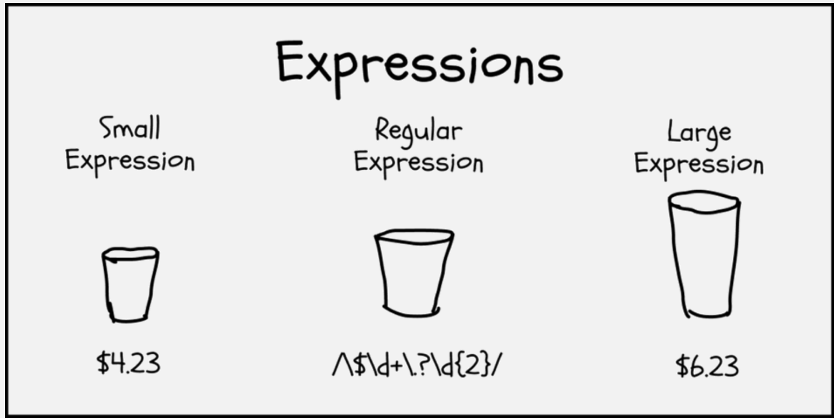
CS 420 / CS 620

Regular Expressions

Monday, October 6, 2025

UMass Boston Computer Science



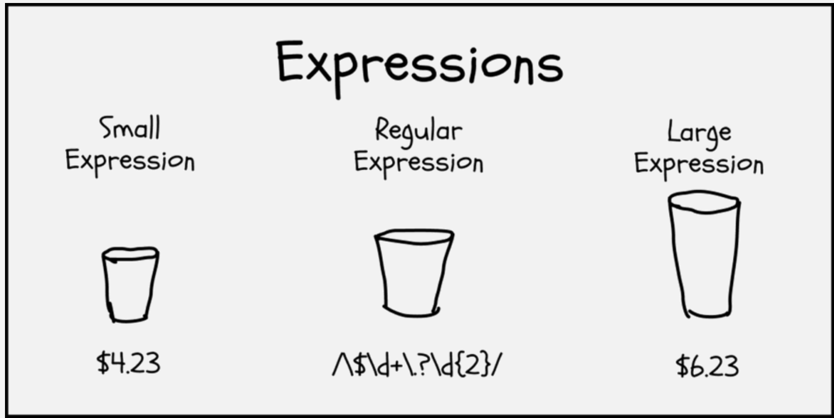
Announcements

- HW 4
  - ~~Due: **Mon 10/6 12pm (noon)**~~

- HW 5
  - Out: **Mon 10/6 12pm (noon)**
  - Due (unofficial): **Mon 10/13 12pm (noon)** (stay on schedule!)
  - Due (up to): **Wed 10/15 12pm (noon)**

- **HW 6** (most likely)
  - Out: **Mon 10/13 12pm (noon)**
  - Due: **Wed 10/20 12pm (noon)**

- **No class:** next **Mon 10/13** (Indigenous Peoples)



In-class question (in gradescope) preview

When used as a string, the epsilon symbol (ε) is equivalent to which of the following?

When used as the empty transition, the epsilon symbol (ε) is equivalent to which of the following?

When used as a regular expression, the epsilon symbol (ε) is equivalent to which of the following?

When used as an input to an NFA's single-step δ function, the epsilon symbol (ε) is which of the following?

When used as an input to an NFA's multi-step $\hat{\hat{\delta}}$ function, the epsilon symbol (ε) is which of the following?

When used as a transition label in a GNFA, the epsilon symbol (ε) is which of the following?

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" <u>zero</u>

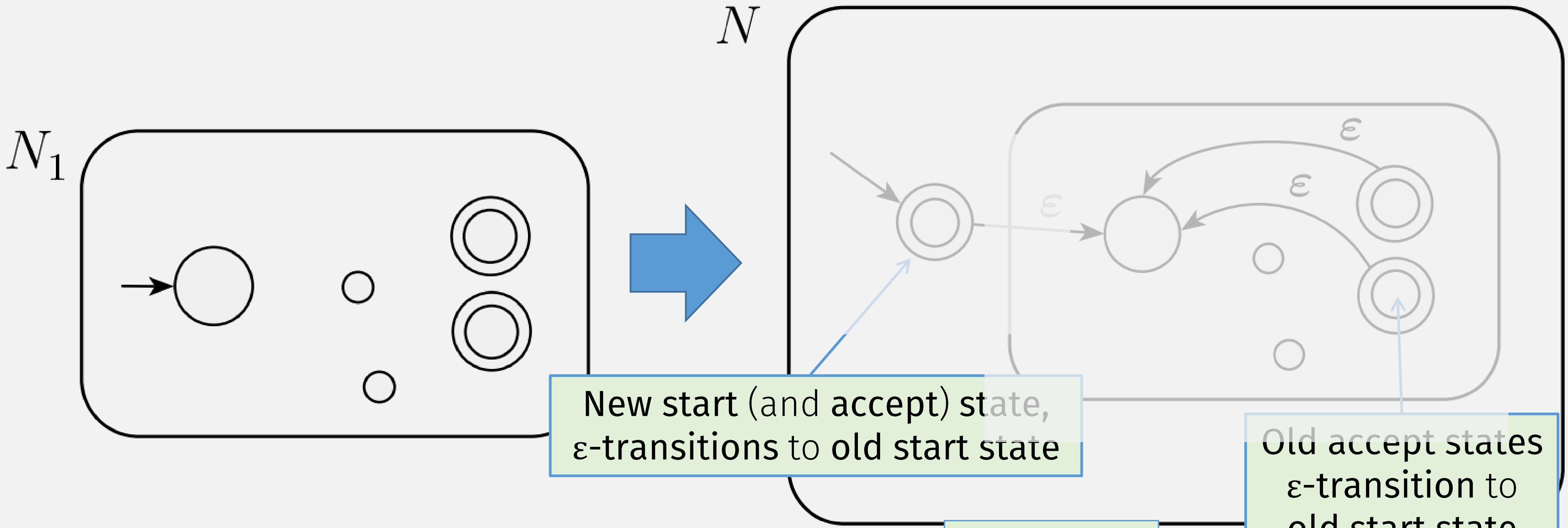
$$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$$

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

(this is an infinite language!)

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

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

Needs to accept
empty string!

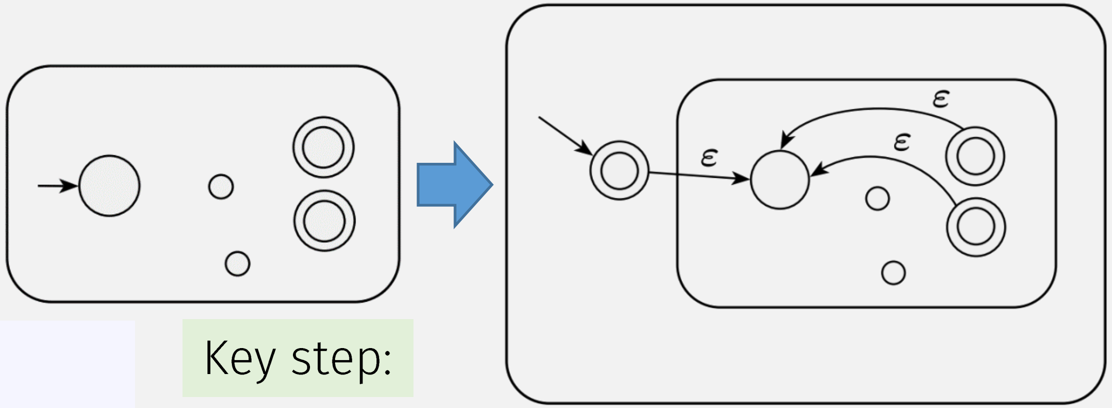
Recognizes language A

Recognizes language A^*

Note: repeats strings in A
<u>zero or more times</u>

In-class exercise

Kleene Star is Closed for Regular Langs



THEOREM

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

Key step:

$\mathrm{STAR}_{\mathrm{NFA}}$: NFA → NFA

Where:

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

$L(N) = L(N_1)^*$

In-class solution

Kleene Star is Closed for Regular Langs

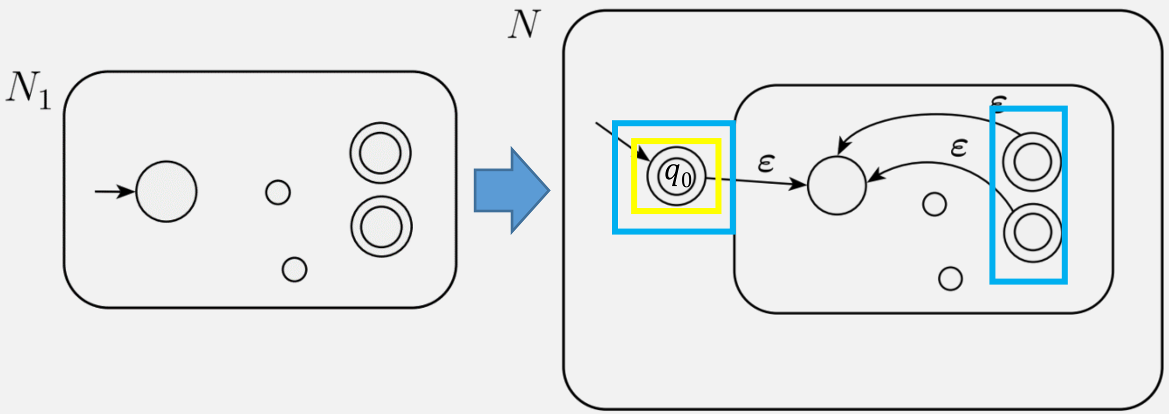
(part of)

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.

$N = \text{STAR}_{\text{NFA}}(N_1)$ $= (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$

Kleene star of a language must accept the empty string!



In-class solution

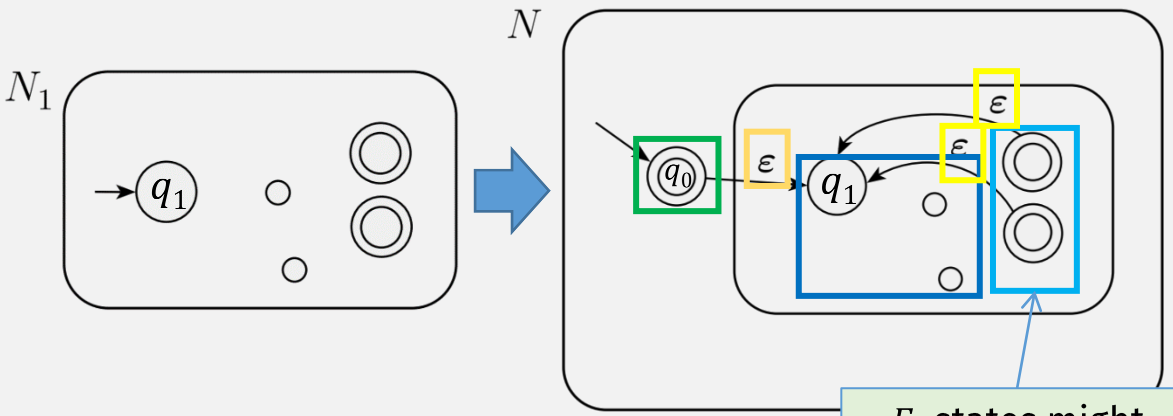
Sipser Thm 1.49

Kleene Star is Closed for Regular Langs

(part of)

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.

$N = \text{STAR}_{\text{NFA}}(N_1)$ $= (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.



F_1 states might previously have empty transitions!

1. $Q = \{q_0\} \cup Q_1$
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a)\,? & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a)\,? & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\}\,? & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\}\,? & q = q_0 \text{ and } a = \varepsilon \\ \emptyset\,? & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

F_1 states might previously have empty transitions!

Old accept states ε-transition to old start state q_1

New start state ε-transitions to old start state q_1

New start state has only ε-transitions

Must add transitions for any added states

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:**

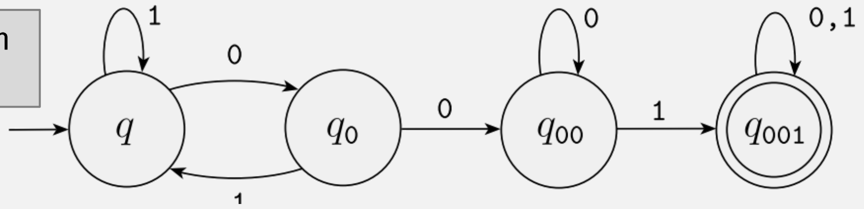
- (language of) **single-char strings** (from some alphabet)**, …**

- And these **three closed operations!**

e.g., lang {"a"} , lang {"b"} , …

So Far: Regular Language Representations

1. State diagram (NFA/DFA)



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

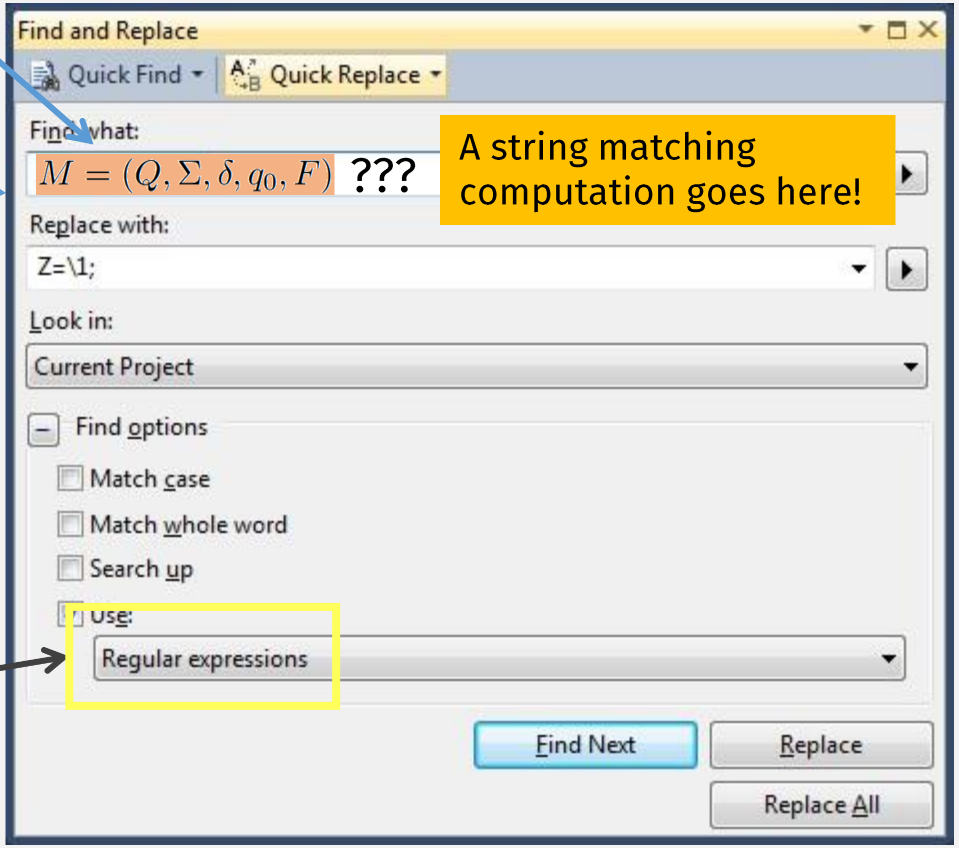
2. Formal description

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. $F = \{q_2\}$

(hard to write)



Our Running Analogy:
- Set of all **regular languages** ~ a "programming language"
- One **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

```
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** $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$
- **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

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

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

Base case

Recursive case

Self-reference

Recursive call with “smaller” argument

Flashback: Recursive Definitions

A **Natural Number** is either:

Self-reference

Base case
- **Zero**, or

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

"smaller" argument

Flashback: Recursive Definitions

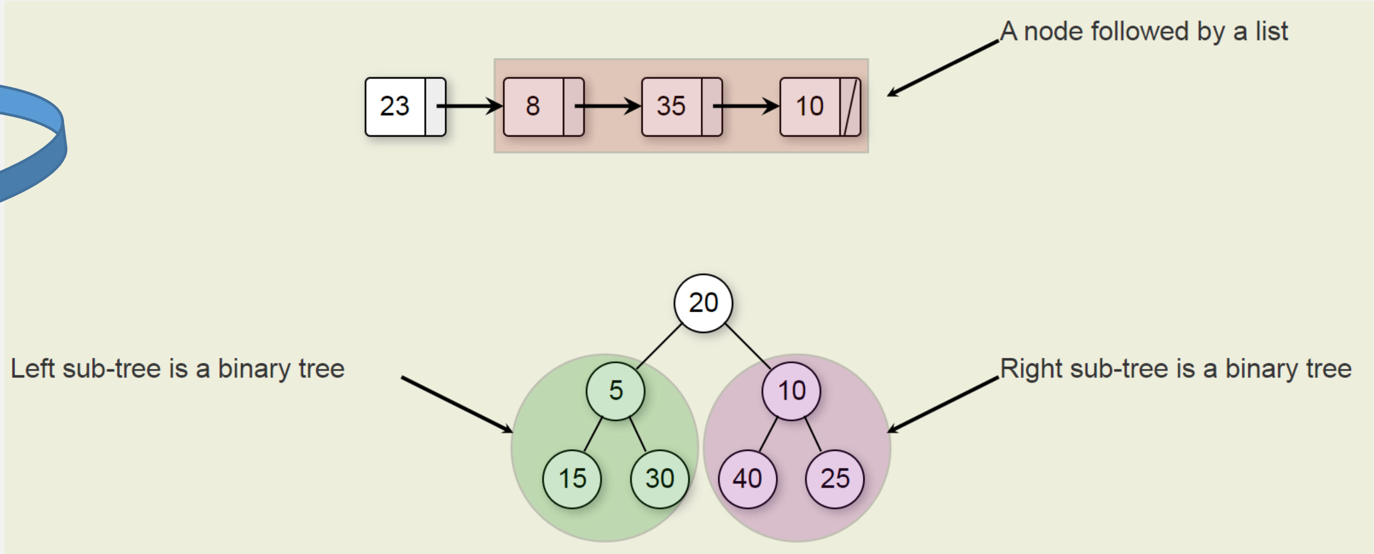
```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

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)
4. union → $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. concat → $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. star → (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

Entire **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 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

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

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

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

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.

We would like to say:

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

(But we have to prove it)

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

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

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

(Easier)

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

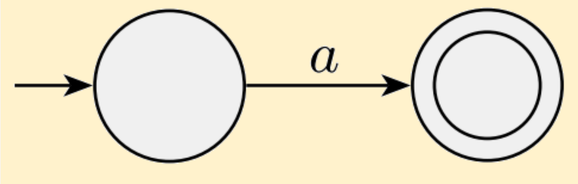
How to show that a language is regular?

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

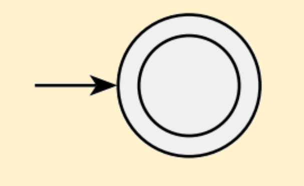
RegExpr→NFA

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

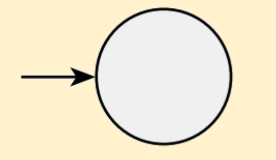
1. a for some a in the alphabet Σ,



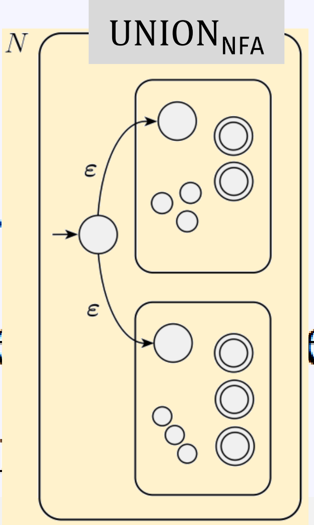
2. ε,



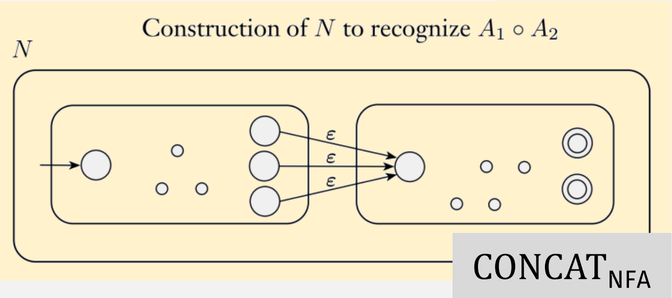
3. $\emptyset$,



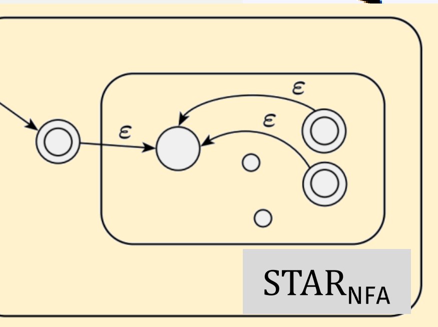
4. $(R_1 \cup R_2)$, where R_1 and R_2 ar … expressions,



5. $(R_1 \circ R_2)$, where R_1 and R_2 are …

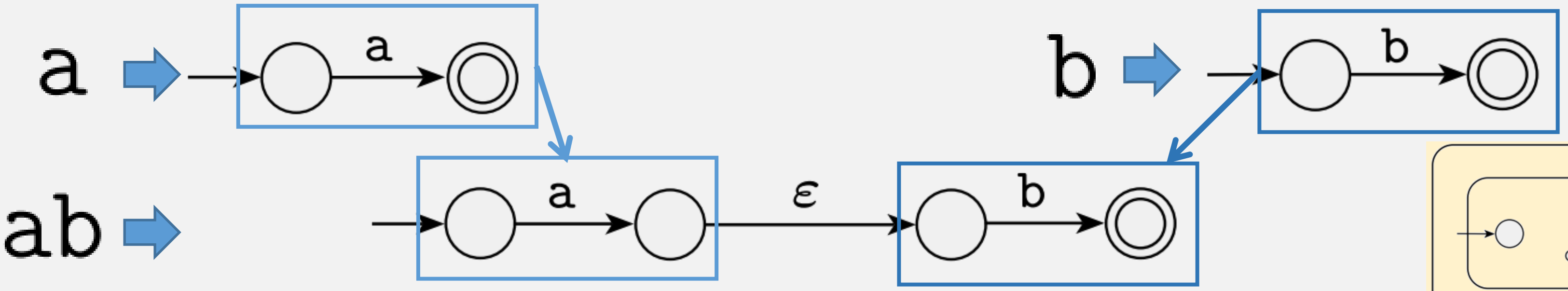


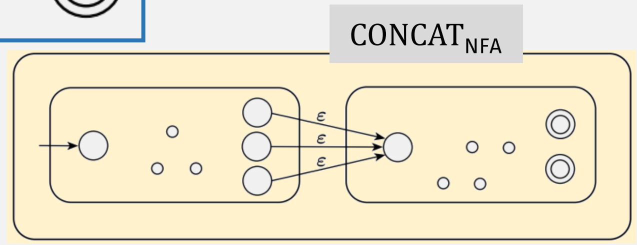
(R_1^*), where R_1 is a regular expr…



RegExpr→NFA: Example

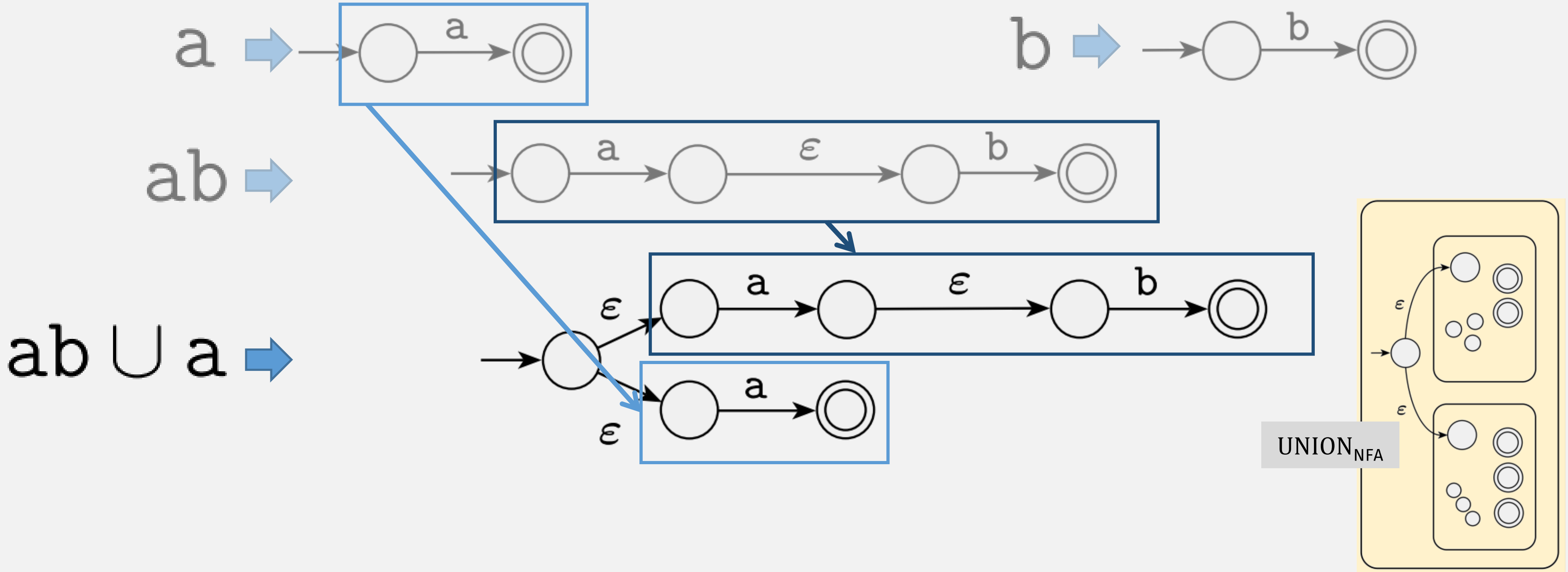
convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA ... step by step





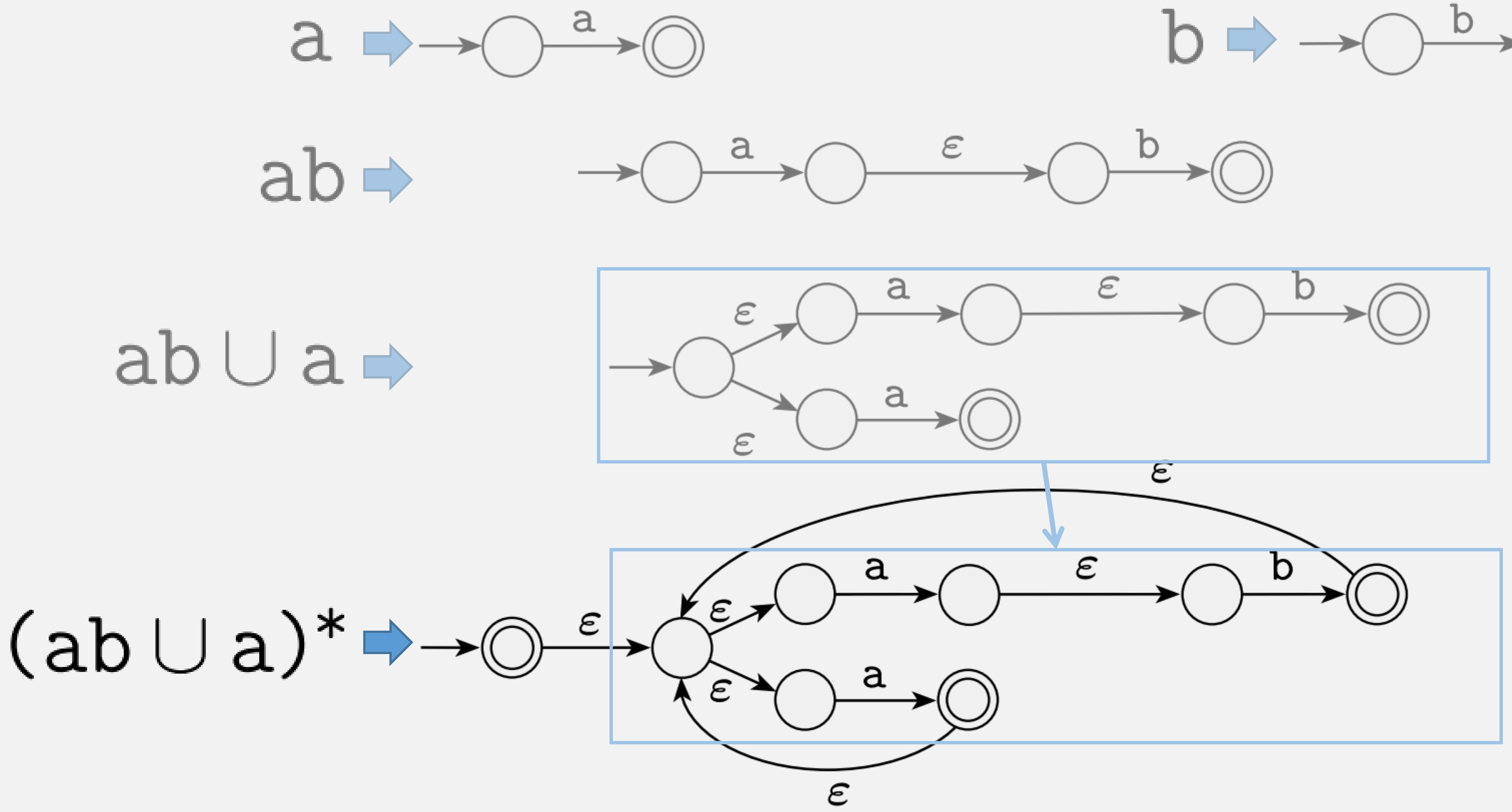
RegExpr→NFA: Example

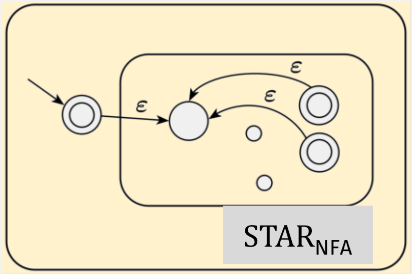
convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA



RegExpr→NFA: Example

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





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

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

(Harder)

GNFA

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

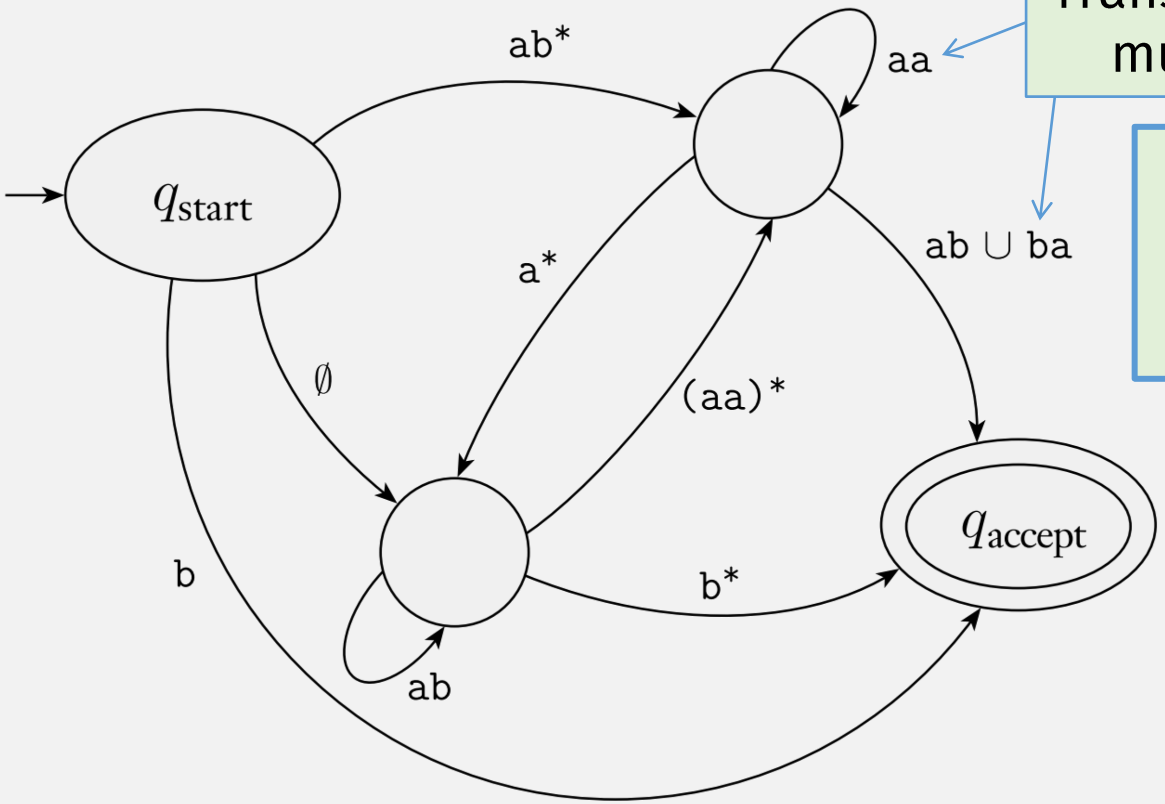
⇐ If **a language is described by a reg expression,** then **it's regular**

(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

plain NFA
= GNFA with single char regular expr transitions

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

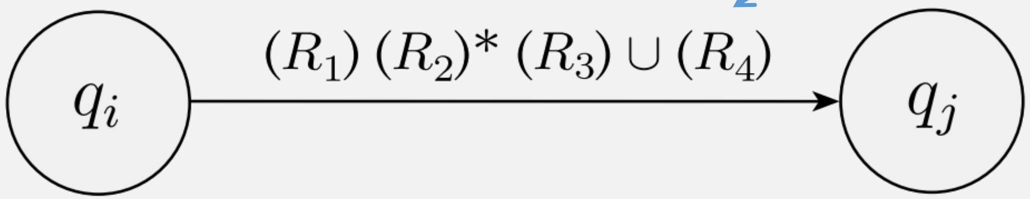
- GNFA = NFA with regular expression transitions

`GNFA→RegExpr` function :

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

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

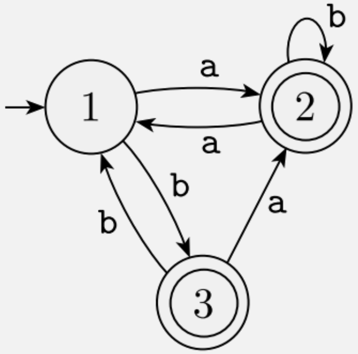
Equivalent regular expression



GNFA

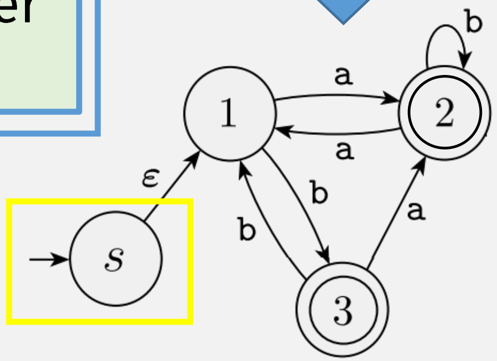
Could there be less than 2 states?

GNFA→RegExpr Preprocessing

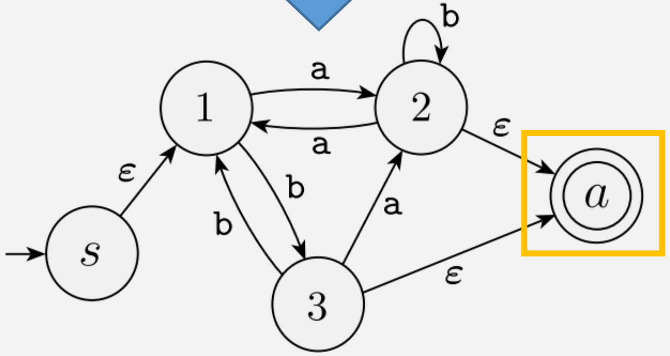


- Modify input machine to have:

> Does this change the language of the machine? i.e., are **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

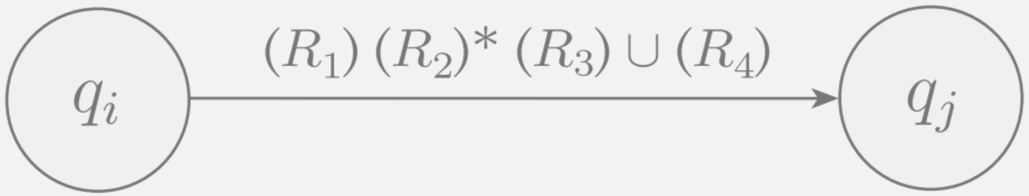
> Modified machine always has 2+ states:
> - <u>New start state</u>
> - <u>New accept state</u>

GNFA→RegExpr function (recursive)

On GNFA input G:

Base Case

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

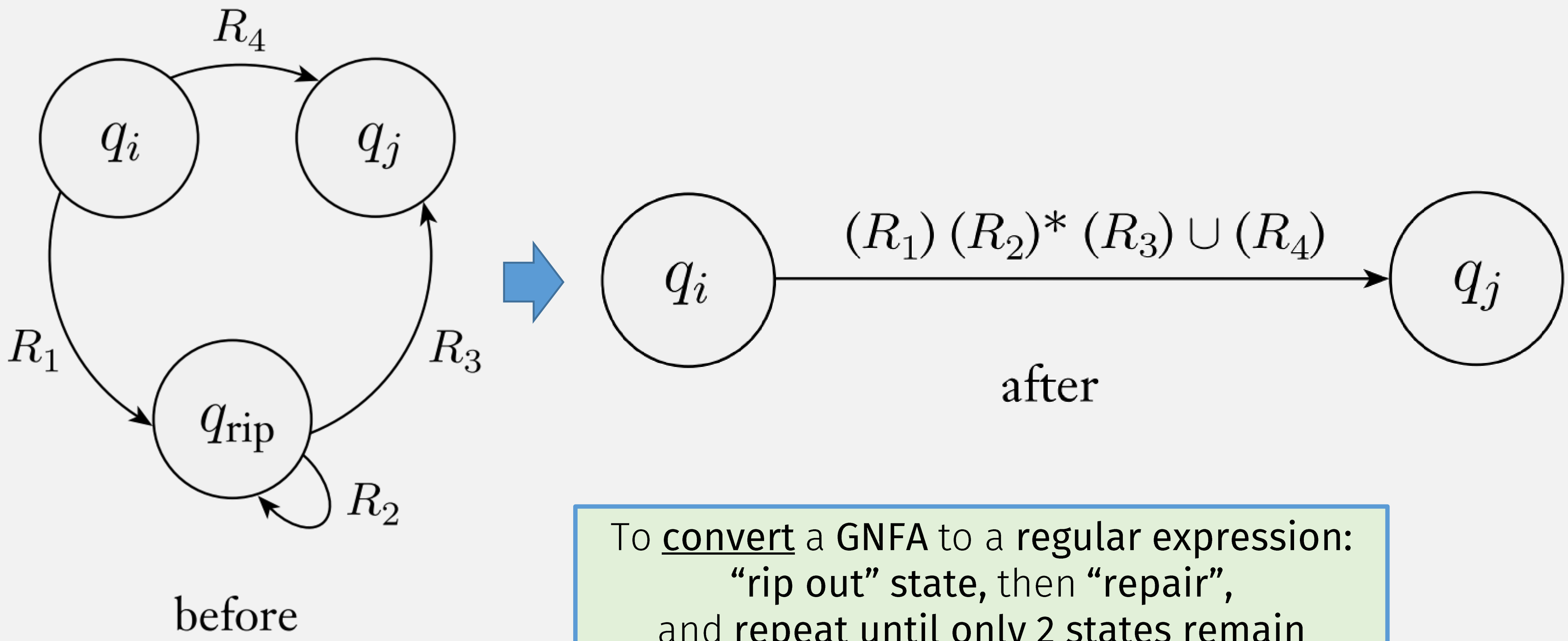


Recursive Case

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

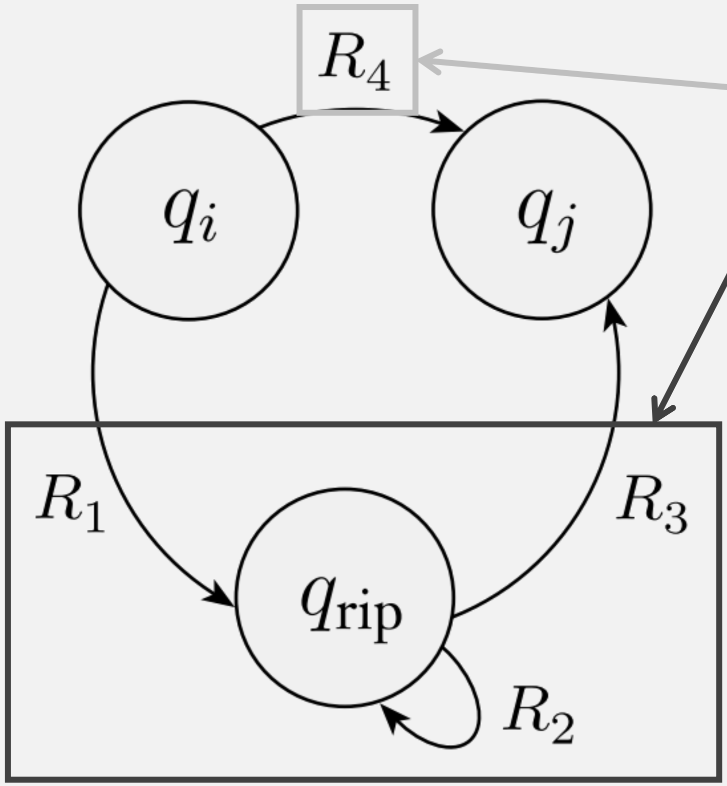
Recursive definitions have:
- base case and
- recursive case
  (with "smaller" self-reference)

`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**

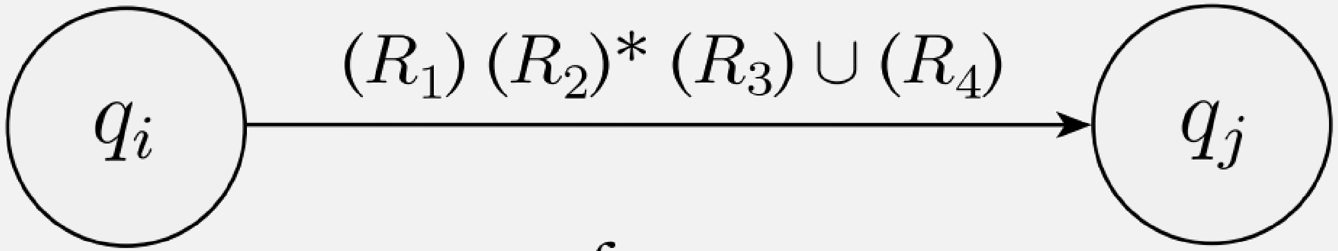
GNFA→RegExpr: "Rip/Repair" step



before

Before: two paths from q_i to q_j:

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

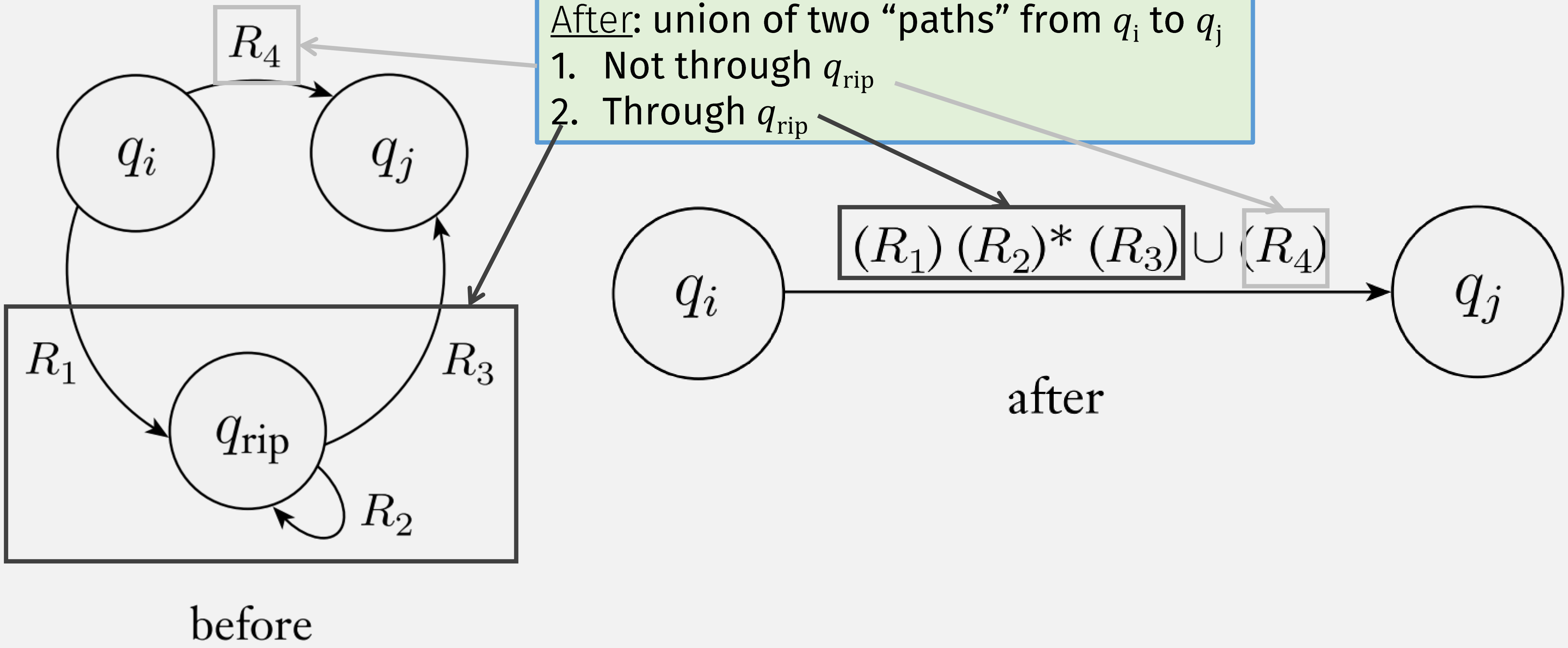


after

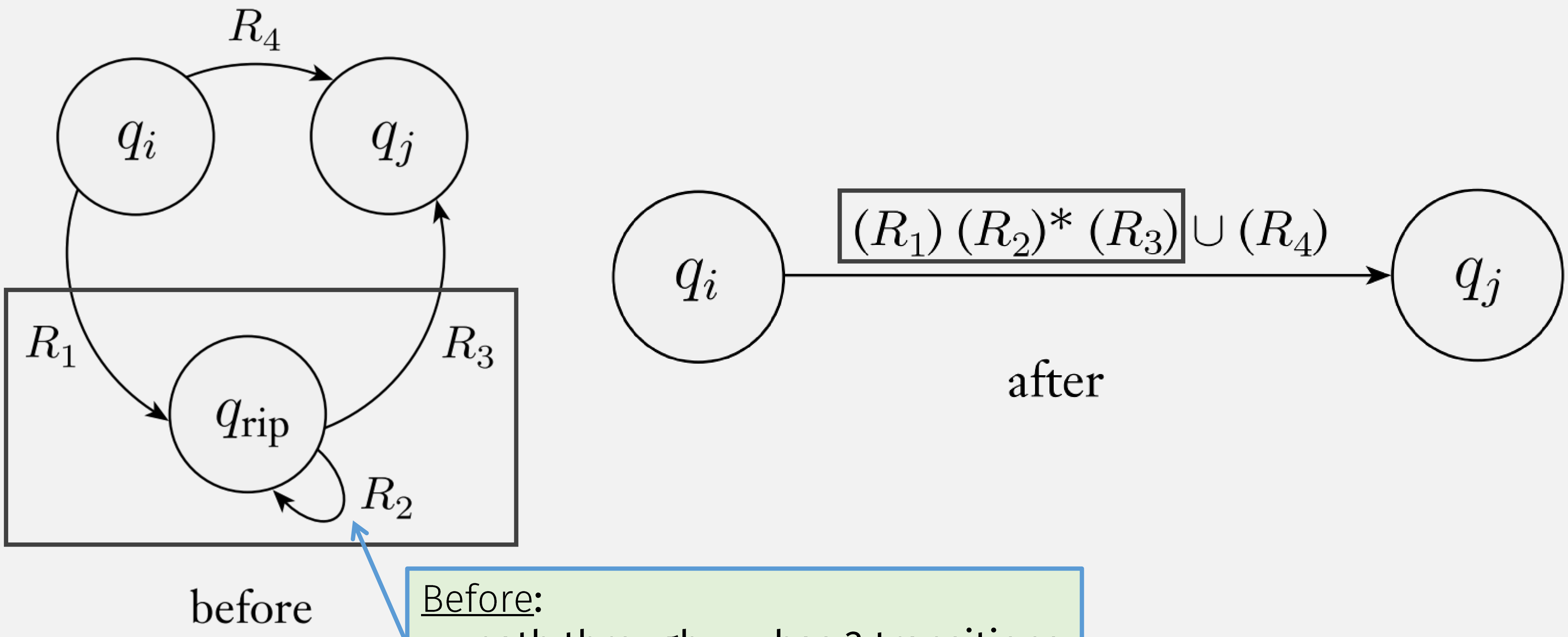
`GNFA→RegExpr`: "Rip/Repair" step

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

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

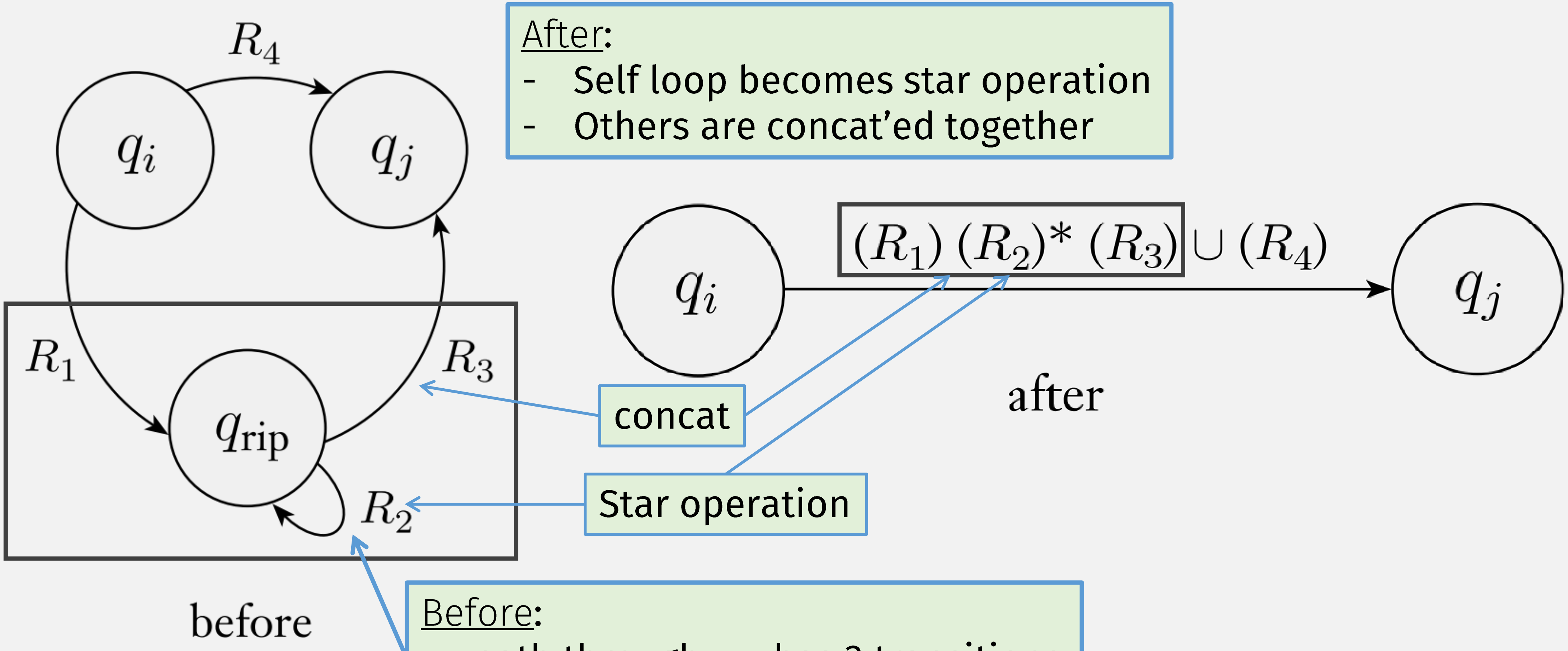


GNFA→RegExpr: "Rip/Repair" step



Before:
- 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

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

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

Need to convert DFA or NFA to Regular Expression …

- Use **GNFA→RegExpr** to convert GNFA → equiv regular expression!

☑ ???

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

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

☑ • Convert regular expression → equiv NFA!

GNFA→RegExpr Correctness

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

Statement to Prove:

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

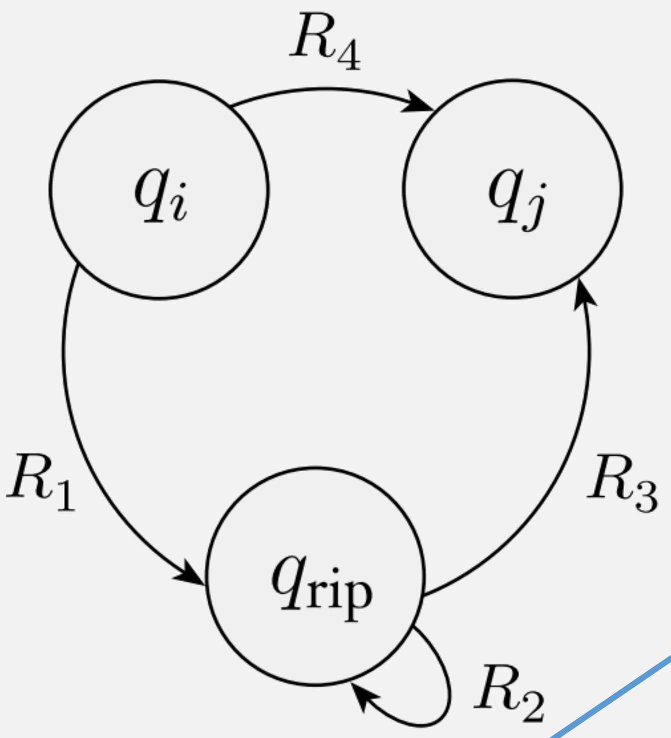
???

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

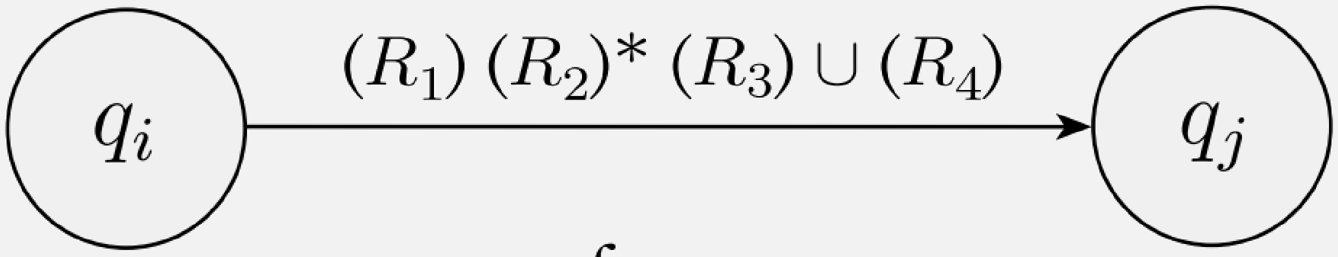
- where:
  - G = a GNFA
  - R = a Regular Expression = GNFA→RegExpr(G)

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

GNFA→RegExpr: Rip/Repair Correctness



before



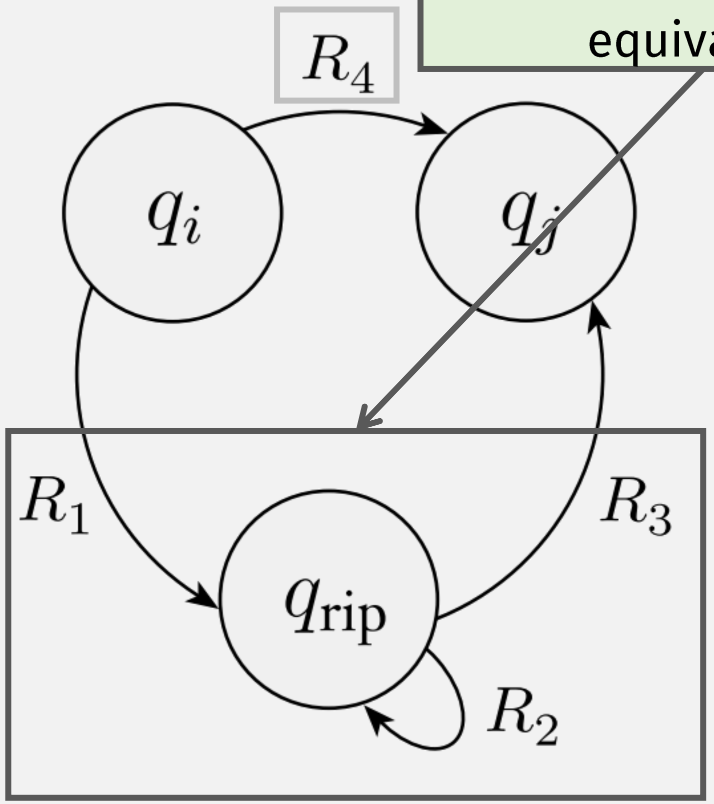
after

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

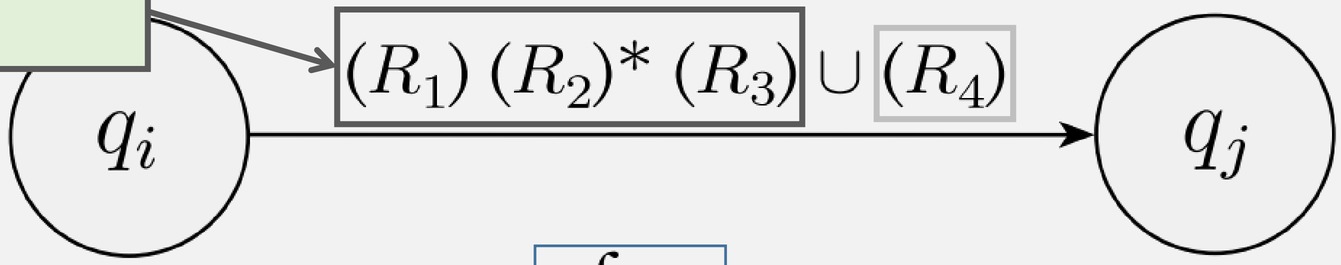
Equivalent =
same language =
accepts the same strings

`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` Equivalence

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

Statement to Prove:

input

output **???**

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

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

- <u>where</u>:
  - G = a GNFA
  - R = a Regular Expression = `GNFA→RegExpr`(G)

Language could be infinite set of strings!

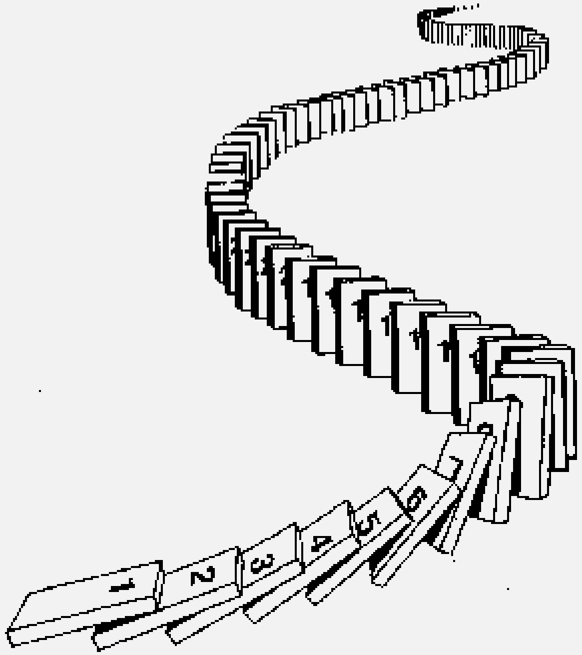
(how can we show **equivalence** for a possibly **infinite set of strings**?)

Recursion!

Next Time

Inductive Proofs

(recursive)



Proof by Induction

Previously:

Recursive function

- Use it when: **writing a function involving** a **recursive definition**

Now:

Proof by induction (recursion) = **“a recursive proof”**

- Use it when: **proving something involving** a **recursive definition**

The recursive definition is (always) **the key!**

A valid recursive definition has:
- **base case**(s) and
- **recursive case**(s) (with “smaller” self-reference)

Proof by Induction

(A proof for **each case** of some recursive definition)

To Prove: ***Statement*** for recursively defined "thing" x:

1. Prove: *Statement* for base case of x
2. Prove: *Statement* for recursive case of x:
   - Assume: **induction hypothesis** (IH)

     I.e., *Statement* **is true for** some x_{smaller} (This is just the **recursive part** from the **recursive definition**!)
     - E.g., if x **is number,** then **"smaller" = lesser number**

   - Prove: *Statement* **for** x**, using IH** (and known definitions, theorems …)
     - Typically: **show that going from** x_{smaller} **to larger** x **is true!**

i.e., a normal proof

A valid recursive definition has:
- **base case**(s) and
- **recursive case**(s) (with **"smaller"** self-reference)

Natural Numbers Are Recursively Defined

A **Natural Number** is:

Base Case
- **0**

Recursive Case
- Or $\boldsymbol{k+1}$**,** where k is a **Natural Number**

Self-reference

Recursive definition is valid because self-reference is "smaller"

So, **proving** things about:
recursive Natural Numbers <u>requires</u>
recursive proof,
i.e., **proof by induction!**

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

Proof By Induction Example (Sipser Ch 0)

Prove true: $$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right)$$

- P_t = loan balance after t months
- t = # months
- P = principal = original amount of loan
- M = interest (multiplier)
- Y = monthly payment

(Details of these variables not too important here)

Proof By Induction Example (Sipser Ch 0)

Prove true: $P_t = PM^{\boxed{t}} - Y\left(\frac{M^{\boxed{t}} - 1}{M - 1}\right)$

Proof: by **induction** on natural number $\boxed{t}$

A proof by induction follows the cases of the **recursive definition** (here, **natural numbers**) that the **induction** is "on"

Base Case, $t = 0$:

A Natural Number is:

- 0 ☑
- Or $k + 1$, where k is a natural number

$$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right) = P$$

Plug in $t = 0$

Simplify

$P_0 = P$ is a true statement !
(amount owed at start = loan amount)

Proof By Induction Example (Sipser Ch 0)

A **proof by induction** follows cases of **recursive definition** (here, **natural numbers**) that the **induction is "on"**

Prove true: $$P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right)$$

A Natural Number is:

- 0 ☑
- $k + 1$, for some **nat num** k

Inductive Case: $t = k + 1$, for some **natural num** k

- **Inductive Hypothesis** (IH), **assume statement is true** for some $t =$ (smaller) k

IH plugs in "smaller" k

$$P_k = PM^k - Y\left(\frac{M^k - 1}{M - 1}\right)$$

Goal **statement to prove,** for $t = k+1$:

$$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right)$$

Write $t = k+1$ case in terms of "smaller" k

Plug in IH for P_k

Simplify, to get to goal statement

- Proof of Goal:

$$P_{k+1} = P_k M - Y$$

Definition of Loan:
amt owed in month $k+1$ =
amt owed in month k * interest M – amt paid Y

In-class Exercise: Proof By Induction

A **proof by induction** follows cases of **recursive definition** (here, **natural numbers**) that the **induction is "on"**

A Natural Number is:

- 0
- $k + 1$, for some **nat num** k

Prove: $(z \neq 1)$

$$\sum_{i=0}^{m} z^i = \frac{1 - z^{m+1}}{1 - z}$$

Use Proof by Induction.

Make sure to: clearly **state what the induction** is "**on**"

i.e., which **recursively defined value** (and its **type**) will the **proof focus on**

Proof by Induction: CS 420 Example

Statement to prove: $\boxed{\text{LANGOF}(G) = \text{LANGOF}(\ R = \textbf{GNFA→RegExpr}(G)\ )}$

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

- i.e., **GNFA→RegExpr** must not change the language!

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

Proof by Induction: CS 420 Example

Statement to prove: LANGOF (G) = LANGOF (**GNFA→RegExpr**(G))

Recursively defined "thing"

Proof: by Induction on # of states in G

1. Prove *Statement* is true for base case

G has 2 states

Why is this an ok base case (instead of zero)?

(Modified) **Recursive definition:**

A "NatNumber > 1" is:

- 2
- Or k + 1, where k is a "NatNumber > 1"

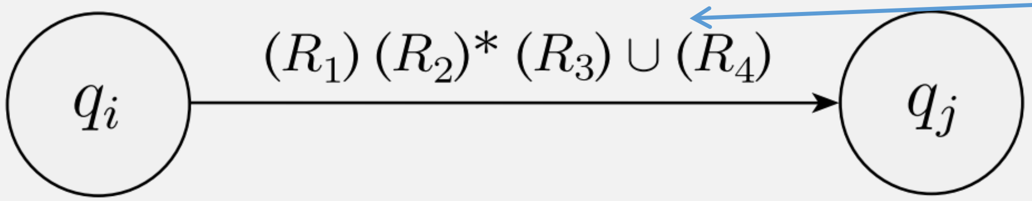
Last Time

GNFA→RegExpr (recursive) function

On GNFA input G:

Base Case

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



Equivalent regular expression

GNFA

Proof by Induction: CS 420 Example

Statement to prove: $\boxed{\textsc{LangOf}\,(\,G\,) = \textsc{LangOf}\,(\,\texttt{GNFA→RegExpr}(\,G\,)\,)}$

Plug in

Proof: by Induction on # of states in G

☑ 1. Prove *Statement* is true for base case

G has 2 states

$q_i \xrightarrow{R} q_j$

Statements	**Justifications**
1. $\textsc{LangOf}\,(\,q_i \xrightarrow{R} q_j\,) = \textsc{LangOf}\,(\,R\,)$	1. Definition of GNFA
2. $\texttt{GNFA→RegExpr}(\,q_i \xrightarrow{R} q_j\,) = R$	2. Definition of **GNFA→RegExpr** (base case)
3. $\textsc{LangOf}\,(\,q_i \xrightarrow{R} q_j\,) = \textsc{LangOf}\,(\,\texttt{GNFA→RegExpr}(\,q_i \xrightarrow{R} q_j\,)\,)$	3. From (1) and (2)

Plug in R

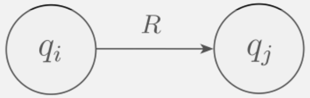
Goal

Don't forget the
Statements / Justifications !

Proof by Induction: CS 420 Example

Statement to prove: $\boxed{\text{LangOf}(G) = \text{LangOf}(\texttt{GNFA→RegExpr}(G))}$

Proof: by Induction on # of states in G

☑ 1. Prove *Statement* is true for base case $\boxed{G \text{ has 2 states}}$ 

2. Prove *Statement* is true for recursive case: $\boxed{G \text{ has} > 2 \text{ states}}$

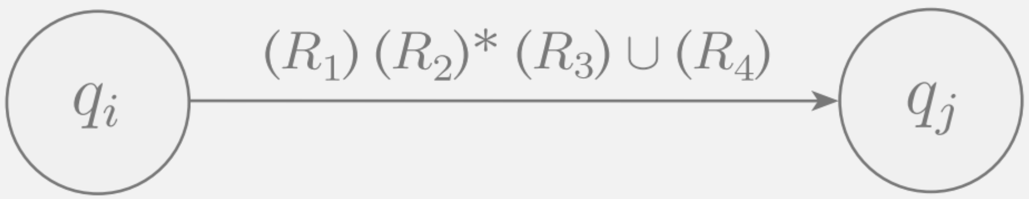
Last Time

GNFA→RegExpr (recursive) function

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

Base Case

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



- Else:

Recursive Case

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

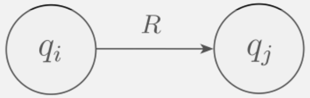
Recursive call (with a "smaller" G')

Proof by Induction: CS 420 Example

Statement to prove: $\boxed{\textsc{LangOf}\,(\,G\,) = \textsc{LangOf}\,(\,\textbf{GNFA→RegExpr}(\,G\,)\,)}$

<u>Proof</u>: by Induction on # of states in *G*

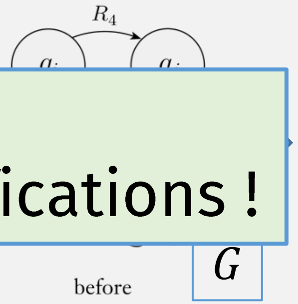
☑ 1. <u>Prove</u> *Statement* is true for <u>base case</u> $\boxed{G \text{ has 2 states}}$



2. <u>Prove</u> *Statement* is true for <u>recursive case</u>: $\boxed{G \text{ has} > 2 \text{ states}}$
   - <u>Assume</u> the **induction hypothesis** (IH):
     - *Statement* is true for smaller *G’*
   - <u>Use</u> it to prove *Statement* is true for *G* > 2 states
     - Show that going from *G* to smaller *G’* is true!

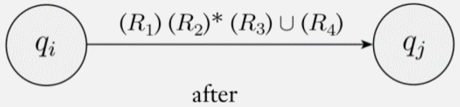
IH Assumption

LANGOF (*G’*)
=
LANGOF (**GNFA→RegExpr**(*G’*))
(Where *G’* has less states than *G*)



G

Don’t forget the Statements / Justifications !



smaller *G’*

Show that “rip/repair” step ☑ converts *G* to smaller, equivalent *G’*

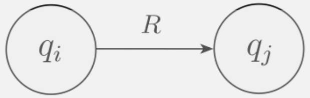
Proof by Induction: CS 420 Example

Statement to prove: $\boxed{\textsc{LangOf}\,(\,G\,) = \textsc{LangOf}\,(\,\textbf{GNFA→RegExpr}(\,G\,)\,)}$

Proof: by Induction on # of states in G

☑ 1. Prove *Statement* is true for base case

G has 2 states



☑ 2. Prove *Statement* is true for recursive case:

G has > 2 states

- Assume the i
  - *Statement*
- Use it to prov tates
  - Show that e!

Known "facts" available to use:
- ☑IH
- ☑Equiv of Rip/Repair step
- ☑Def of GNFA->RegExpr

LangOf (G')
=
LangOf (**GNFA→RegExpr**(G'))
(Where G' has less states than G)

Statements	**Justifications**
1. LangOf (G') = LangOf (**GNFA→RegExpr**(G'))	1. IH
2. LangOf (G) = LangOf (G')	2. Equivalence of Rip/Repair step (prev)
3. **GNFA→RegExpr**(G) = **GNFA→RegExpr**(G') Plug in	3. Def of **GNFA→RegExpr** (recursive call)
Goal 4. LangOf (G) = LangOf (**GNFA→RegExpr**(G))	4. From (1), (2), and (3)

<u>Thm</u>: A Lang is Regular **iff** Some Reg Expr Describes It

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

☑ • Use **GNFA→RegExpr** to convert GNFA → equiv regular expression!

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

☑ • Convert regular expression → equiv NFA! ■

Now: **we can 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!

So Far: How to Prove A Language Is Regular?

Key step, either:

- 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.

Proof by Induction

To Prove: a ***Statement*** about a recursively defined "thing" x:

1. Prove: *Statement* for base case of x
2. Prove: *Statement* for recursive case of x:
   - Assume: **induction hypothesis** (IH)

     I.e., *Statement* is true for some x_{smaller}
     - E.g., if x is number, then "smaller" = lesser number
     - E.g., if x is regular expression, then "smaller" = …
   - Prove: *Statement* for x, using IH (and known definitions, theorems …)
     - Usually, must show that going from x_{smaller} to larger x is true!

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

"smaller"

3. $\emptyset$,

Whole reg expr

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.

Thm: Reverse is Closed for Regular Langs

Example string: $\mathtt{abc}^{\mathcal{R}} = \mathtt{cba}$

For any string $w = w_1 w_2 \cdots w_n$, the ***reverse*** of w, written $w^{\mathcal{R}}$, is the string w in reverse order, $w_n \cdots w_2 w_1$.

For any language A, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} | \, w \in A\}$

Example language:

$\{\mathtt{a}, \mathtt{ab}, \mathtt{abc}\}^{\mathcal{R}} = \{\mathtt{a}, \mathtt{ba}, \mathtt{cba}\}$

Theorem: if A is regular, so is $A^{\mathcal{R}}$

Proof: by induction on the regular expression of A

Thm: Reverse is Closed for Regular Langs

if A is regular, so is $A^{\mathcal{R}}$

Proof: by Induction on regular expression of A: (6 cases)

Base cases

1. a for some a in the alphabet Σ, same reg. expr. represents $A^{\mathcal{R}}$ so it is regular
2. ε, same reg. expr. represents $A^{\mathcal{R}}$ so it is regular
3. $\emptyset$, same reg. expr. represents $A^{\mathcal{R}}$ so it is regular

Inductive 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.

Need to Prove: if A is a regular language, described by reg expr $R_1 \cup R_2$, then $A^{\mathcal{R}}$ is regular

IH1: if A_1 is a regular language, described by reg expr R_1, then $A_1{}^{\mathcal{R}}$ is regular

IH1: if A_2 is a regular language, described by reg expr R_2, then $A_2{}^{\mathcal{R}}$ is regular

"smaller"

Thm: Reverse is Closed for Regular Langs

if A is regular, so is $A^{\mathcal{R}}$

Proof: by Induction on regular expression of A: (Case # 4)

Statements	Justifications
1. Language A is regular, with reg expr $R_1 \cup R_2$	1. Assumption of IF in IF-THEN
2. R_1 and R_2 are regular expressions	2. Def of Regular Expression
3. R_1 and R_2 describe regular langs A_1 and A_2	3. Reg Expr ⇔ Reg Lang (Prev Thm)
4. If A_1 is a regular language, then $A_1{}^{\mathcal{R}}$ is regular	4. IH
5. If A_2 is a regular language, then $A_2{}^{\mathcal{R}}$ is regular	5. IH
6. $A_1{}^{\mathcal{R}}$ and $A_2{}^{\mathcal{R}}$ are regular	6. By (3), (4), and (5)
7. $A_1{}^{\mathcal{R}} \cup A_2{}^{\mathcal{R}}$ is regular	7. Union Closed for Reg Langs
8. $A_1{}^{\mathcal{R}} \cup A_2{}^{\mathcal{R}} = (A_1 \cup A_2)^{\mathcal{R}}$	8. Reverse and Union Ops Commute
9. $A = A_1 \cup A_2$	9. By (1), (2), and (3)
Goal 10. $A^{\mathcal{R}}$ is regular	10. By (7), (8), (9)

Thm: Reverse is Closed for Regular Langs

if A is regular, so is $A^{\mathcal{R}}$

Proof: by Induction on regular expression of A: (6 cases)

Base cases

☑ **1.** a for some a in the alphabet Σ,

☑ **2.** ε,

☑ **3.** $\emptyset$,

Inductive 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.

Remaining cases will use similar reasoning

Next Time

Non-Regular Languages?

- <u>Are</u> there **languages** that are **not regular languages?**

- <u>How</u> can we **prove** that a **language** is **not a regular language?**