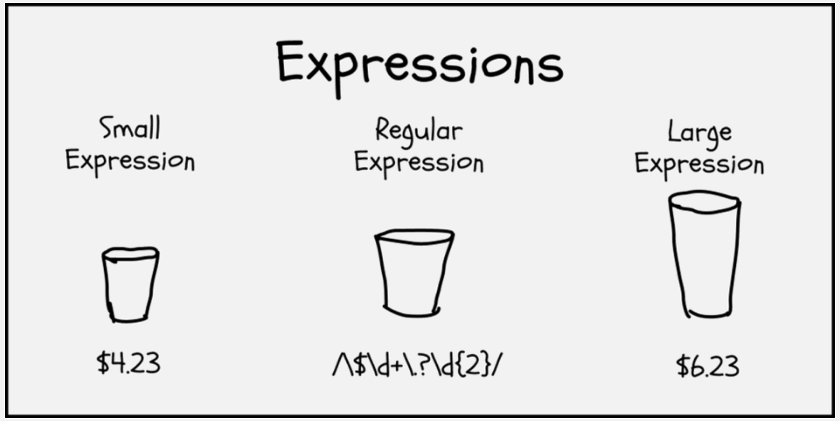
UMB CS 420

Regular Expressions

Tuesday, October 4, 2022



Announcements

- HW 2 in
  - ~~Due Sun 10/2 11:59pm EST~~
- HW 3 out
  - Due Sun 10/9 11:59pm EST
- Sean's office hours
  - Mon 4-5pm EST (McCormack 3rd floor room 139)
- HW 1 issues – many submitted solutions do not answer the question
  - Example Question: "Prove that language L is regular"
  - Example Good Answer: "Language L is regular because …"
  - Example Bad Answer: "Here are some sets of stuff, called $Q, \Sigma, \ldots$"

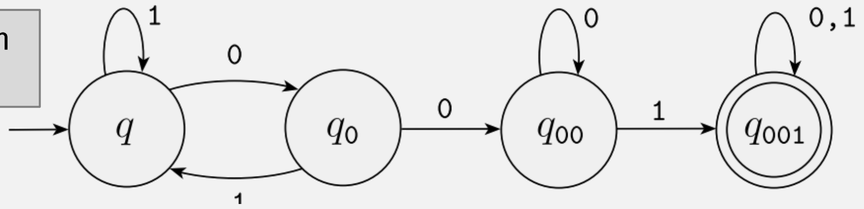
Last Time: Why These (Closed) Operations?

- Union
- Concat
- Kleene star

All regular languages can be constructed from:
- single-char strings, and
- these operations!

So Far: Regular Language Representations

1. State diagram (NFA/DFA)



2. Formal description

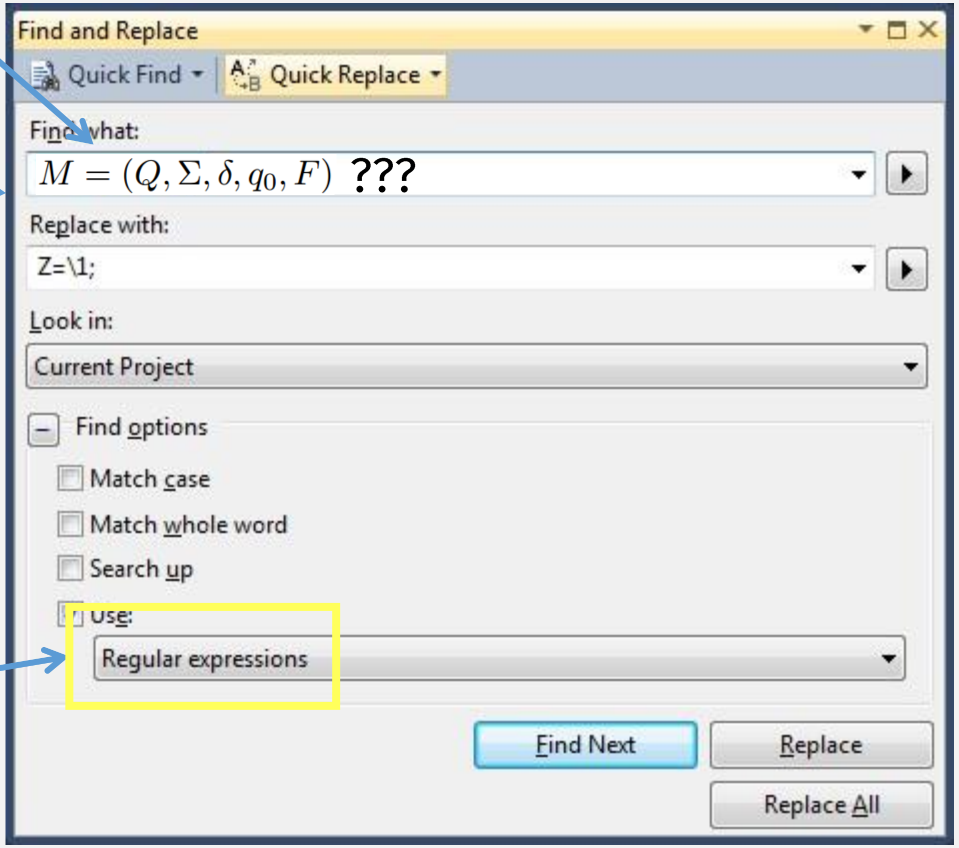
1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	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)

Actually, it's a real programming language: for **text search**



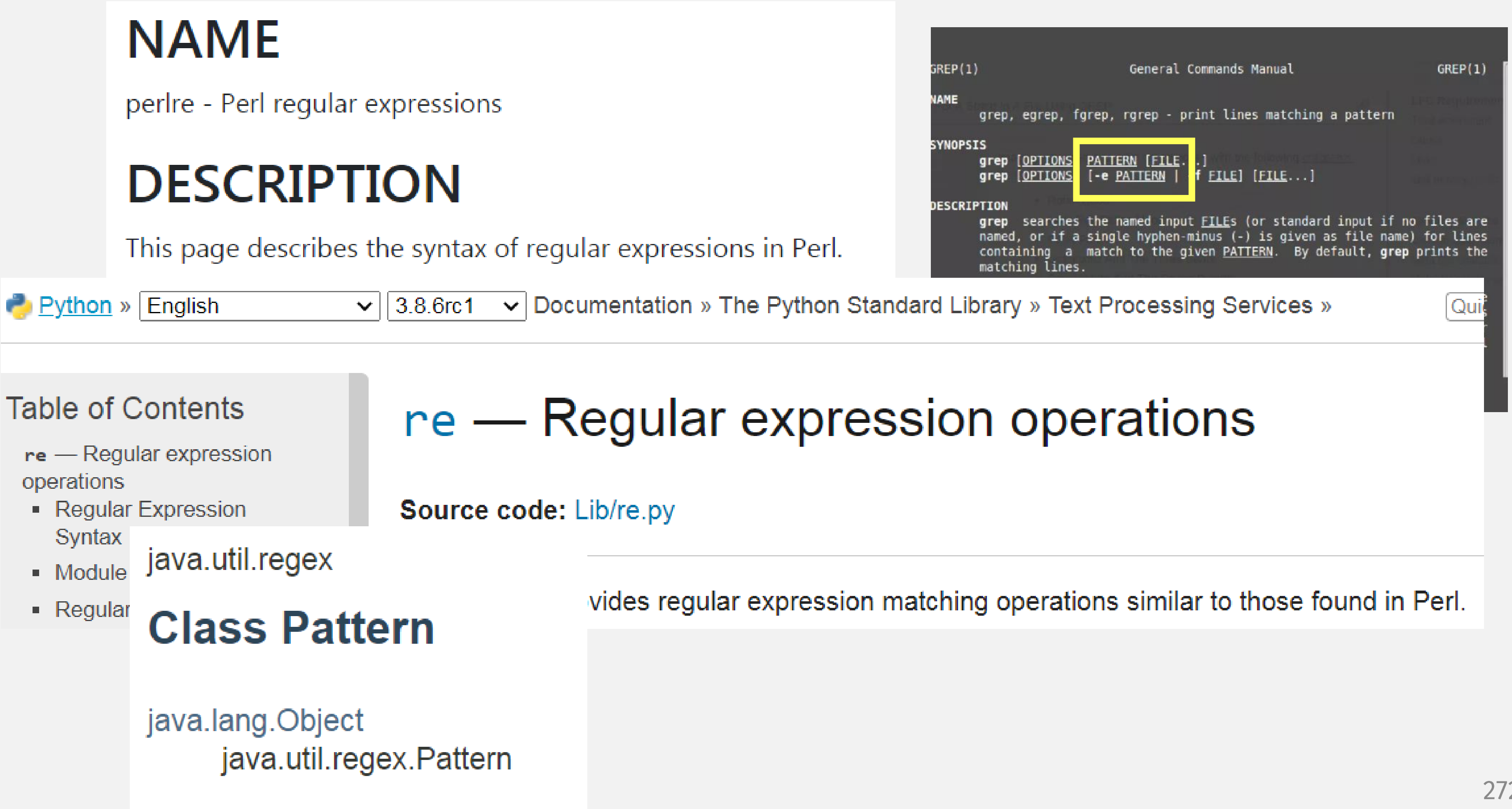
<u>Our Running Analogy:</u>
- <u>Class</u> of **regular languages** ~ a "programming language"
- <u>One</u> **regular language** ~ a "program"

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

Need a more concise (textual) notation??

Regular Expressions: A Widely Used Programming Language
(inside other programming languages)

- Unix
- Perl
- Python
- Java



Why These (Closed) Operations?

- Union
- Concat
- Kleene star

All regular languages can be constructed from:
- single-char strings, and
- these operations!

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

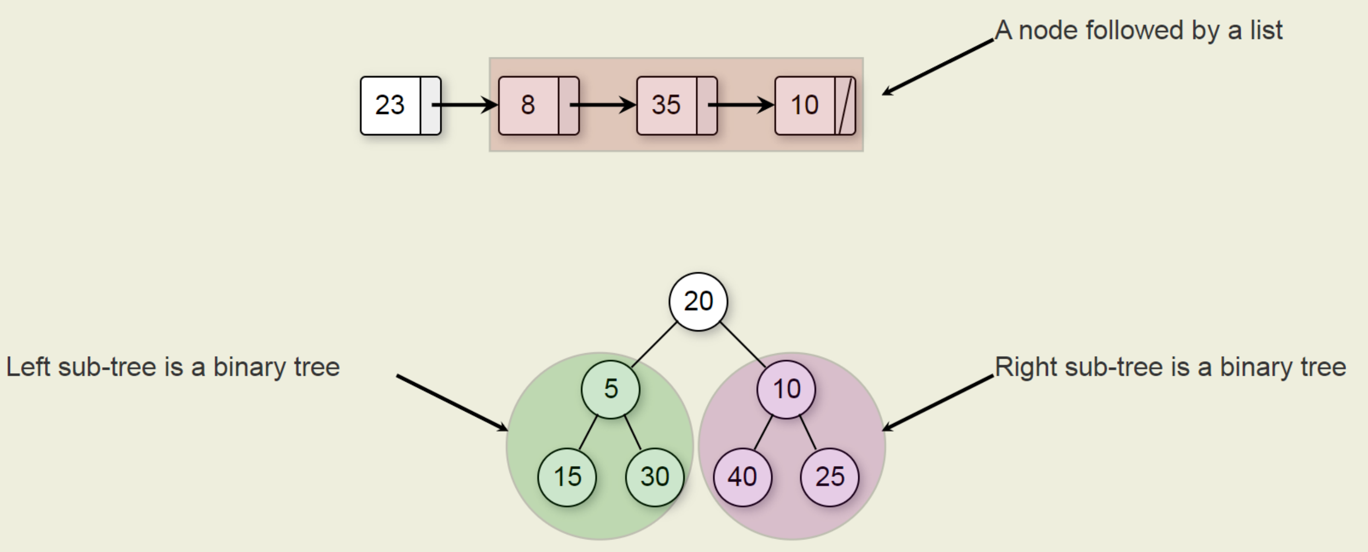
Regular Expressions: Formal Definition

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>

Recursive Definitions



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

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

This is a <u>recursive definition:</u>
Node used before it's defined
(but must be "smaller")

Regular Expressions: Formal Definition

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

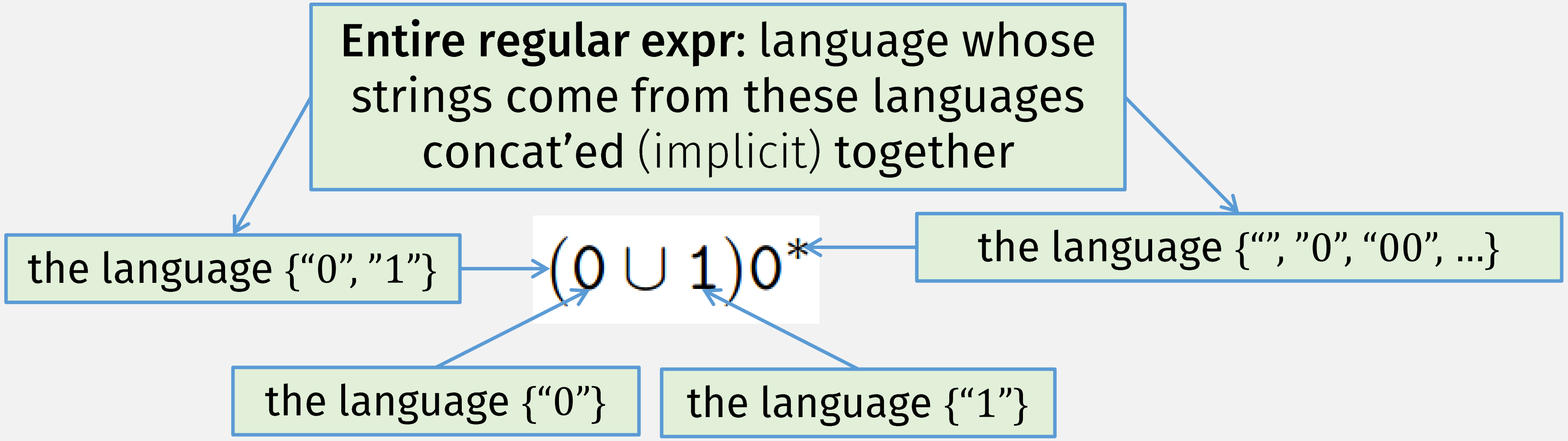
3 Base Cases

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

3 Recursive Cases

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions, ← union
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or ← concat
6. (R_1^*), where R_1 is a regular expression. ← star

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.

Regular Expressions = Regular Langs?

3 Base Cases

3 Recursive Cases

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.

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

(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)

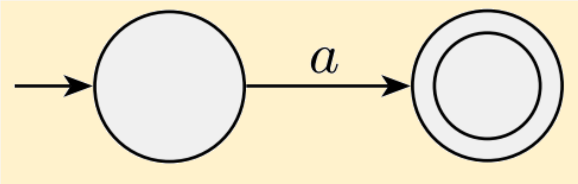
- To prove this part: **convert reg expr → equivalent NFA!**
- (Hint: we mostly did this already when discussing closed ops)

How to show that a language is regular?

Construct a DFA *or* NFA!

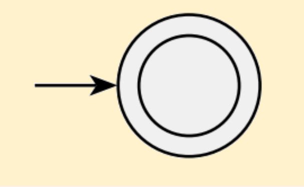
RegExpr→NFA

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

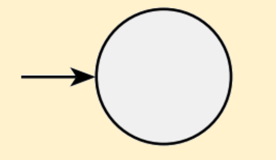


1. a for some a in the alphabet Σ,

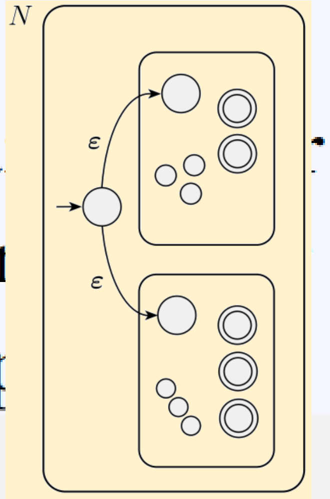
2. ε,



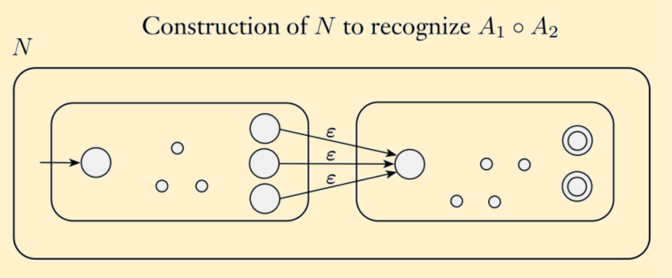
3. $\emptyset$,



4. $(R_1 \cup R_2)$, where R_1 and R_2 a

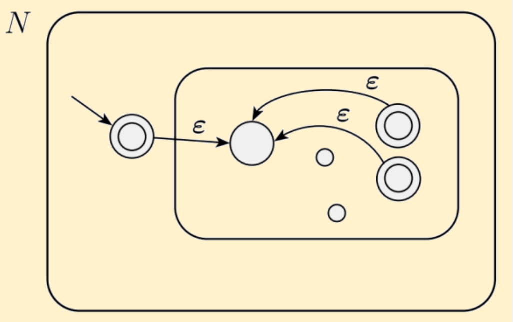


r e



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

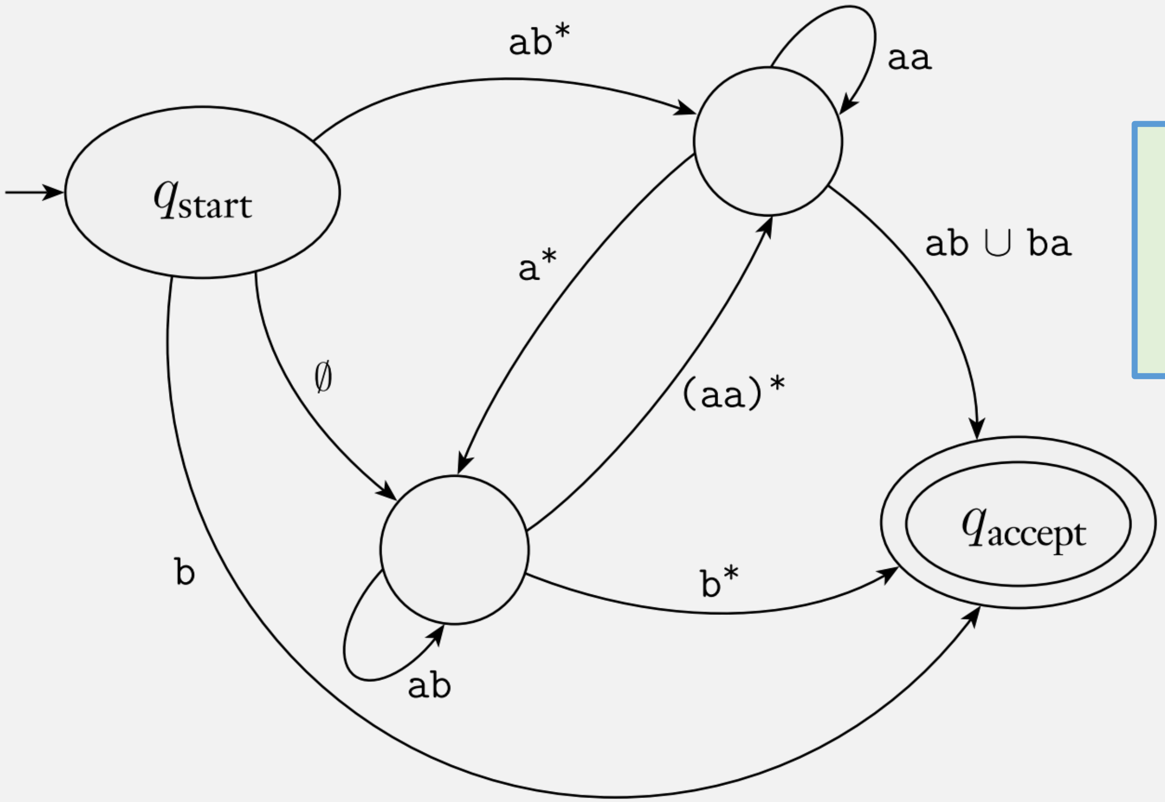
⇒ If **a language is regular,** it **is described by a reg expression**
(Harder)

- To prove this part: 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)

☑ • Convert the regular expression → an equivalent NFA!

Generalized NFAs (GNFAs)



plain NFA
= GNFA with single char
regular expr transitions

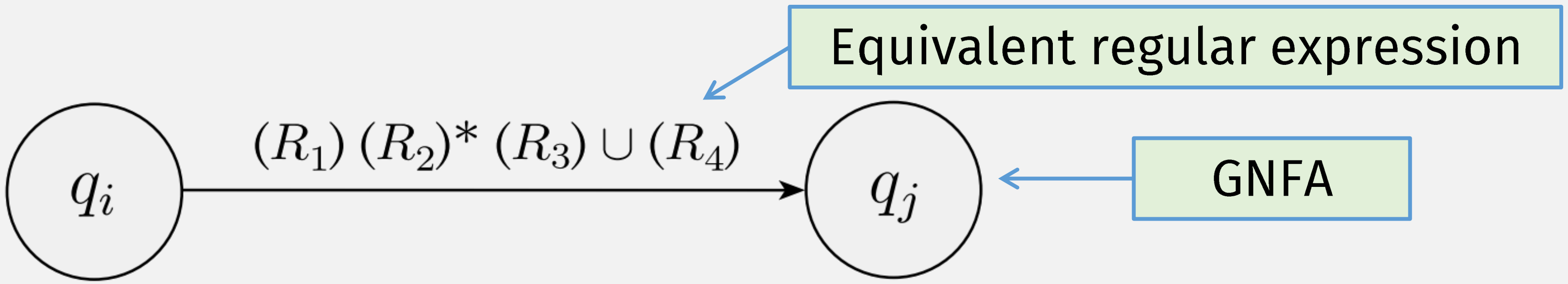
Goal: **convert GNFAs to Regular Exprs**

- 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 transition), e.g.:

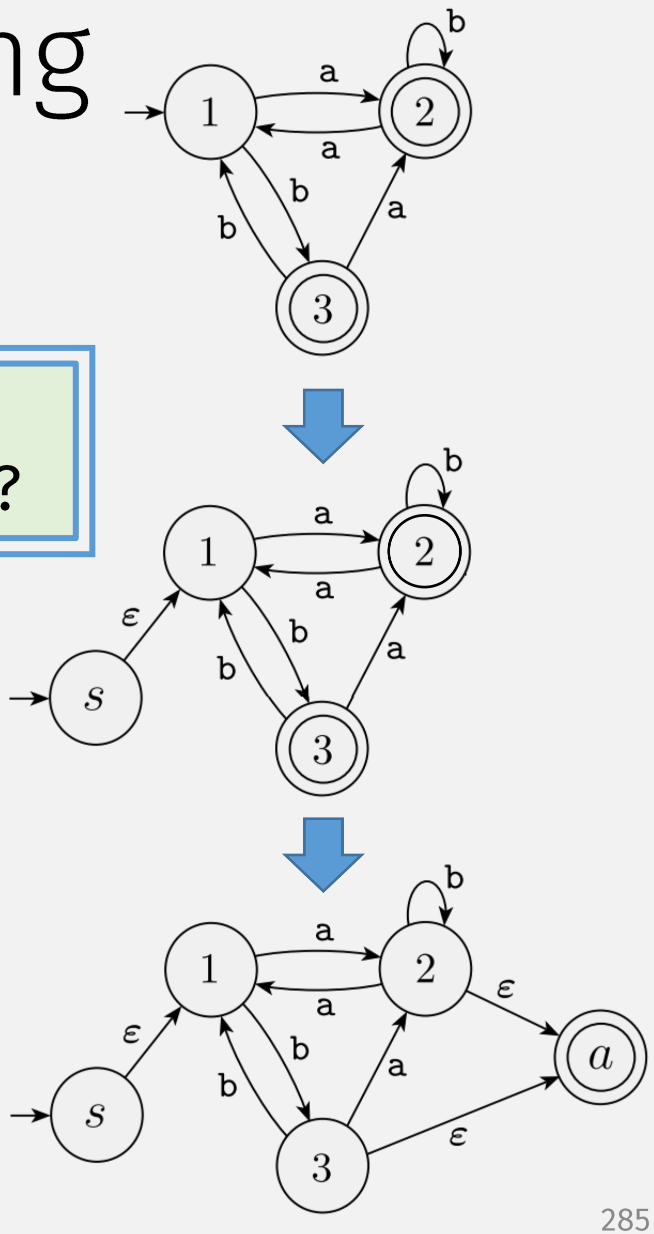


Could there be less than 2 states?

GNFA→RegExpr Preprocessing

- First, modify input machine to have:
  - New start state:
    - No incoming transitions
    - ε transition to old start state
  - New, single accept state:
    - With ε transitions from old accept states

Does this change the language of the machine?

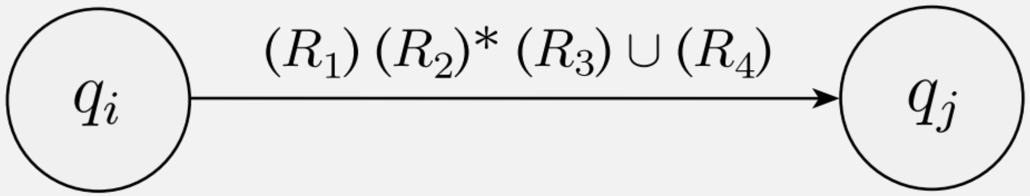


GNFA→RegExpr function (recursive)

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

Base Case

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

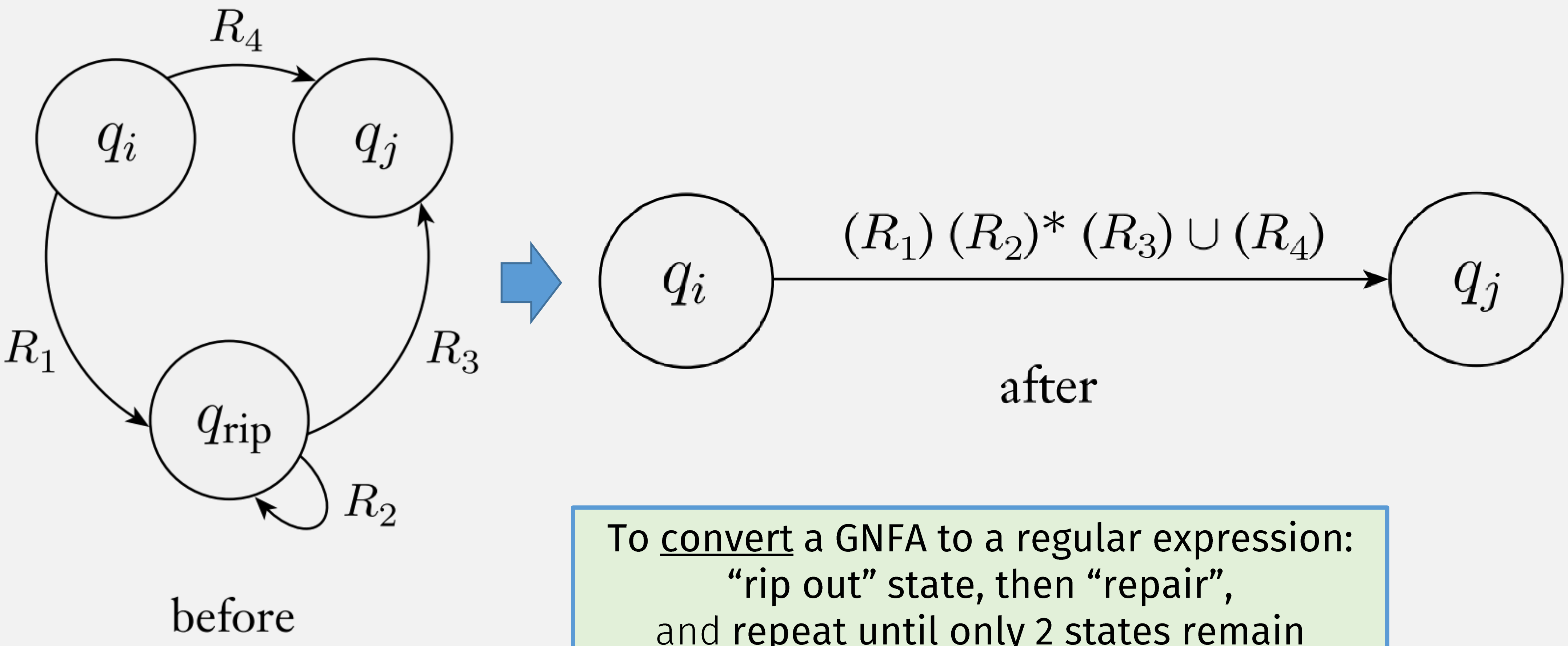


Recursive Case

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

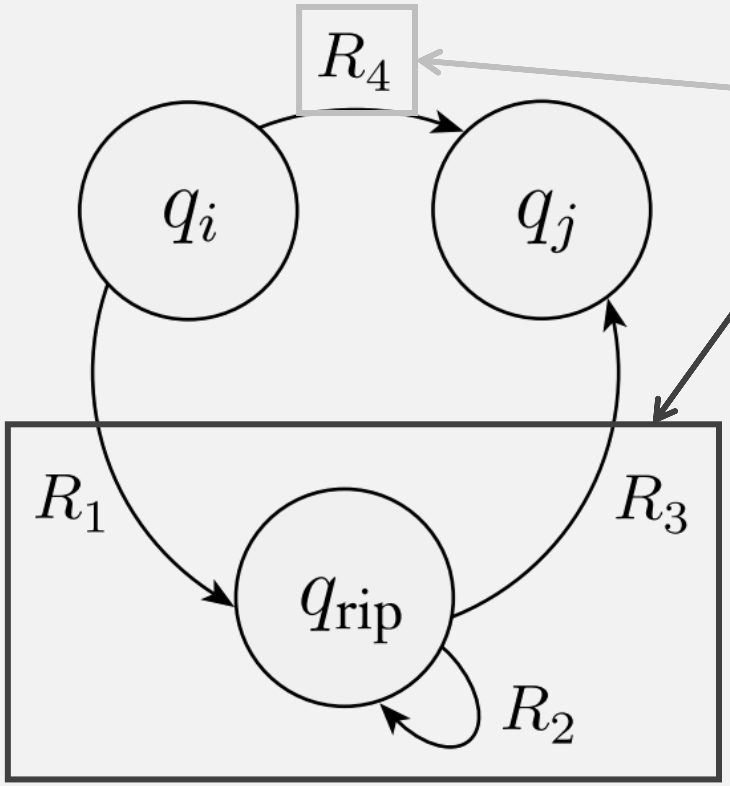
Recursive definitions have:
- <u>base case</u> and
- <u>recursive case</u>
  (with a "smaller" object)

`GNFA→RegExpr`: “Rip/Repair” step



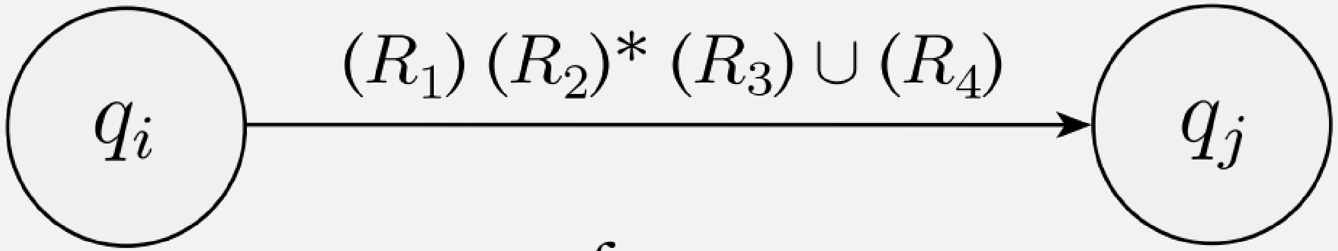
To convert a GNFA to a regular expression:
“rip out” state, then “repair”,
and repeat until only 2 states remain

GNFA→RegExpr: "Rip/Repair" step



before

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

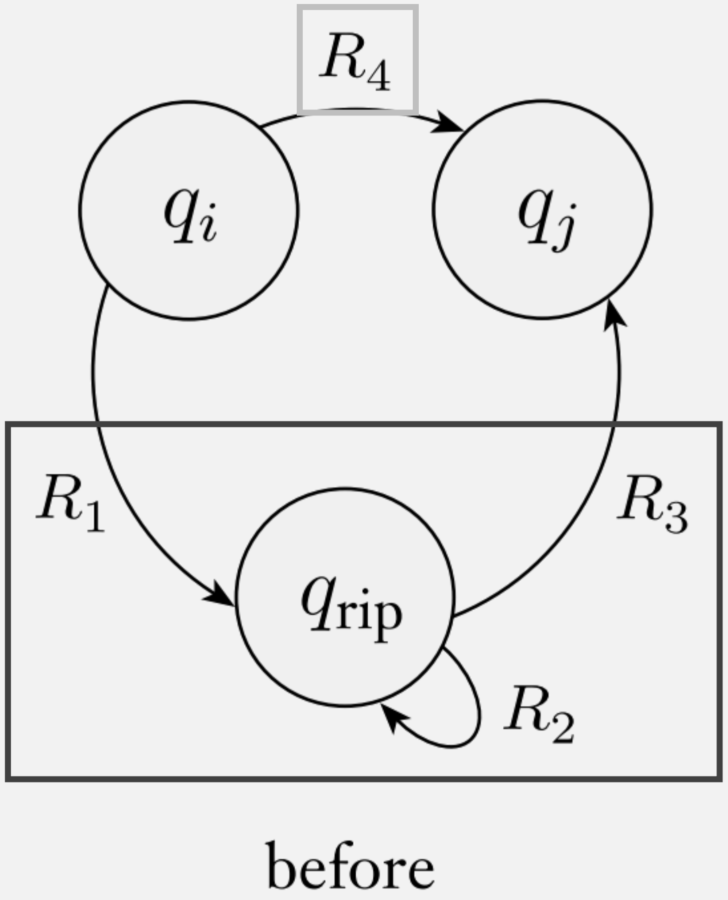


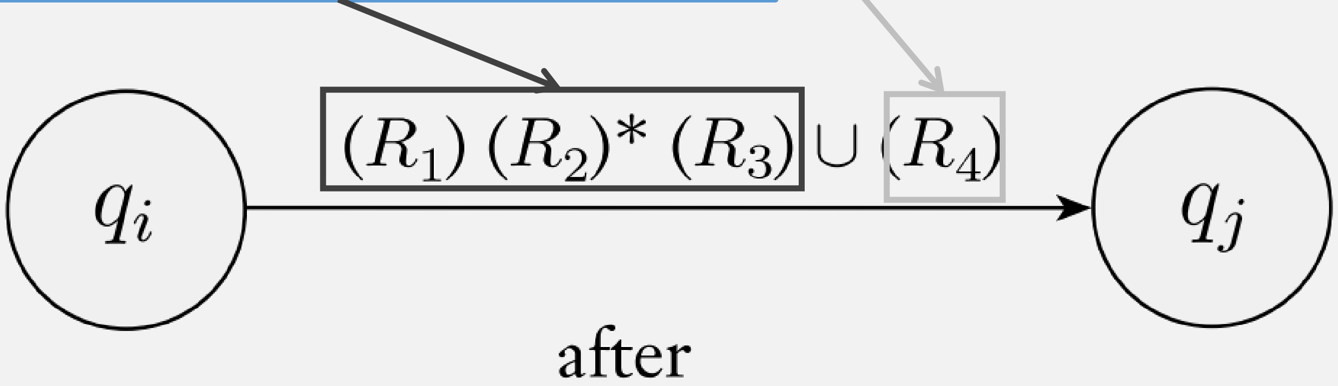
after

GNFA→RegExpr: "Rip/Repair" step

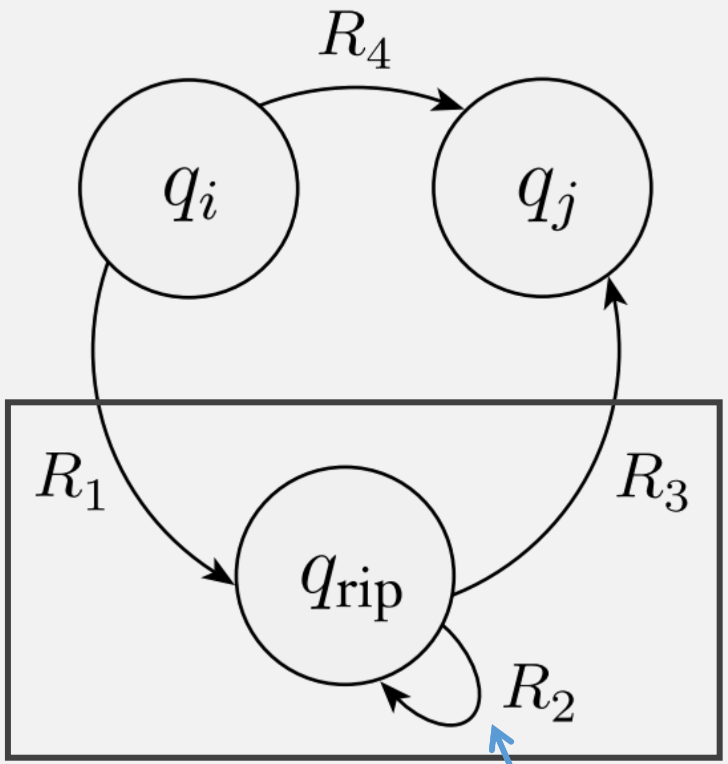
After: still two "paths" from q_i to q_j

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

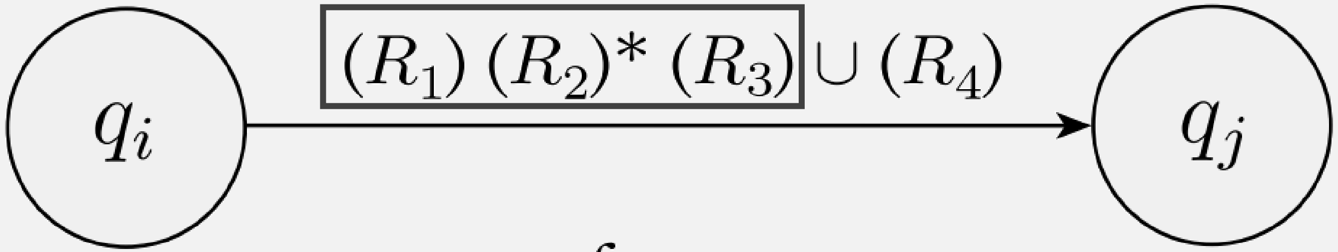




`GNFA→RegExpr`: “Rip/Repair” step



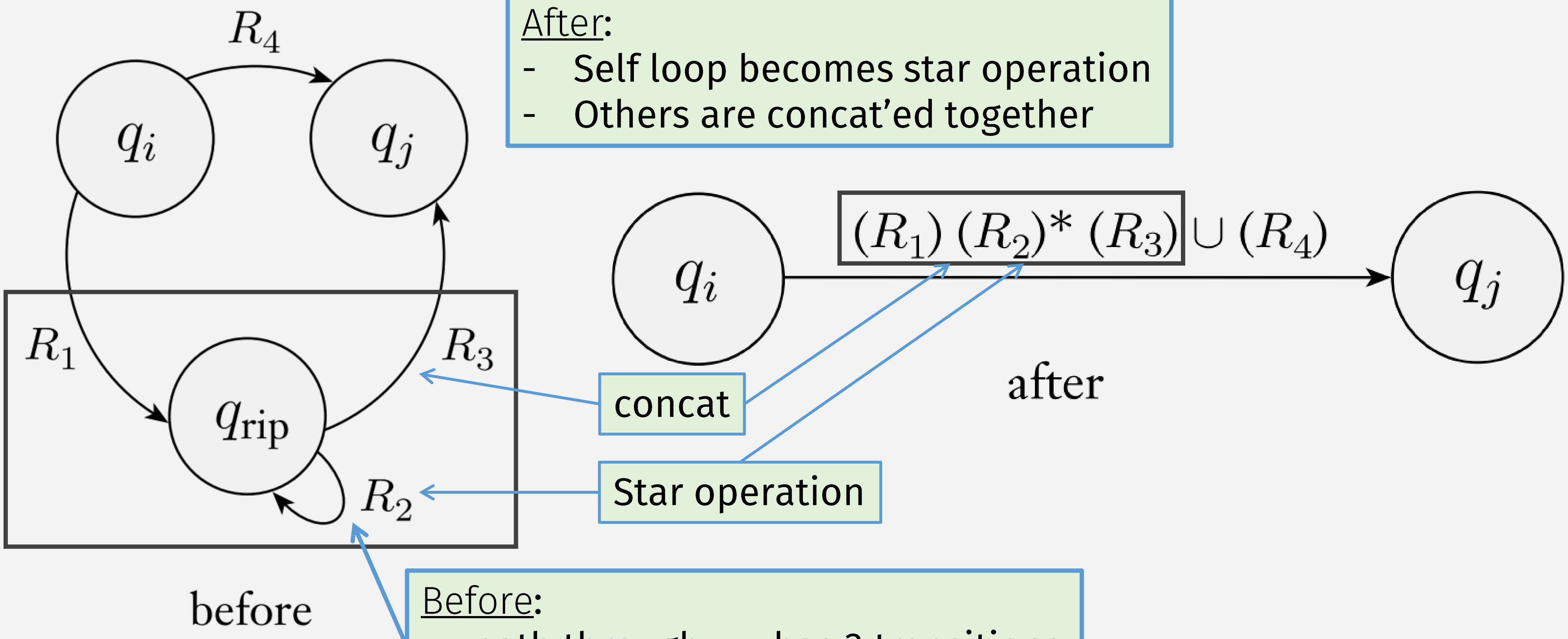
before



after

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

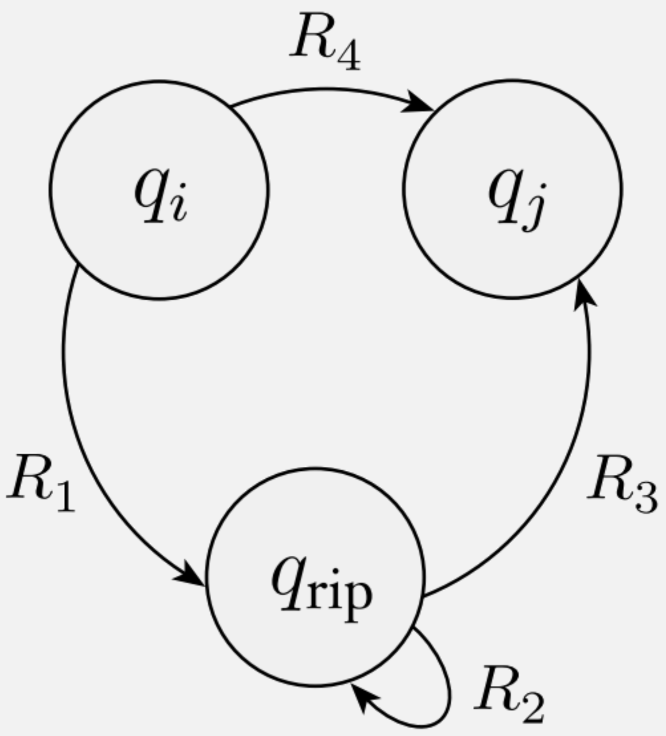
GNFA→RegExpr: "Rip/Repair" step



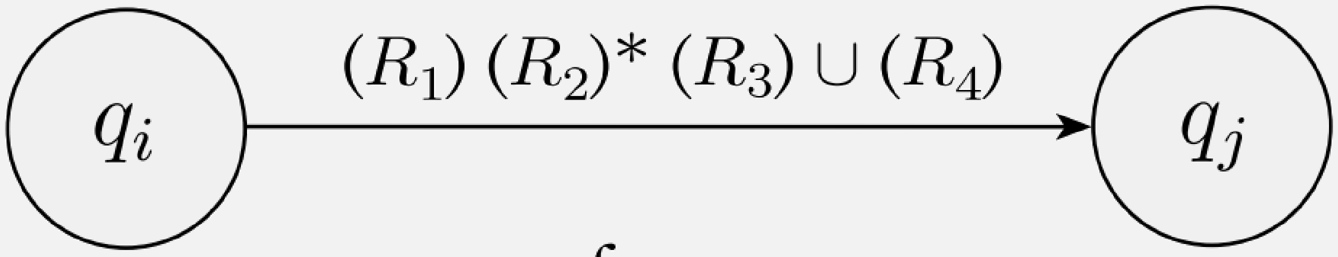
After:
- Self loop becomes star operation
- Others are concat'ed together

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

GNFA→RegExpr: Rip/Repair "Correctness"



before



after

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

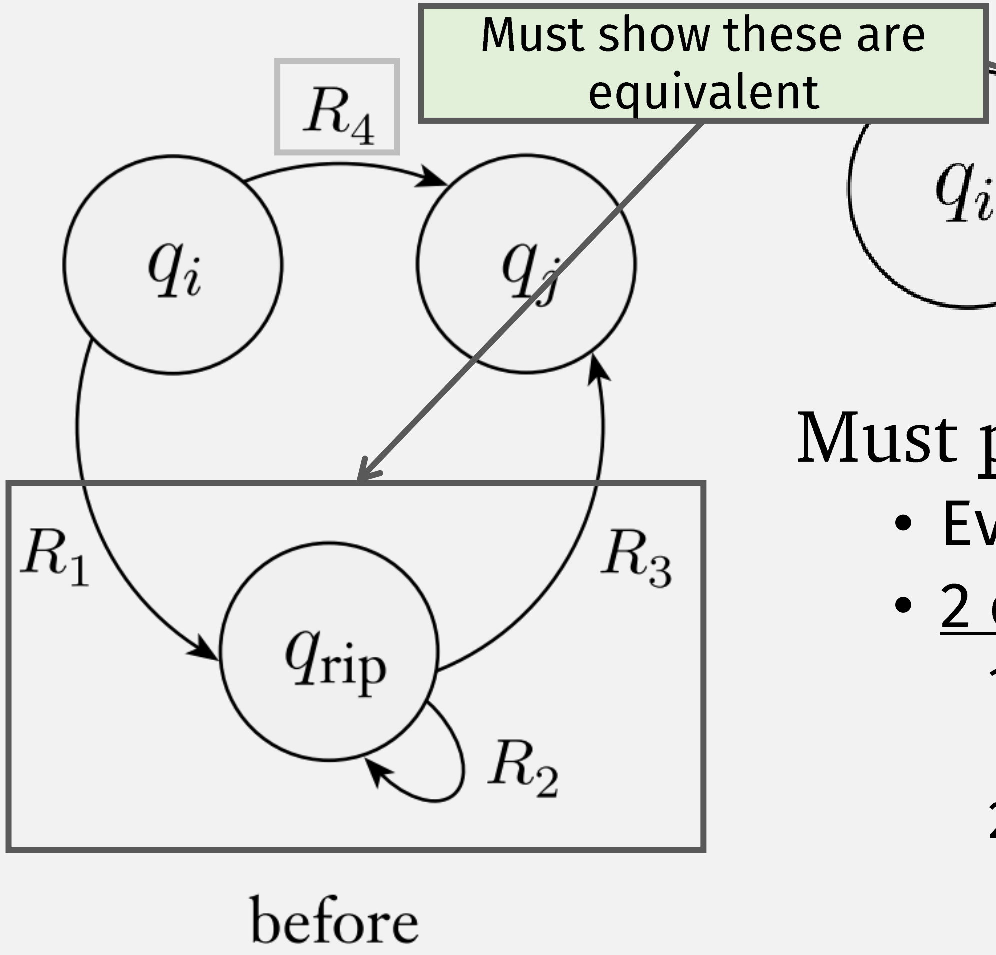
GNFA→RegExpr “Correctness”

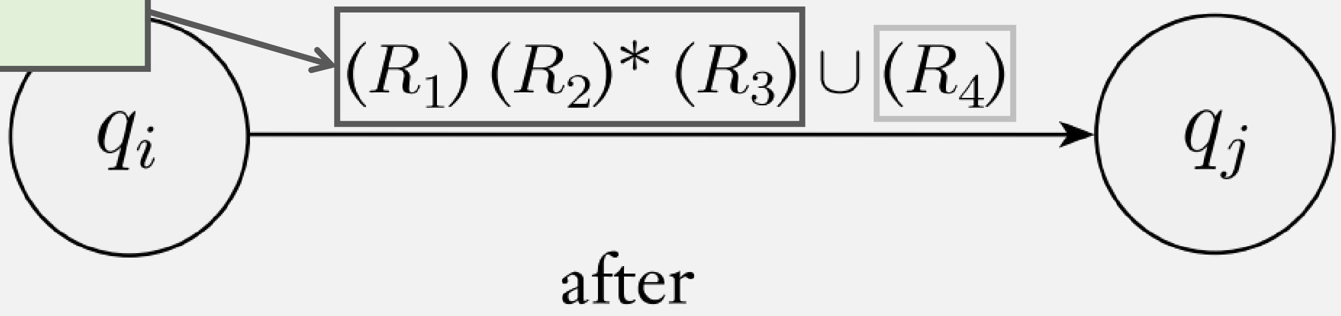
- “Correct” / “Equivalent” means:

$$\text{LangOf}\,(\,G\,) = \text{LangOf}\,(\,\textbf{GNFA→RegExpr}(\,G\,)\,)$$

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

GNFA→RegExpr: Rip/Repair “Correctness”





Must <u>prove</u>:

- Every string accepted <u>before</u>, is accepted <u>after</u>
- <u>2 cases:</u>
  1. Accepted string does not go through q_{rip}
     - ☑ Acceptance unchanged (both use R_4 transition part)
  2. String goes through q_{rip}
     - Acceptance unchanged?
     - ☑ Yes, via our previous reasoning

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 → equiv regular expression!

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

☑ • Convert regular expression → equiv NFA!

■

Now we may use regular expressions to represent regular langs.

So a regular language has these equivalent representations:

- DFA
- NFA
- Regular Expression

So we also have another way to prove things about regular languages!

How to Prove A Language Is Regular?

- Construct DFA
- Construct NFA
- Create Regular Expression ← Slightly different because of recursive 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.

Kinds of Mathematical Proof

- **Deductive proof** (from before)
  - Starting from assumptions and known definitions,
  - Reach conclusion by making logical inferences

- **Inductive proof** (now)
  - …
  - Use this when working with <u>recursive</u> definitions

In-Class quiz 10/4

See gradescope