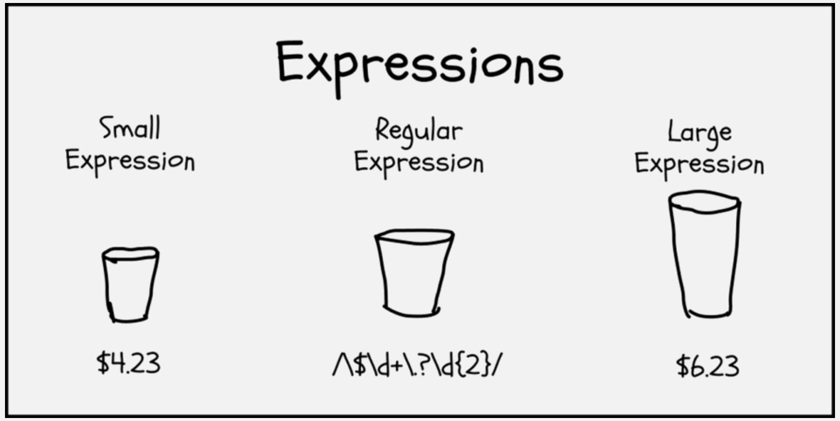
UMB CS 420

Regular Expressions

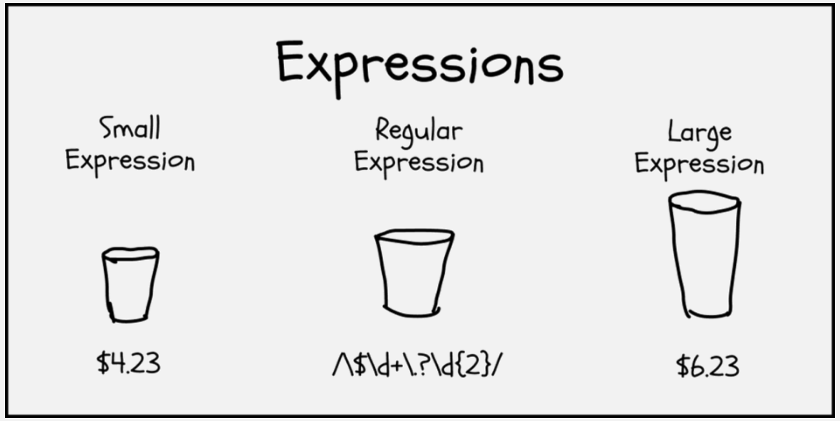
Wednesday February 28, 2024



Announcements

- HW 3 out
  - Due Mon 3/4 12pm EST (noon)

- Reminder: Use Gradescope re-grade request for all grading questions / complaints!



List of Closed Ops for Reg Langs (so far)

☑ • Union $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$

☑ • Concatentation $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$

• 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 $\{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$.

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

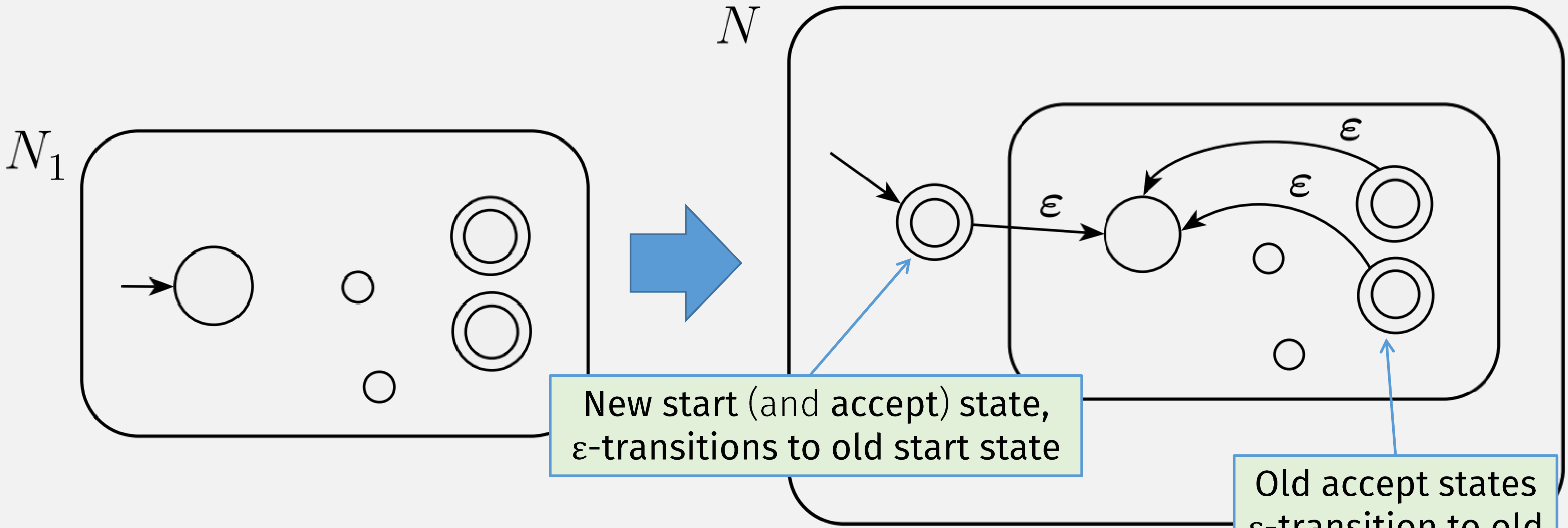
$$A^* = \begin{array}{l}\{\varepsilon, \texttt{good}, \texttt{bad}, \texttt{goodgood}, \texttt{goodbad}, \texttt{badgood}, \texttt{badbad}, \\ \texttt{goodgoodgood}, \texttt{goodgoodbad}, \texttt{goodbadgood}, \texttt{goodbadbad}, \ldots\}\end{array}$$

Note: **repeat <u>zero or more times</u>**

(this is an infinite language!)

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

Kleene Star is Closed for Regular Langs?



New start (and accept) state,
ε-transitions to old start state

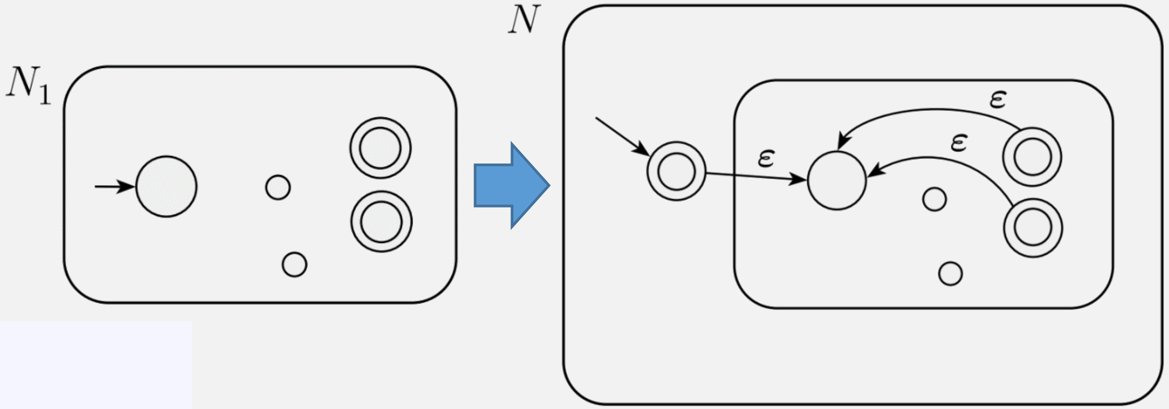
Old accept states ε-transition to old start state

Recognizes language A

Recognizes language A^*

HW4 exercise

Kleene Star is Closed for Regular Langs



THEOREM

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

Why These (Closed) Operations?

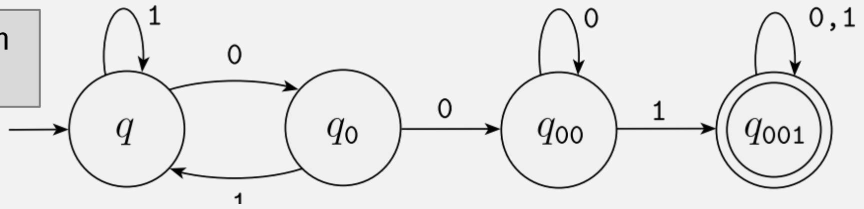
- **Union** $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$
- **Concatenation** $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$
- **Kleene star** (repetition) $A^* = \{x_1 x_2 \ldots x_k | \, k \geq 0 \text{ and each } x_i \in A\}$

<u>All</u> **regular languages** can be <u>constructed</u> **from:**

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

So Far: Regular Language Representations

1. State diagram (NFA/DFA)



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

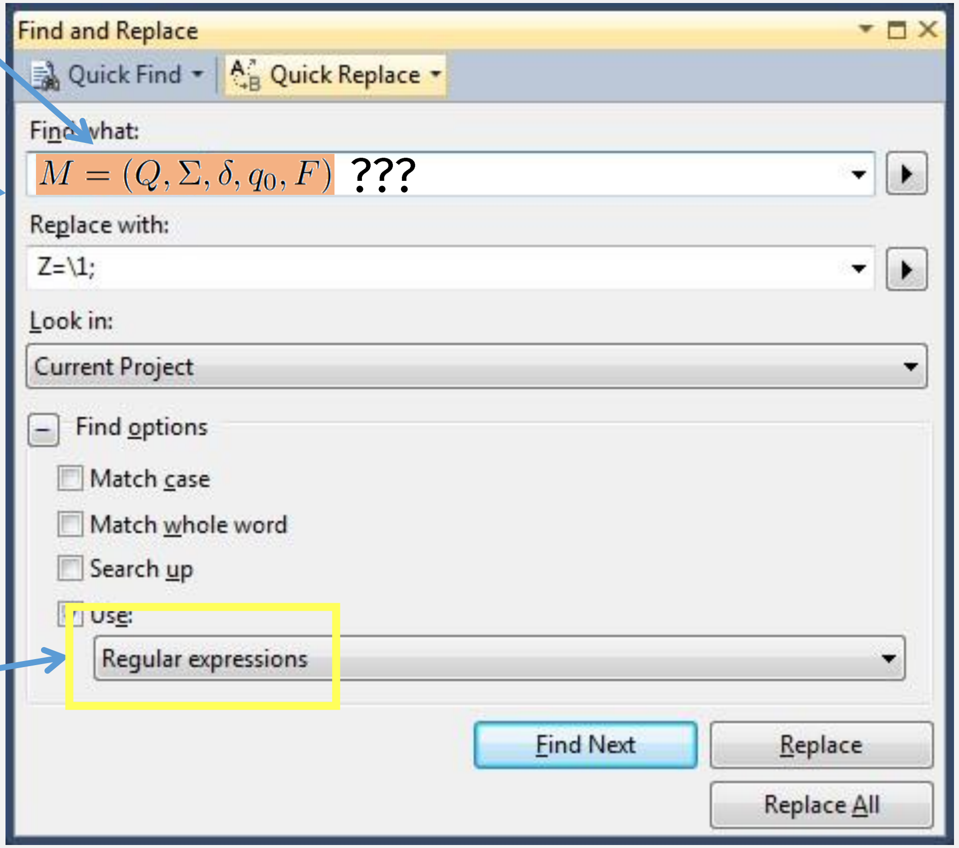
2. Formal description

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state
5. $F = \{q_2\}$

(doesn't fit)



<u>Our Running Analogy:</u>
- <u>Set</u> of all **regular languages** ~ a "programming language"
- <u>One</u> **regular language** ~ a "program"

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

Need a more concise (textual) notation??

Regular Expressions:
A Widely Used Programming Language
(in other tools / languages)

- Unix / Linux
- Java
- Python
- Web APIs

java.util.regex

Class Pattern

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

```
GREP(1)                 General Commands Manual                 GREP(1)

NAME
       grep, egrep, fgrep, rgrep - print lines matching a pattern

SYNOPSIS
       grep [OPTIONS PATTERN [FILE. .]
       grep [OPTIONS [-e PATTERN | f FILE] [FILE...]

DESCRIPTION
       grep  searches the named input FILEs (or standard input if no files are
       named, or if a single hyphen-minus (-) is given as file name) for lines
       containing  a  match to the given PATTERN.  By default, grep prints the
       matching lines.
```

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

— Regular expression operations

ce code: Lib/re.py

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

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.

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

<u>All</u> **regular languages** can be <u>constructed</u> **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

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 <u>recursive definition</u>

Flashback: Recursive Definitions

Recursive definitions are definitions with a <u>self-reference</u>

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

Flashback: Recursive Definitions

Base case

Recursive case

Self-reference

Recursive call with
“smaller” argument

```
function factorial( n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

Flashback: # Recursive Definitions

A **Natural Number** is either:

Base case
- **Zero**, or

Recursive case
- the **Successor** of a **Natural Number**

Self-reference

"smaller" argument

Flashback: Recursive Definitions

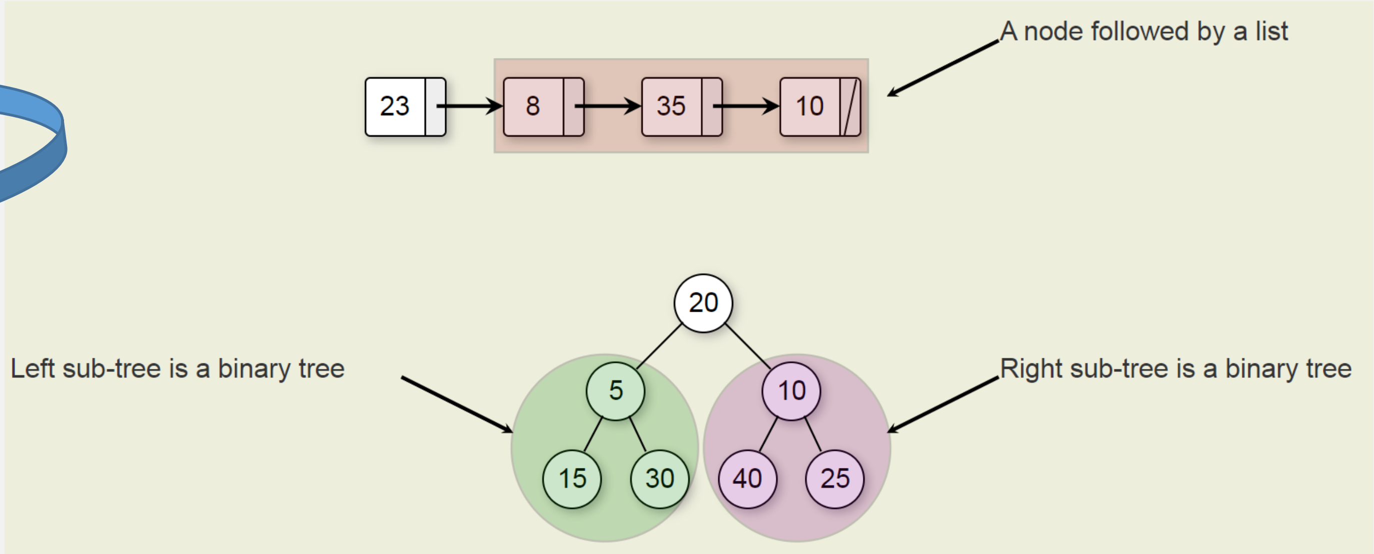
```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

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

3 Base Cases

1. a for some a in the alphabet Σ, (A lang containing a) **length-1 string**
2. ε, (A lang containing) **the empty string** (This is the 3rd **use** of the **ε symbol!**)
3. $\emptyset$, **The empty set** (i.e., a lang containing no strings)

union → 4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

concat → 5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or

star → 6. (R_1^*), where R_1 is a regular expression.

3 Recursive Cases

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

Regular Expression: Concrete Example

<u>Entire</u> **regular expr**: language whose strings come from these languages concat'ed (implicit) together

the language { 0, 1 }

$(0 \cup 1)0^*$

the language { ε, 0, 00, …}

the language { 0 }

the language { 1 }

- Operator <u>Precedence</u>:
  - 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

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

$$\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$$

Σ in regular expression = "any char"

$$1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$$

let R^+ be shorthand for RR^*

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

$0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$

$$1^*\emptyset = \emptyset$$

$A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$

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 = Regular Langs?

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

3 Base Cases

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,

3 Recursive Cases

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.

Prove: Any **regular language** can be constructed from: base cases + union, concat, Kleene star

We would like:
- A **regular expression** represents a **regular language**
- The *set of all* **regular expressions** represents the *set of* **regular languages**

(But we have to prove it)

Thm: A Lang is Regular **iff** Some Reg Expr Describes It

⇒ If **a language is regular,** it **is described by a reg expression**

⇐ If **a language is described by a reg expression,** it **is regular**

(Easier)

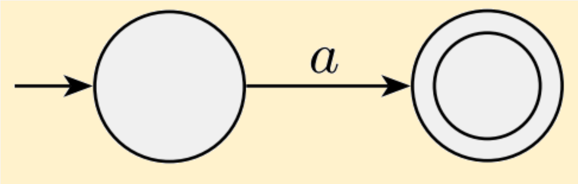
- Key step: **convert reg expr → equivalent NFA!**
- (Hint: we mostly did this already when discussing closed ops)

How to show that a language is regular?

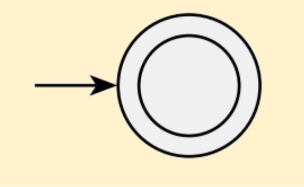
Construct a **DFA** *or* **NFA!**

RegExpr→NFA

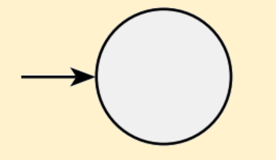
R is a ***regular expression*** if R is



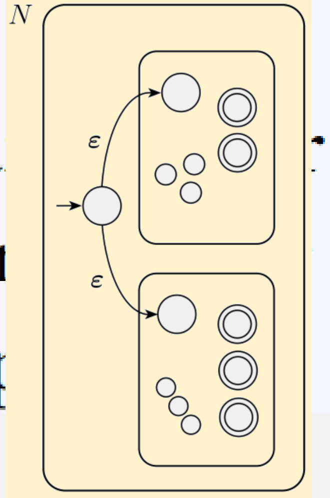
1. a for some a in the alphabet Σ,
2. ε,

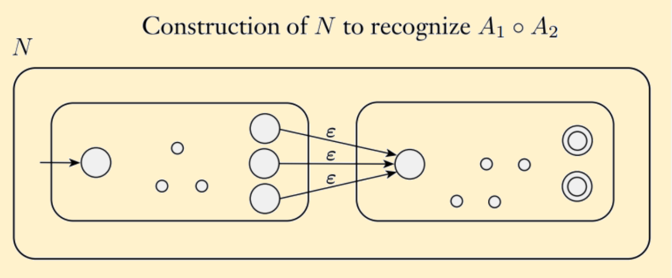


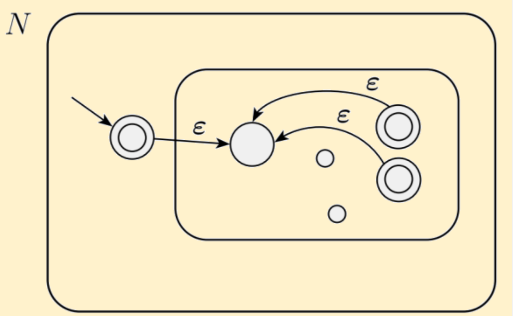
3. $\emptyset$,



4. $(R_1 \cup R_2)$, where R_1 and R_2 a … r e
5. $(R_1 \circ R_2)$, where R_1 and R_2 ar … expressions, or
6. (R_1^*), where R_1 is a regular exp







Thm: A Lang is Regular iff Some Reg Expr Describes It

⇒ If **a language is regular,** it **is described by a reg expression**
(Harder)

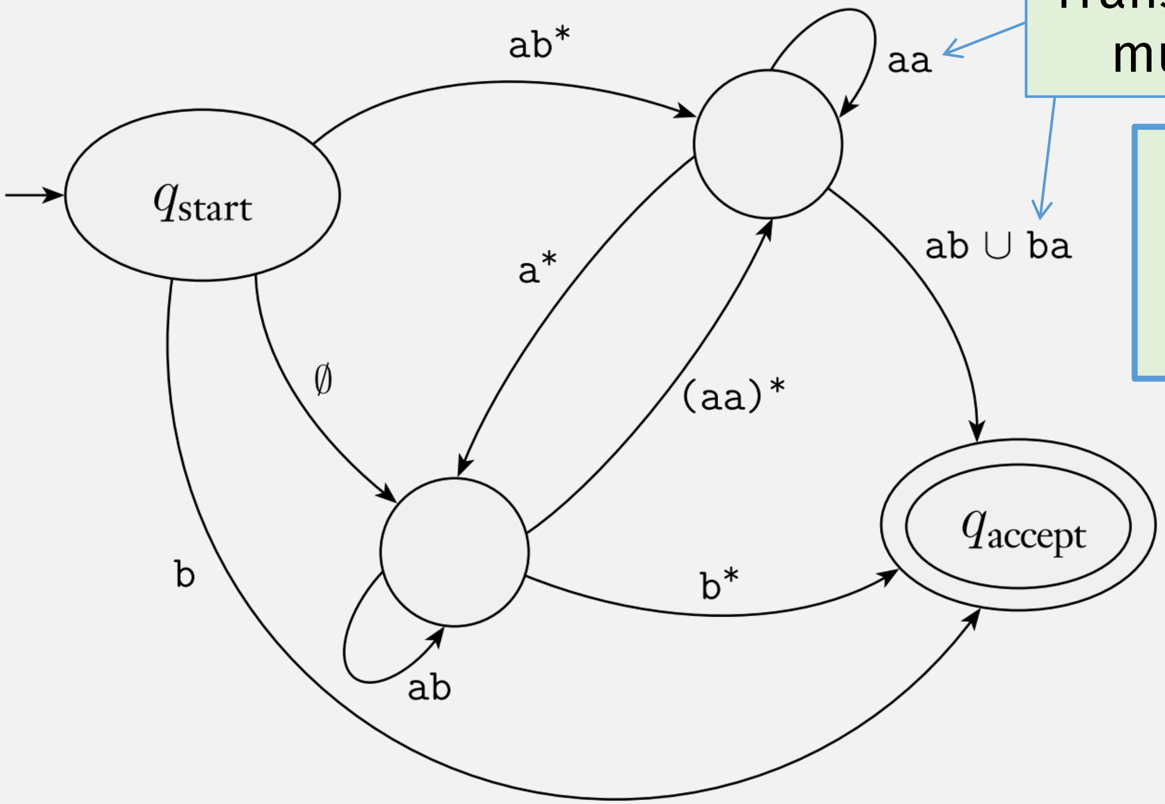
- Key step: Convert an DFA or NFA → equivalent Regular Expression
- To do so, we first need another kind of finite automata: a **GNFA**

⇐ If **a language is described by a reg expression,** it **is regular**
(Easier)

☑ • Key step: Convert the regular expression → an equivalent NFA!

(full proof requires writing Statements and Justifications, and creating an "Equivalence" Table)

Generalized NFAs (GNFAs)



Transition can read multiple chars

A plain NFA
= a GNFA with single char regular expr transitions

Goal: convert **GNFAs** to <u>equivalent</u> **Regular Exprs**

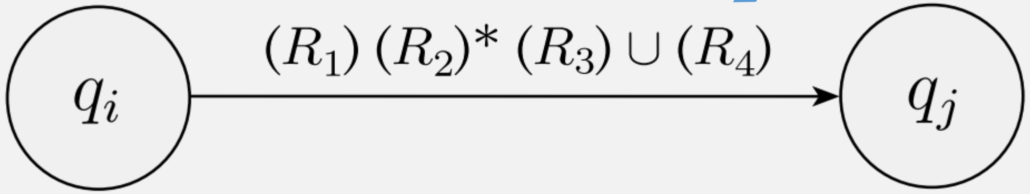
- GNFA = NFA with regular expression transitions

`GNFA→RegExpr` function

On **GNFA** <u>input</u> G:

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

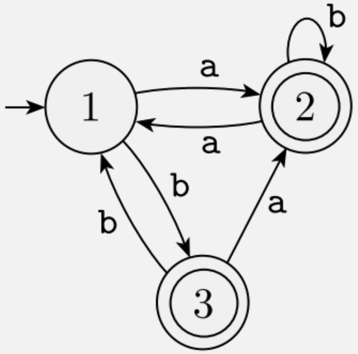
Equivalent regular expression



GNFA

Could there be less than 2 states?

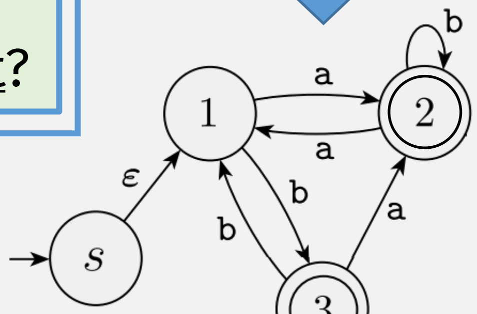
GNFA→RegExpr Preprocessing



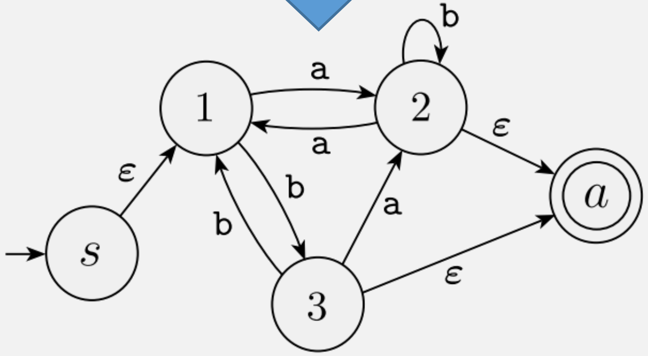
- First, modify input machine to have:

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

  - New start state:
    - No incoming transitions
    - ε transition to old start state



  - New, single accept state:
    - With ε transitions from old accept states



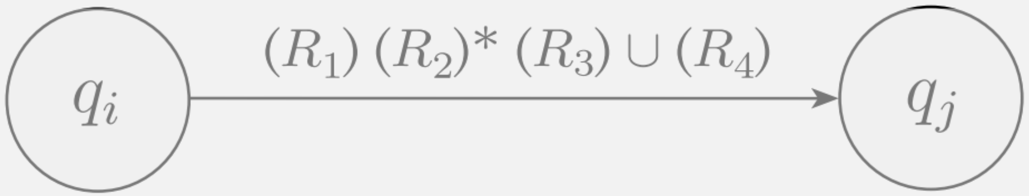
> Modified machine always has 2+ states:
> - <u>New start state</u>
> - <u>New accept state</u>

GNFA→RegExpr function (recursive)

On **GNFA** <u>input</u> G:

Base Case

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

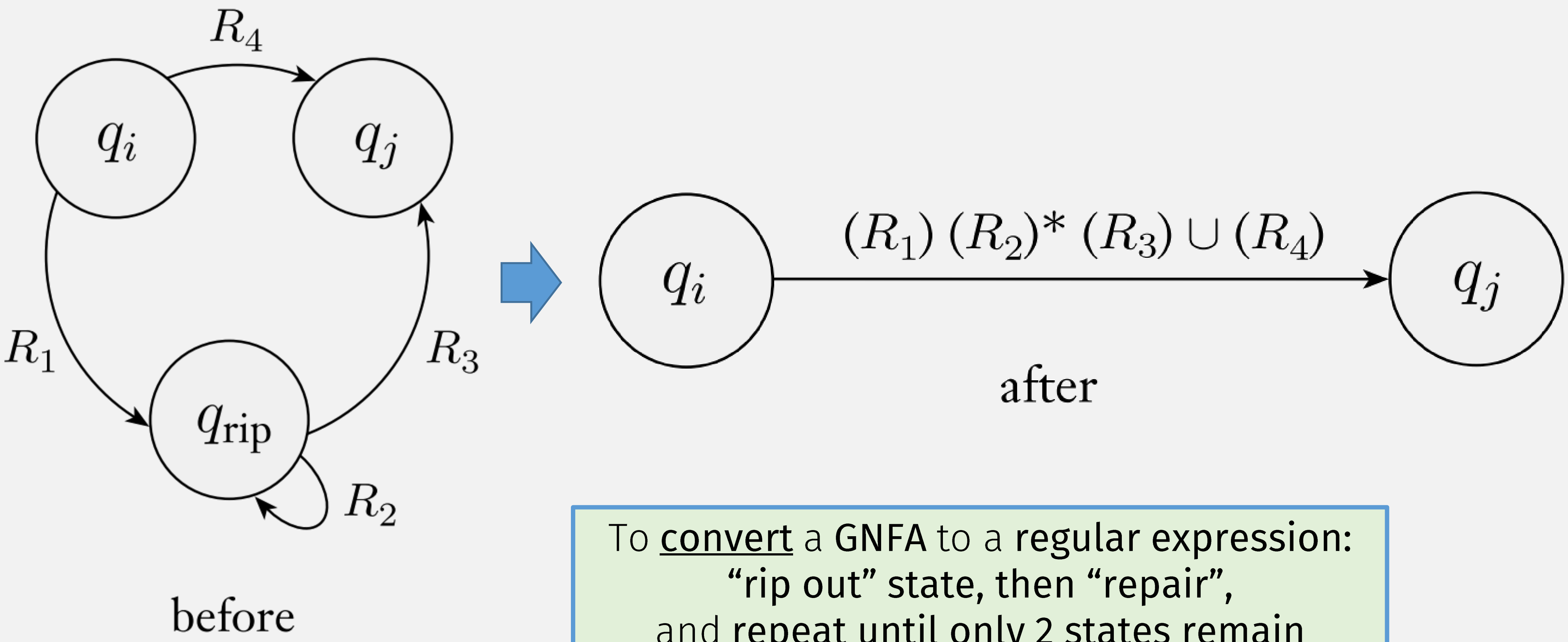


Recursive Case

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

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

GNFA→RegExpr: "Rip/Repair" step

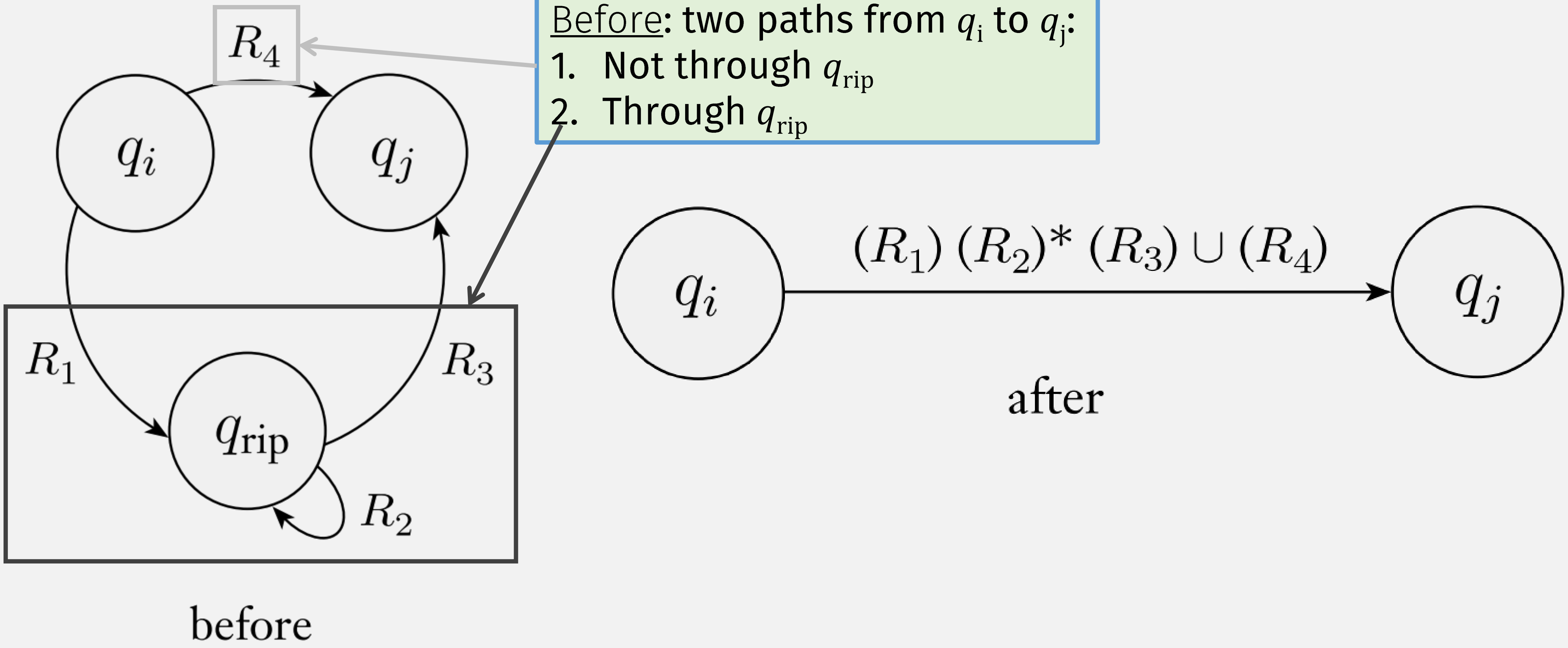


To <u>convert</u> a GNFA to a **regular expression**:
"**rip out**" state, then "**repair**",
and **repeat until only 2 states remain**

GNFA→RegExpr: "Rip/Repair" step

Before: two paths from q_i to q_j:

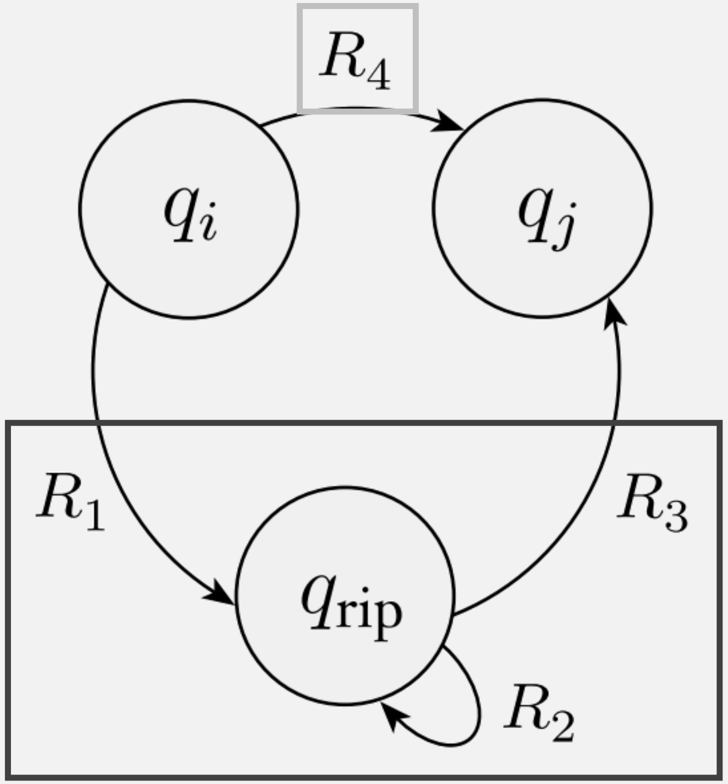
1. Not through q_{rip}
2. Through q_{rip}



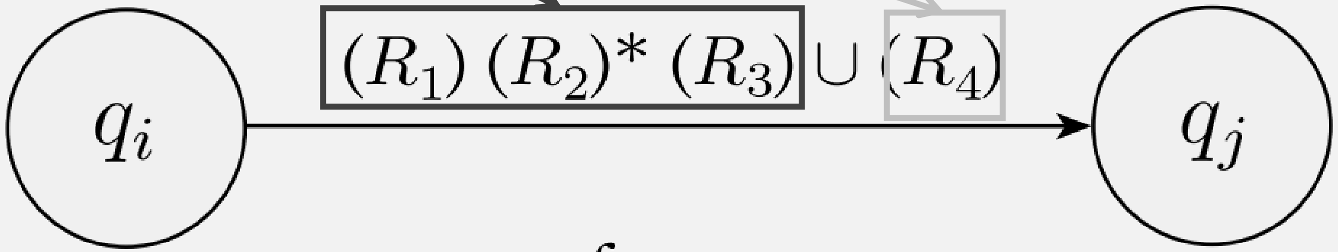
GNFA→RegExpr: “Rip/Repair” step

<u>After</u>: union of two “paths” from q_i to q_j

1. Not through q_rip
2. Through q_rip

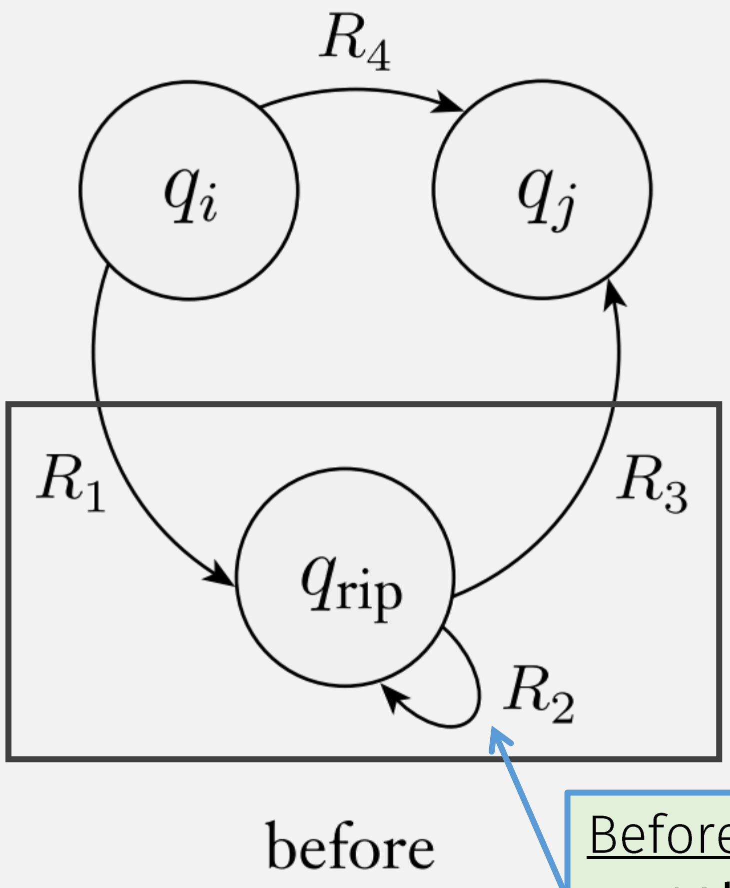


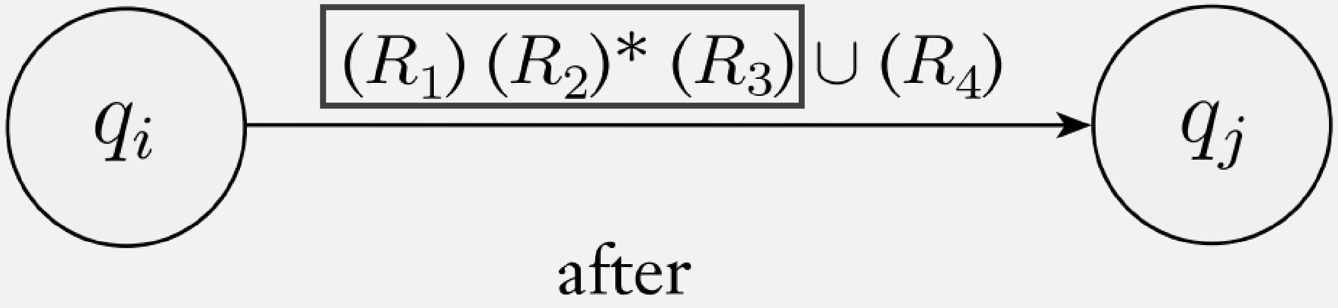
before



after

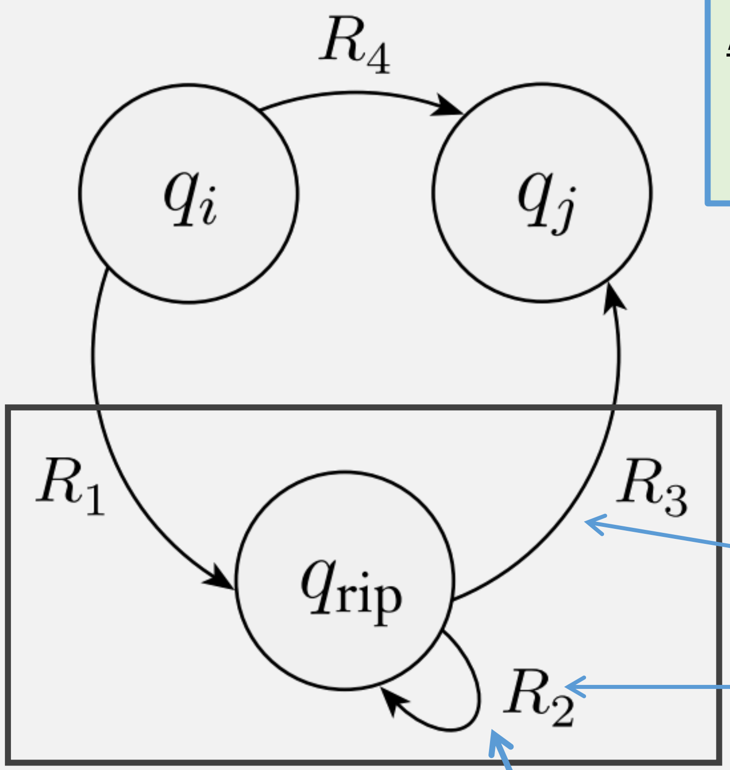
GNFA→RegExpr: "Rip/Repair" step



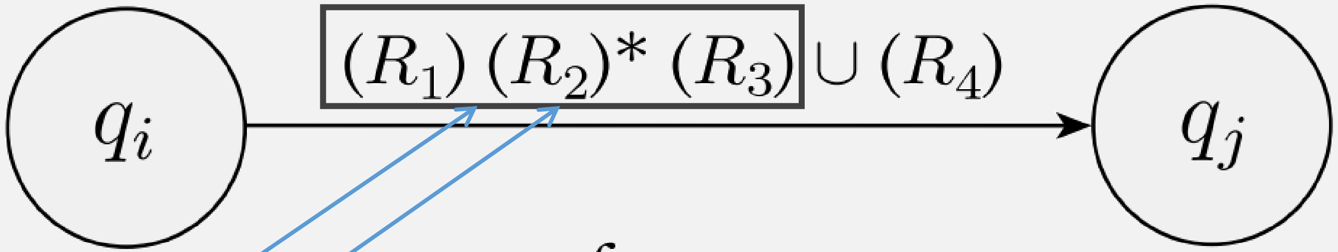


<u>Before</u>:
- path through q_{rip} has 3 transitions
- One is self-loop

GNFA→RegExpr: “Rip/Repair” step



before



after

After:
- Self loop becomes star operation
- Others are concat’ed together

concat

Star operation

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

Thm: A Lang is Regular **iff** Some Reg Expr Describes It

⇒ If **a language is regular**, it **is described by a regular expr**

Need to convert DFA or NFA to Regular Expression …

☑ • Use **GNFA→RegExpr** to convert GNFA → $\boxed{\text{equiv}}$ regular expression!

???

⇐ If a language is described by a regular expr, it is regular

☑ • Convert regular expression → equiv NFA!

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

`GNFA→RegExpr` Correctness

- Correct / Equivalent means:

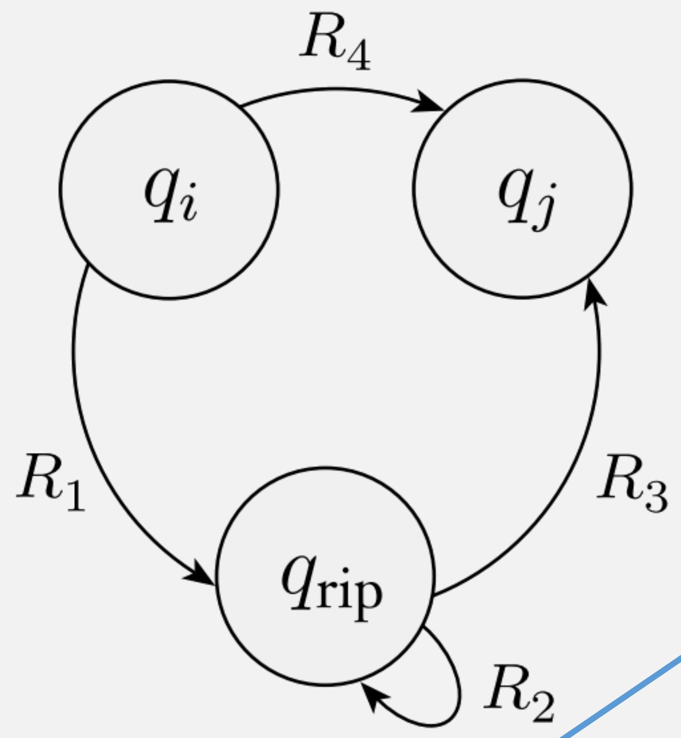
$$\textsc{LangOf}\ (\ G\ ) = \textsc{LangOf}\ (\ R\ )$$

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

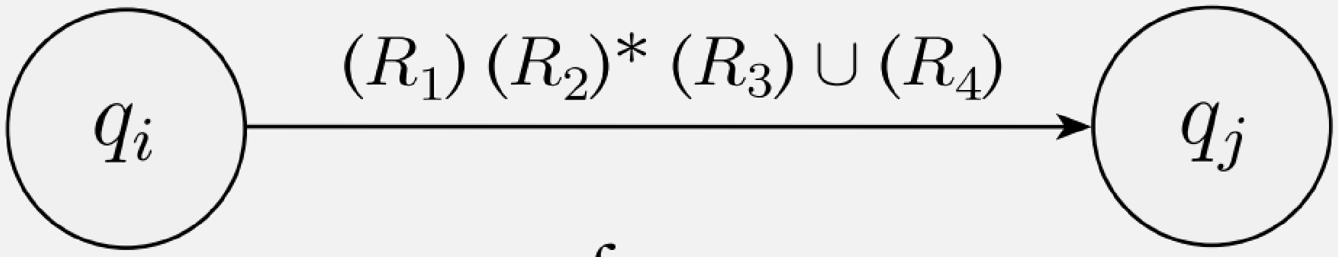
This time, let's really prove equivalence!
(previously, we "proved" it)

- i.e., **`GNFA→RegExpr`** must not change the language!
    - Key step: the rip/repair step

GNFA→RegExpr: Rip/Repair Correctness



before



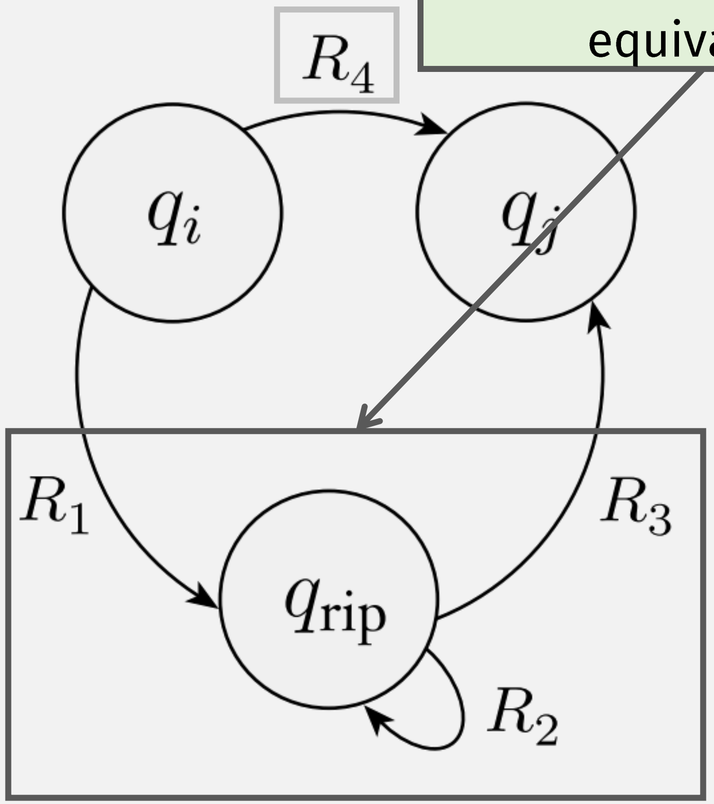
after

Must show these
are <u>equivalent</u>

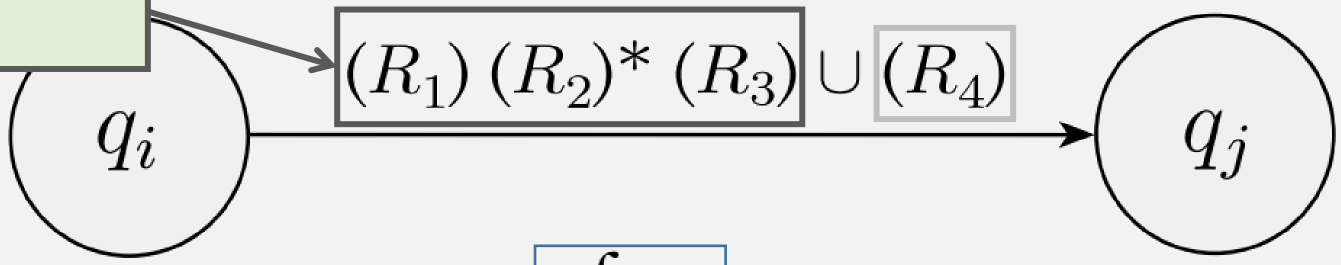
Equivalent =
same language =
accepts the same strings

GNFA→RegExpr: Rip/Repair Correctness

Must show these are equivalent



before



after

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

GNFA→RegExpr "Correctness"

- "Correct" / "Equivalent" means:

How to really prove this part?

$$\text{LangOf}\,(\,G\,) = \text{LangOf}\,(\,R\,)$$

???

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

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

- i.e., **GNFA→RegExpr** must not change the language!
  - Key step: **the rip/repair step** ☑

Next Time

Inductive Proofs