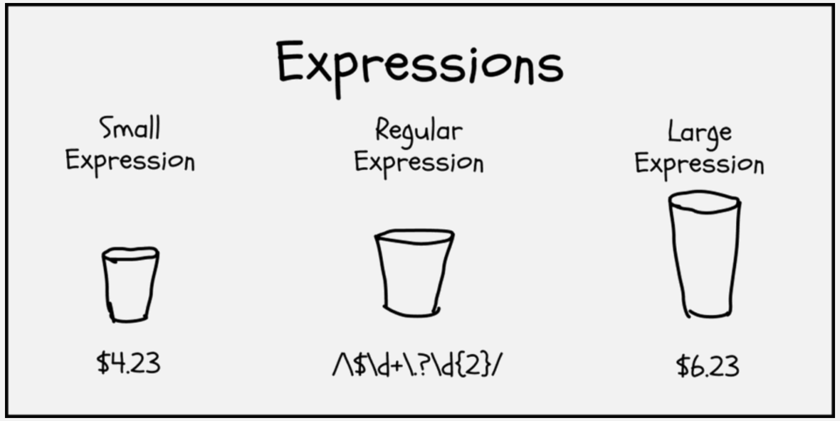
UMB CS 420

Regular Expressions

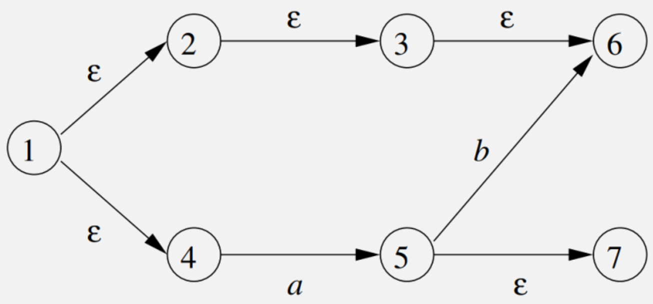
Wednesday February 22, 2023



Announcements

- HW 3 out
  - Due Sun 2/26 11:59pm EST

- <u>Quiz Preview</u>:
  1. What is the alphabet of the following NFA?
     - You may assume that empty transitions are denoted with ε



  2. Which of the following is a representation of a regular language?

Last Time: Why These (Closed) Operations?

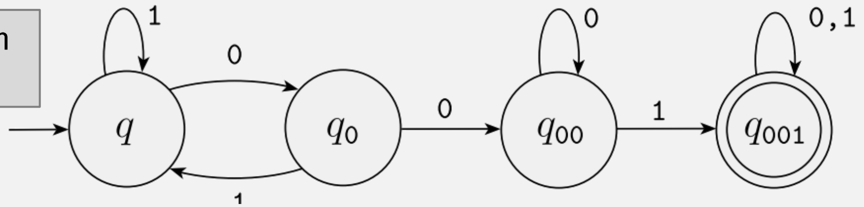
- Union
- Concat
- Kleene star

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

So Far: Regular Language Representations

State diagram (NFA/DFA)

1.



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

Formal description

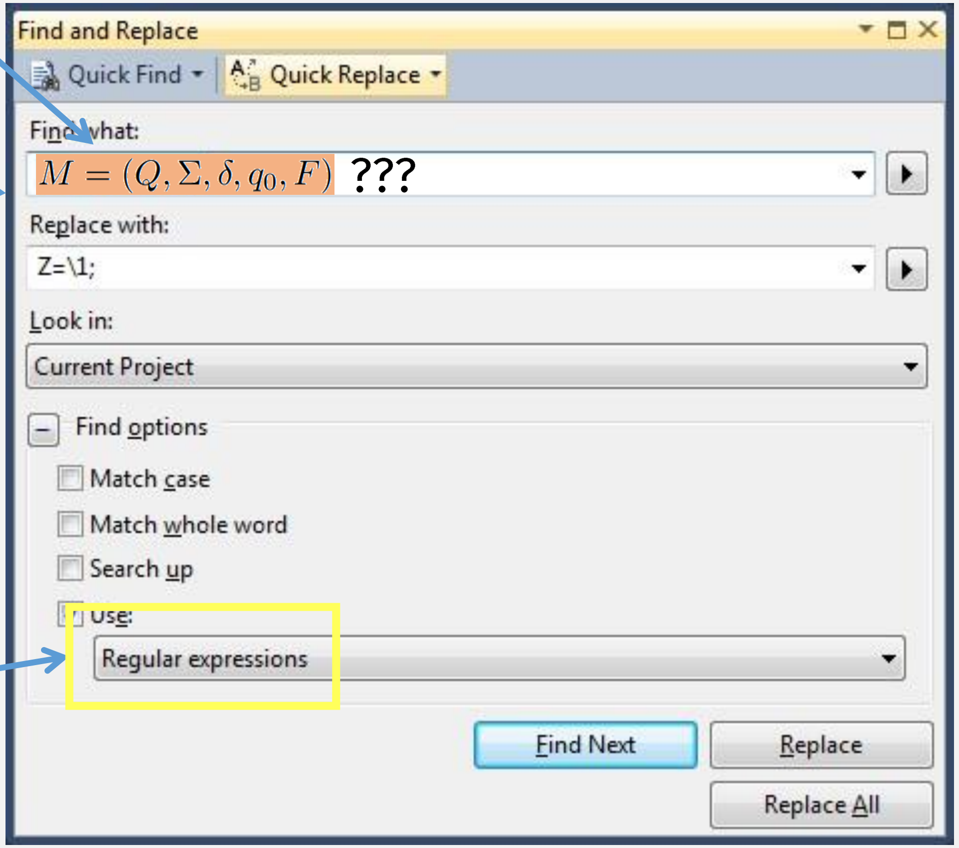
2.

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)



Our Running Analogy:
- Class of **regular languages** ~ a "programming language"
- One **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 »

— 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
- Concat
- Kleene star

All regular languages can be constructed from:
- single-char strings, and
- these three closed 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 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

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

- **Base case** → `if ( n == 0 )`
- **Recursive case** → `else`
- Self-reference → `factorial`
- Recursive call with "smaller" argument → `( n - 1 )`

Flashback: Recursive Definitions

A **Natural Number** is either:

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

Self-reference

Base case

Recursive case

"smaller" argument

Flashback: Recursive Definitions

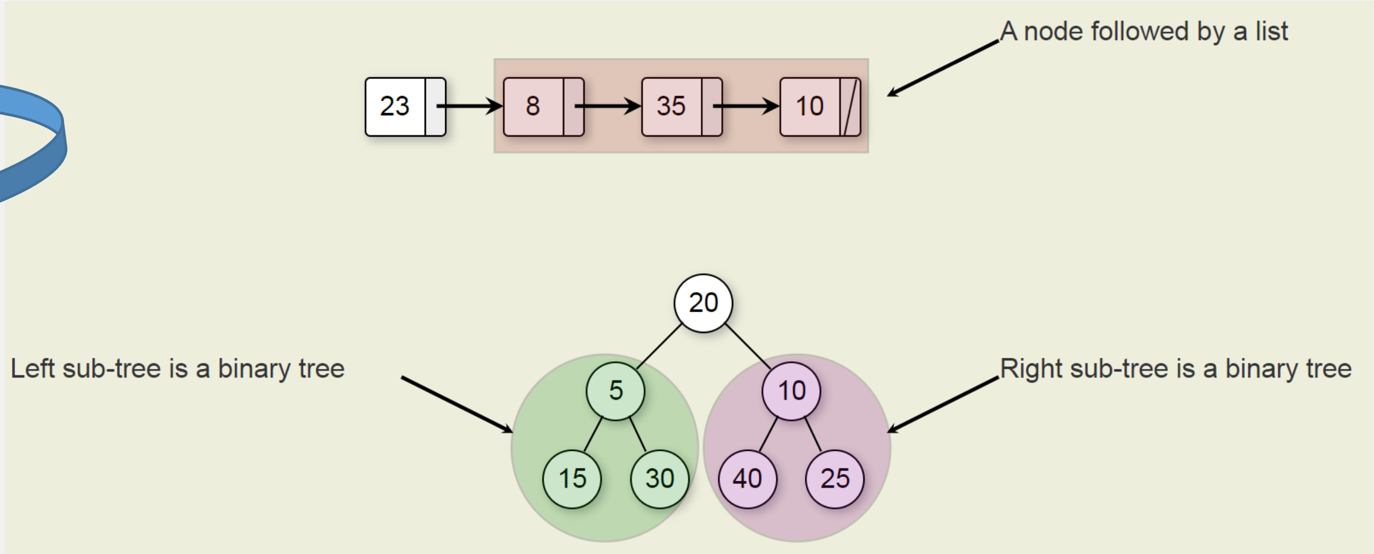
```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

Smaller self-reference

Where's the base case??

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

— *Tony Hoare* —



<u>Data structures</u> are commonly defined <u>recursively</u>

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

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 {"", "0", "00", ...}

the language {"0"}

the language {"1"}

- Operator <u>Precedence</u>:
  - Parentheses
  - Kleene Star
  - Concat (sometimes use $\circ$, 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?

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

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.

<u>Actually</u>:

- A **regular expression** <u>represents</u> a **regular language**
- The *set of all* **regular expressions** <u>represents</u> 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)

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

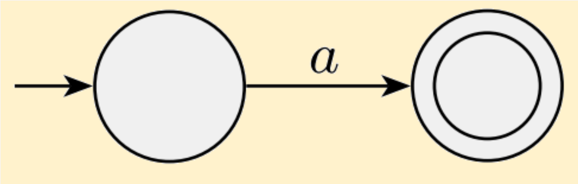
How to show that a language is regular?

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

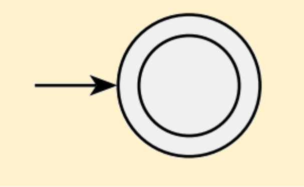
RegExpr→NFA

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

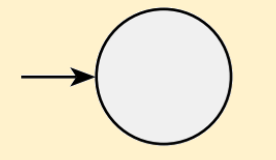
1. a for some a in the alphabet Σ,



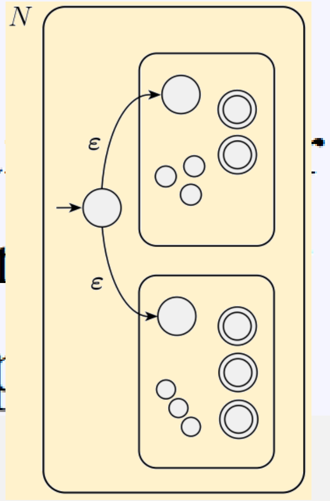
2. ε,

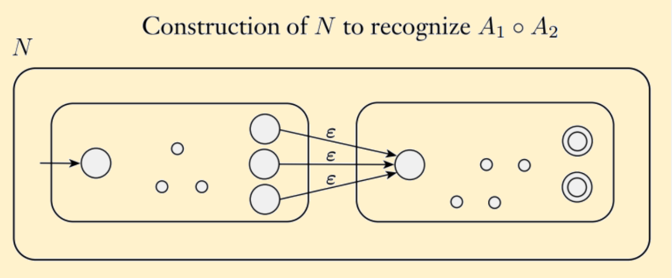


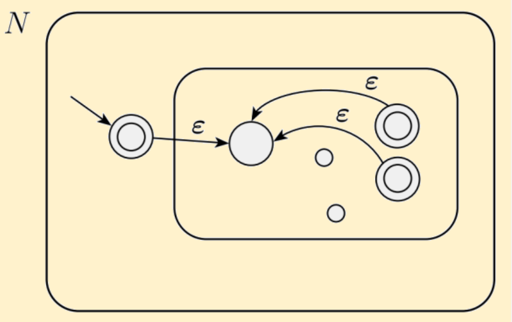
3. $\emptyset$,



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

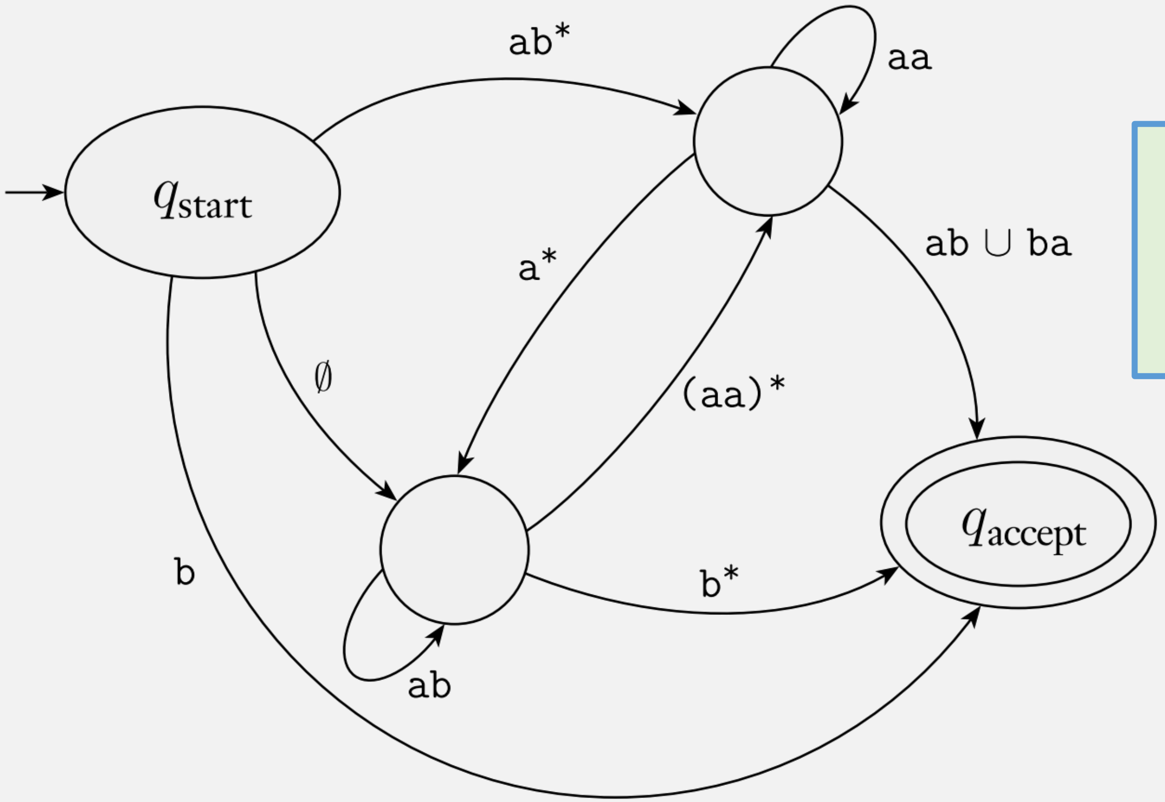
- To prove this part: Convert an DFA or NFA → equivalent Regular Expression
- To do so, **we first need another kind of finite automata: a <u>GNFA</u>**

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

☑ • Convert the regular expression → an equivalent NFA!

(could you write this as Statements and Justifications?)

Generalized NFAs (GNFAs)



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

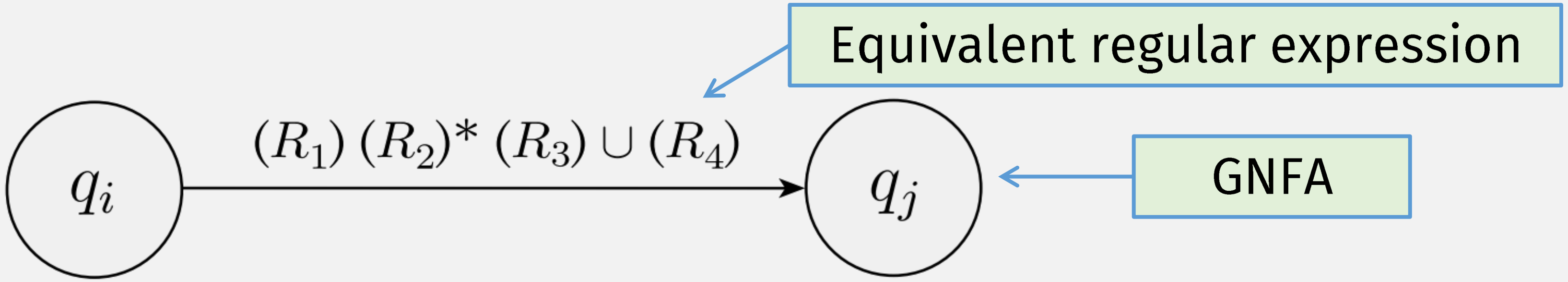
Goal: convert **GNFAs** to **Regular Exprs**

- 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.:



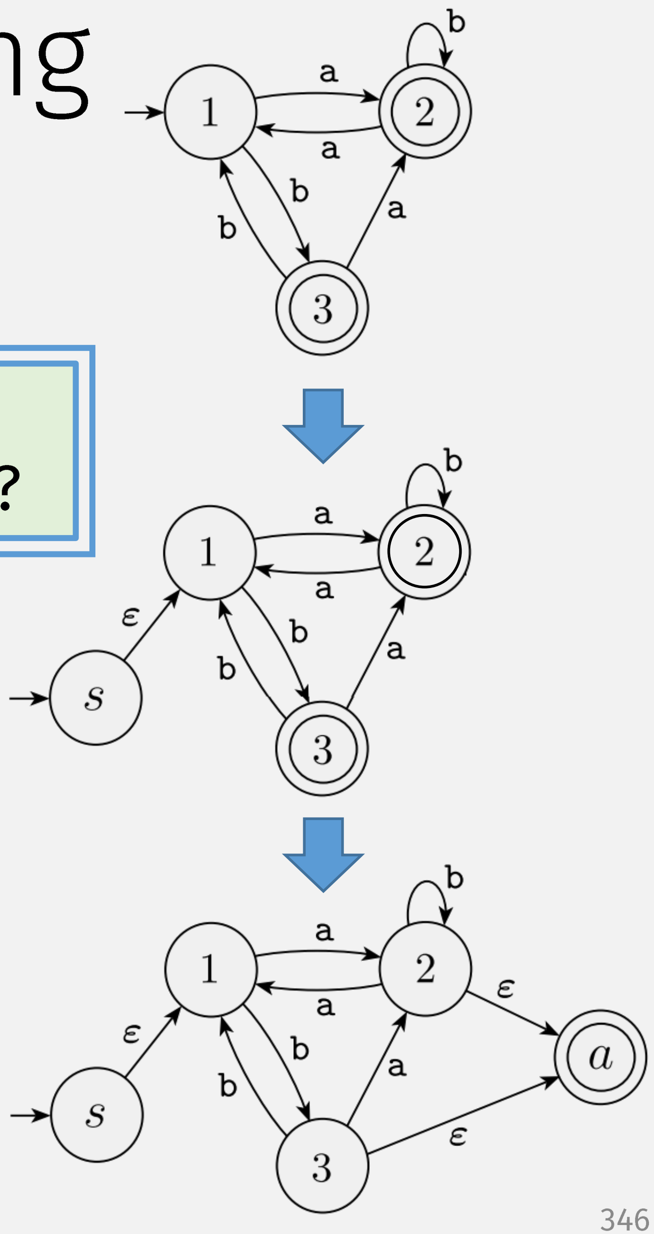
Could there be less than 2 states?

GNFA→RegExpr Preprocessing

- First, modify input machine to have:

Does this change the language of the machine?

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

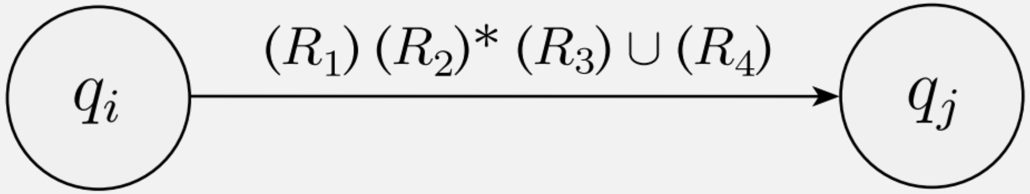


GNFA→RegExpr function (recursive)

On GNFA input G:

Base Case

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

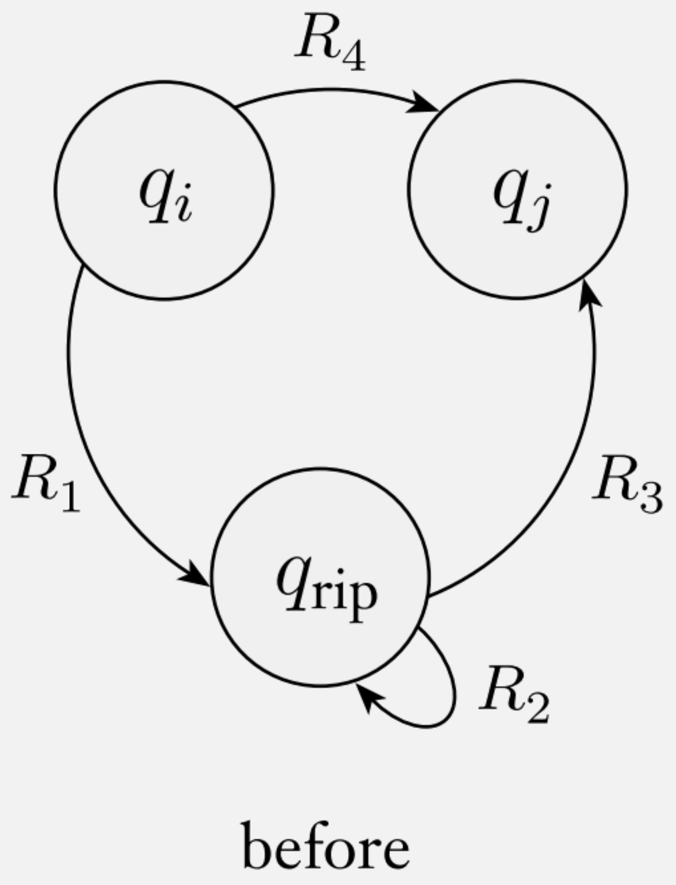


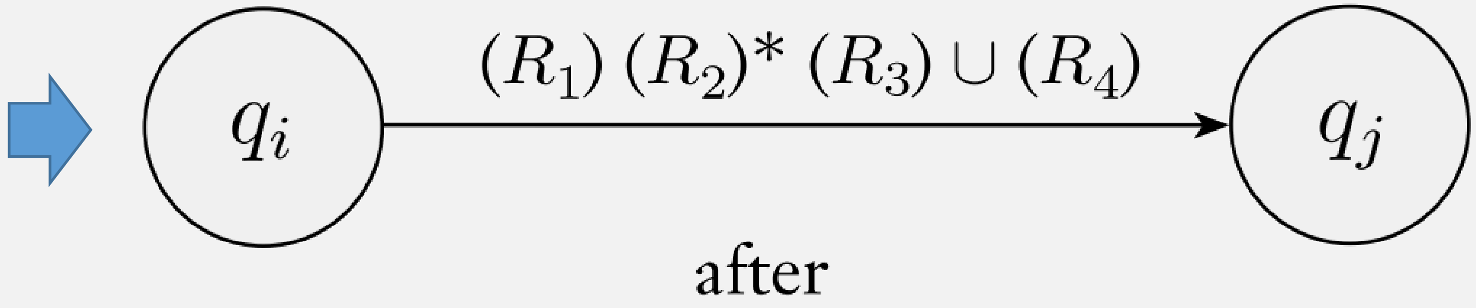
Recursive Case

- 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



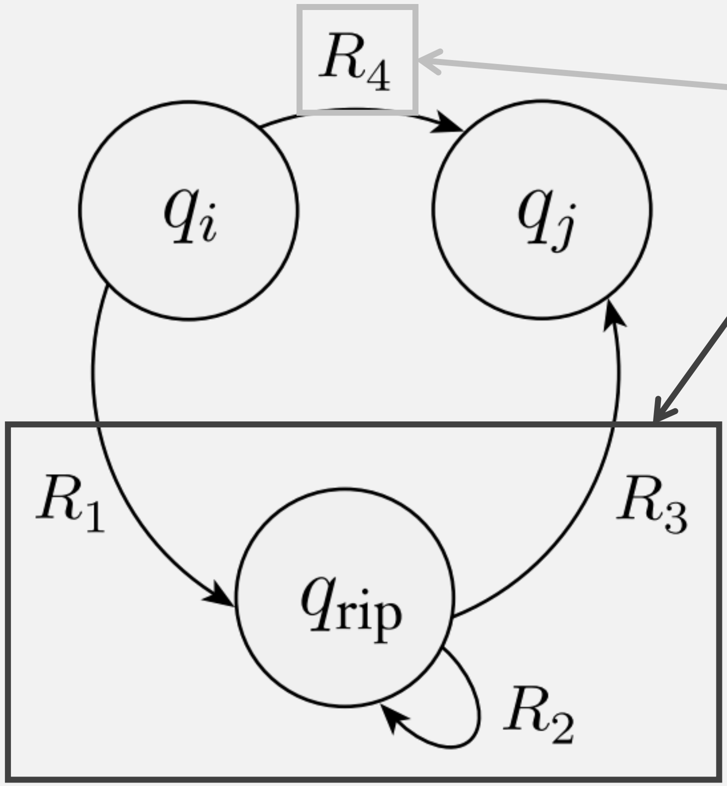


To <u>convert</u> a GNFA to a regular expression:
"rip out" state, then "repair",
and repeat until only 2 states remain

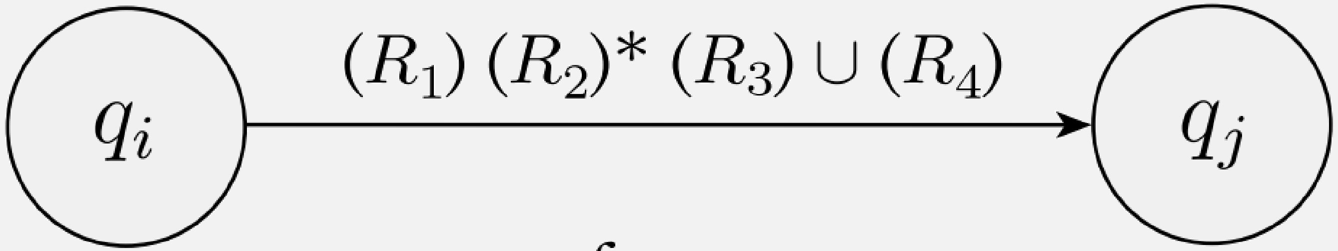
GNFA→RegExpr: “Rip/Repair” step

Before: two paths from q_i to q_j:

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



before

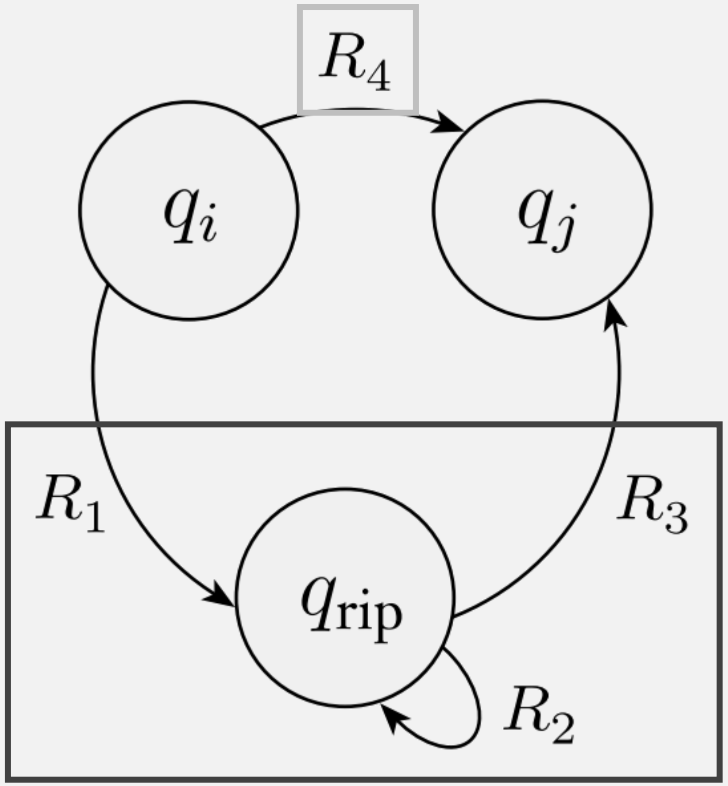


after

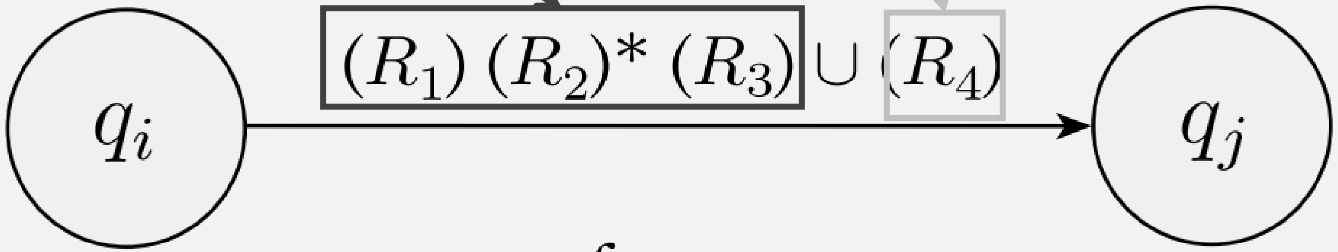
GNFA→RegExpr: "Rip/Repair" step

After: union of two "paths" from q_i to q_j

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

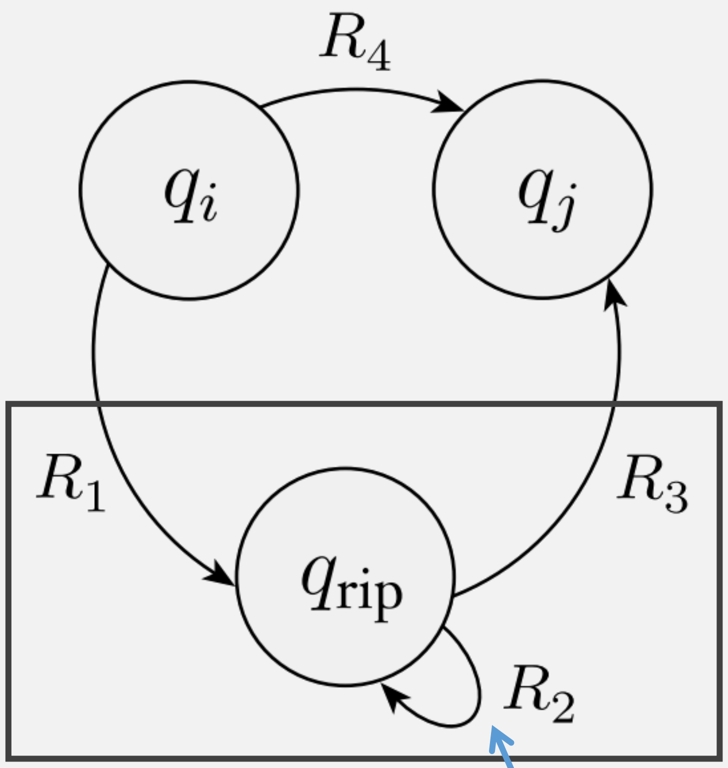


before

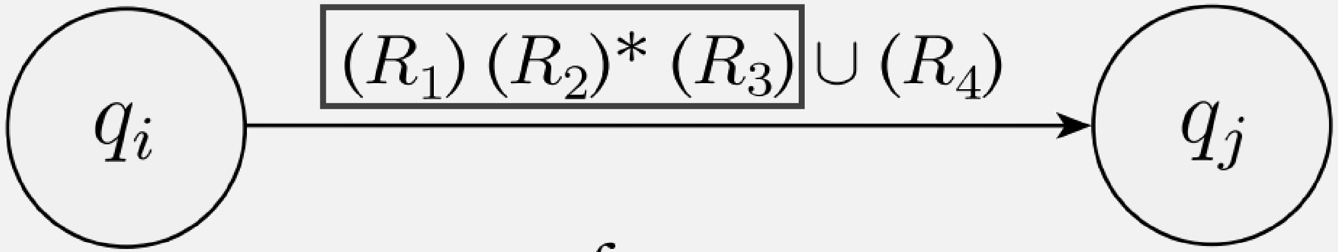


after

GNFA→RegExpr: "Rip/Repair" step



before

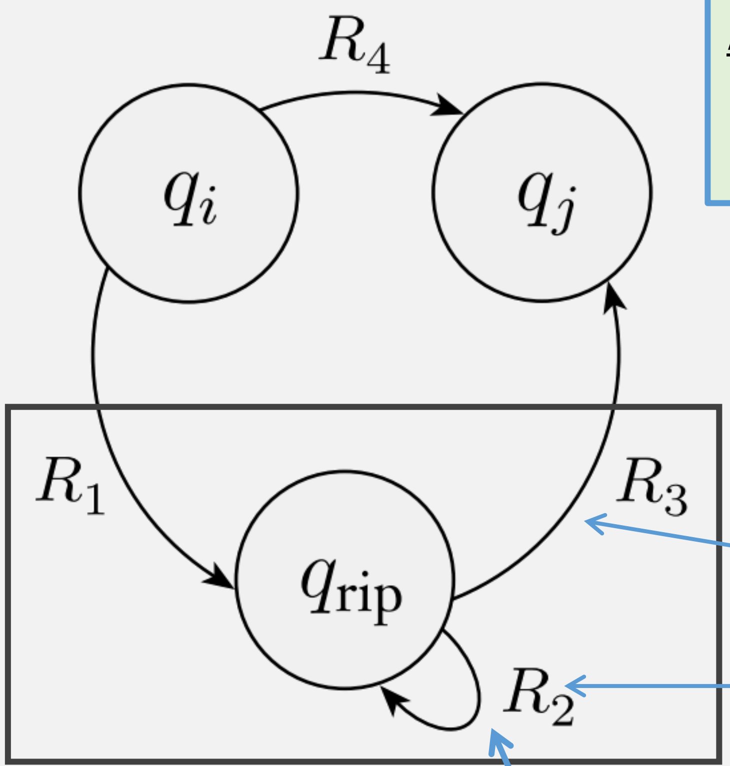


after

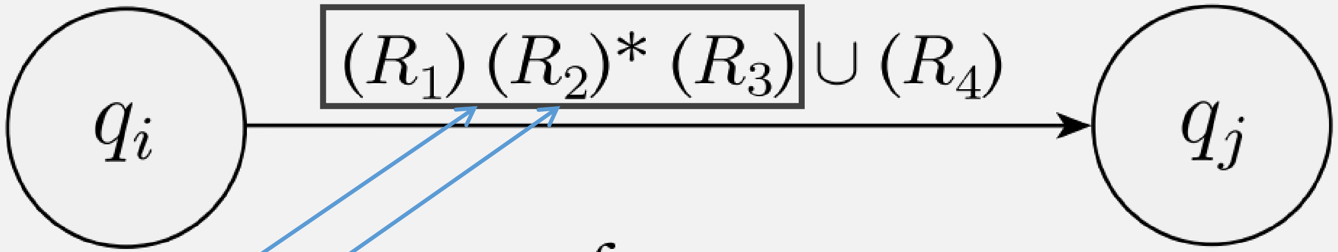
Before:
- path through q_{rip} has 3 transitions
- One is self-loop

GNFA→RegExpr: "Rip/Repair" step

<u>After</u>:
- Self loop becomes star operation
- Others are concat'ed together



before



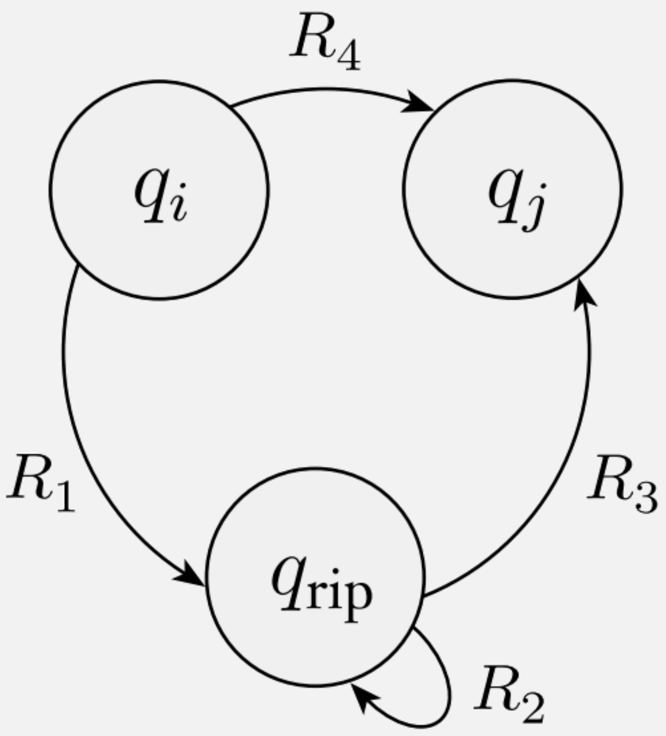
after

concat

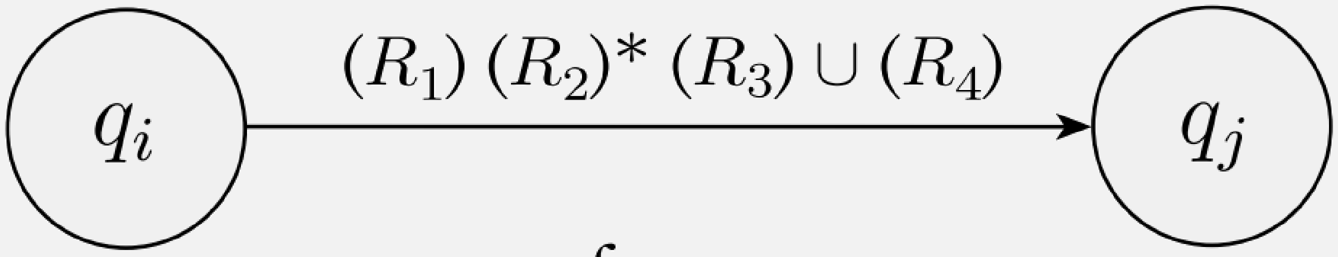
Star operation

<u>Before</u>:
- 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>

Equivalent = same language

GNFA→RegExpr "Correctness"

- "Correct" / "Equivalent" means:

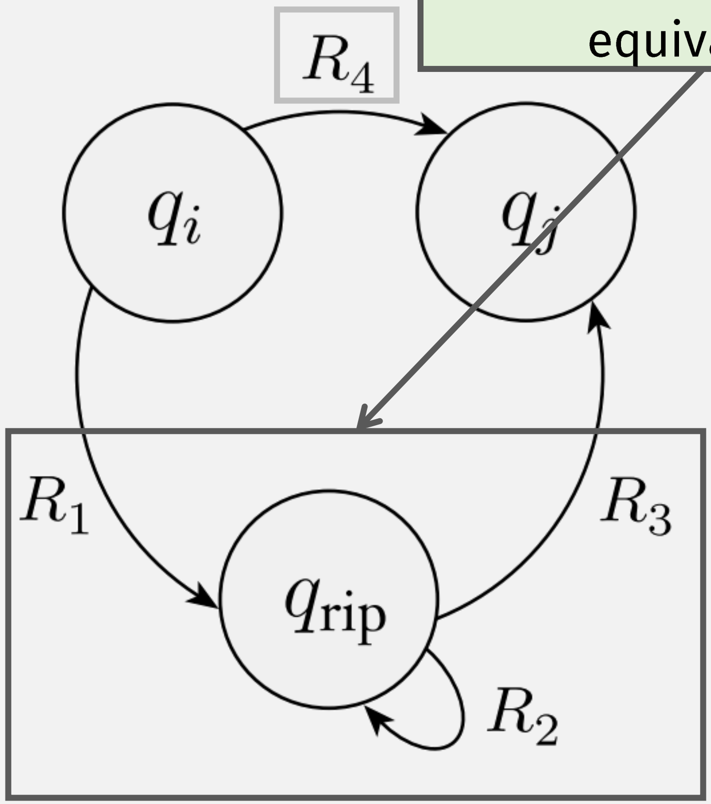
$$\text{LANGOF}(G) = \text{LANGOF}(R)$$

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

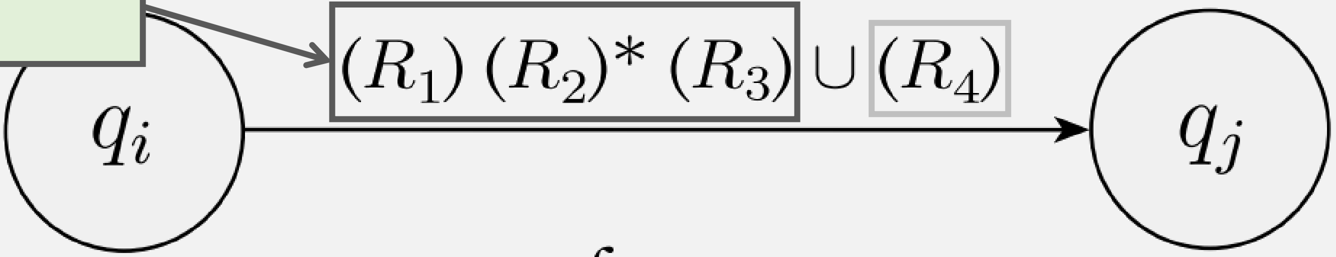
- i.e., **GNFA→RegExpr** must not change the language!
  - Key step: the rip/repair step

GNFA→RegExpr: Rip/Repair "Correctness"

Must show these are equivalent



before



after

Must prove:

- Every string accepted before, is accepted after
- 2 cases:
  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!

In-Class quiz 2/22

See gradescope