip/Repair" step



`GNFA→RegExpr` : GNFA → NFA (more formally)

Base Case

`GNFA→RegExpr`  = R

(= 2 states)

Recursive Case

`GNFA→RegExpr` G = `GNFA→RegExpr` (“Rip/Repair” G)

(> 2 states)

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

Key step: Convert a GNFA → equivalent Regular Expression

☑

???

This time, let's **really prove equivalence!**
(we previously "proved" it with an **Examples Table**)

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

☑ Key step: convert regular expression → equivalent NFA!

GNFA→RegExpr Correctness

- **Correct** = **input** and **output** are **equivalent**
- **Equivalent** = the **language does not change** (same strings)**!**

Statement to Prove:

???

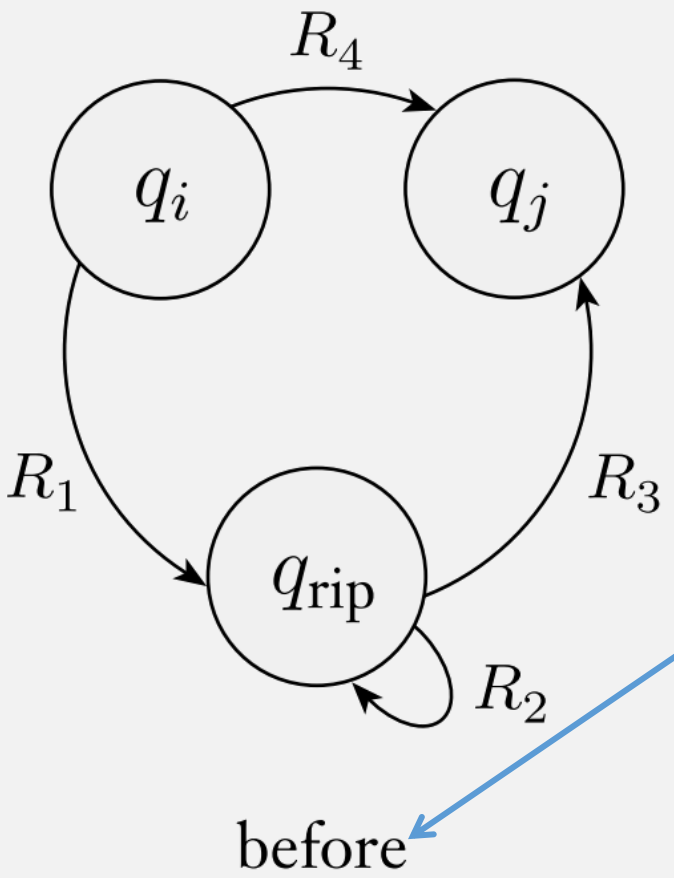
$$\text{LANGOF}(G) = \text{LANGOF}(R)$$

We are ready to **really prove equivalence!**
(we previously "proved" it with some examples)

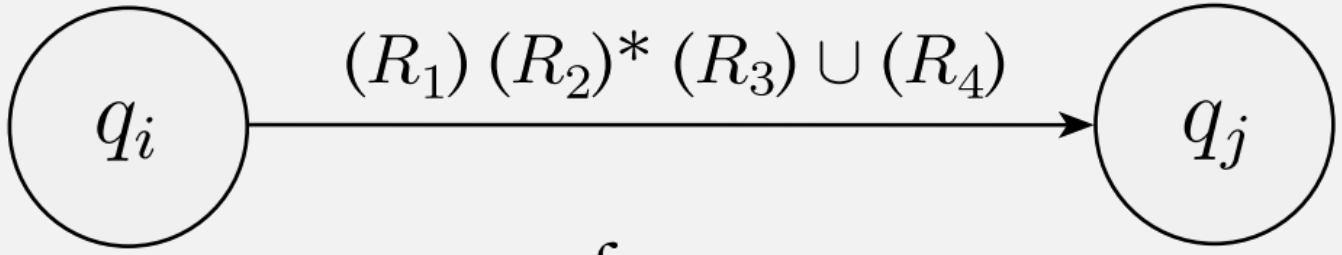
- where:
  - G = a GNFA
  - R = a Regular Expression = **GNFA→RegExpr**(G)

- Key step: **the rip/repair step**

GNFA→RegExpr: Rip/Repair Correctness



before



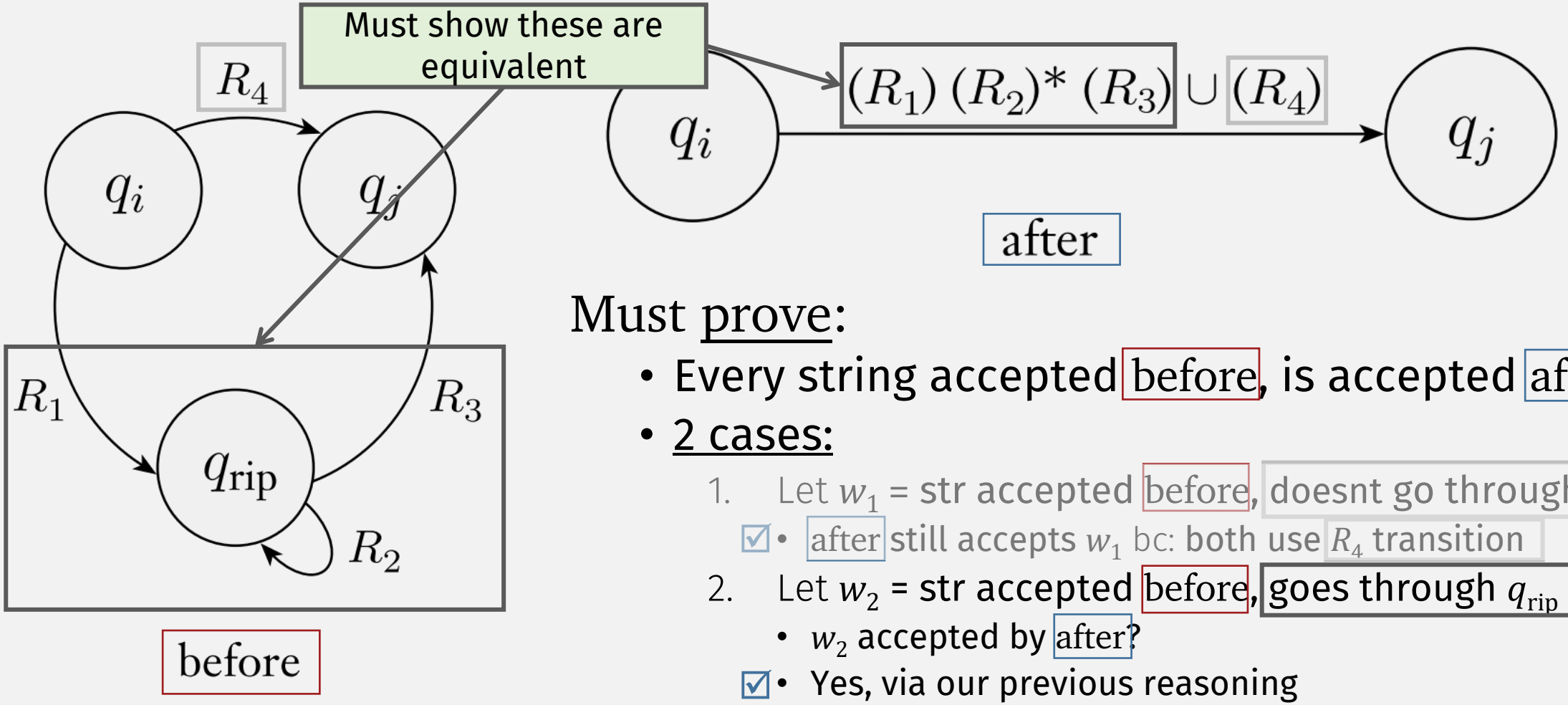
after

Must show these are equivalent

Equivalent =
same language =
accepts the same strings

So: LangOf(G) = LangOf (Rip/Repair(G))

GNFA→RegExpr: Rip/Repair Correctness



Must prove:

- Every string accepted before, is accepted after
- 2 cases:
  1. Let w_1 = str accepted before, doesnt go through q_{rip}
     - ☑ after still accepts w_1 bc: both use R_4 transition
  2. Let w_2 = str accepted before, goes through q_{rip}
     - w_2 accepted by after?
     - ☑ Yes, via our previous reasoning

(Similar to non-concrete Examples Table!)

GNFA→RegExpr Equivalence

- **Equivalent** = the **language does not change** (i.e., same set of strings)**!**

Statement to Prove: input output ???

$$\text{LANGOF}(G) = \text{LANGOF}(R)$$

This time, let's **really prove equivalence!** (we previously "proved" it with some examples)

- where:
  - G = a GNFA
  - R = a Regular Expression = **GNFA→RegExpr**(G)

Language could be infinite set of strings!

(how can we show **equivalence** for a possibly **infinite set of strings**?)

Recursion!