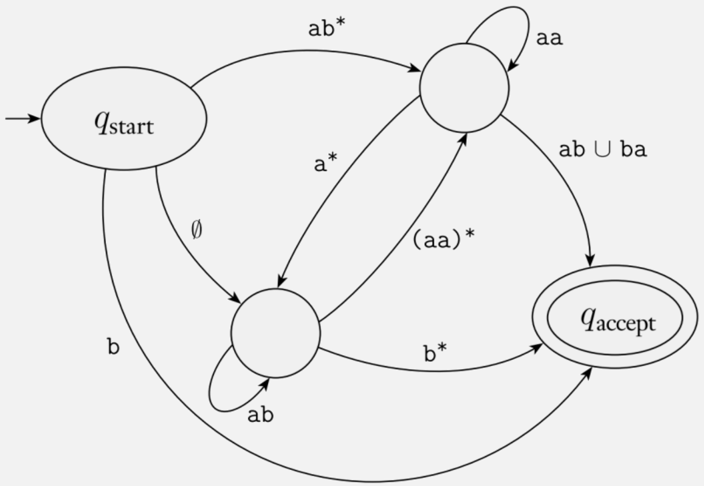
UMB CS 622

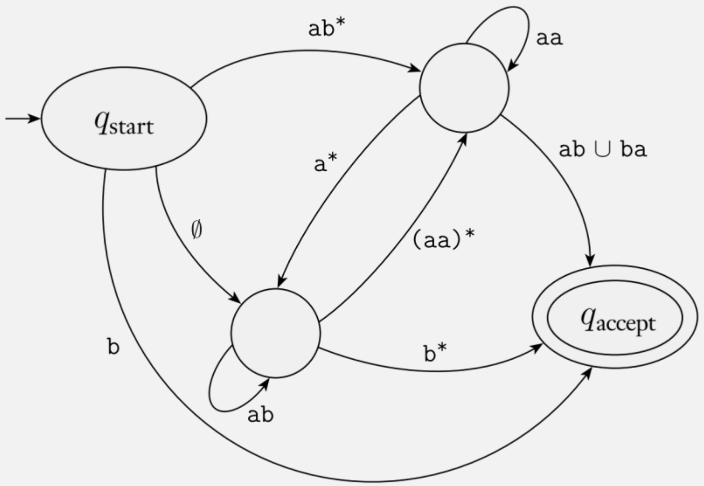
GNFA -> Regular Expression

Friday March 1, 2024



Announcements

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



Previously

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 it if:

- 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)

Previously

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)

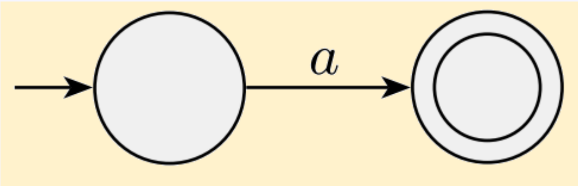
How to show that a language is regular?

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

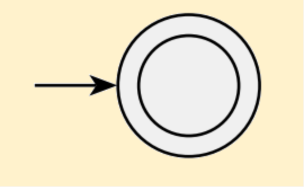
RegExpr→NFA

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

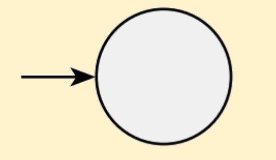
1. a for some a in the alphabet Σ,



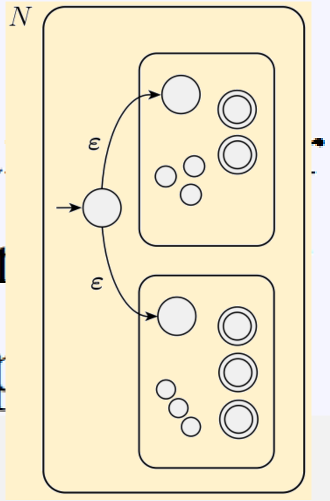
2. ε,

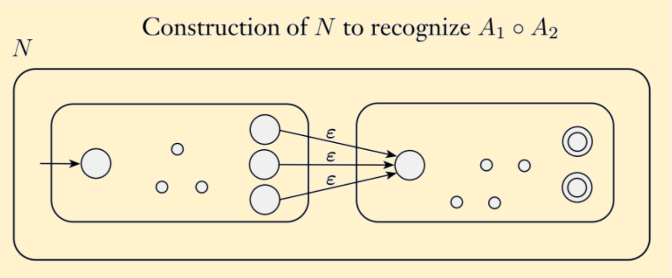


3. $\emptyset$,



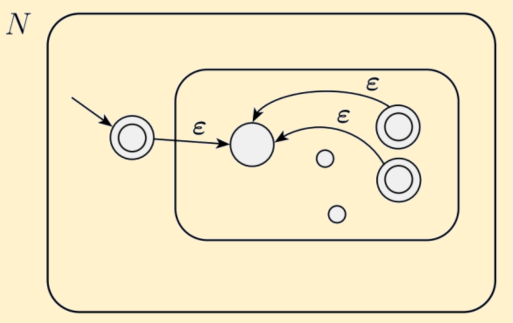
4. $(R_1 \cup R_2)$, where R_1 and R_2 a





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

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,** then **it's described by a reg expression**
(Harder)

GNFA

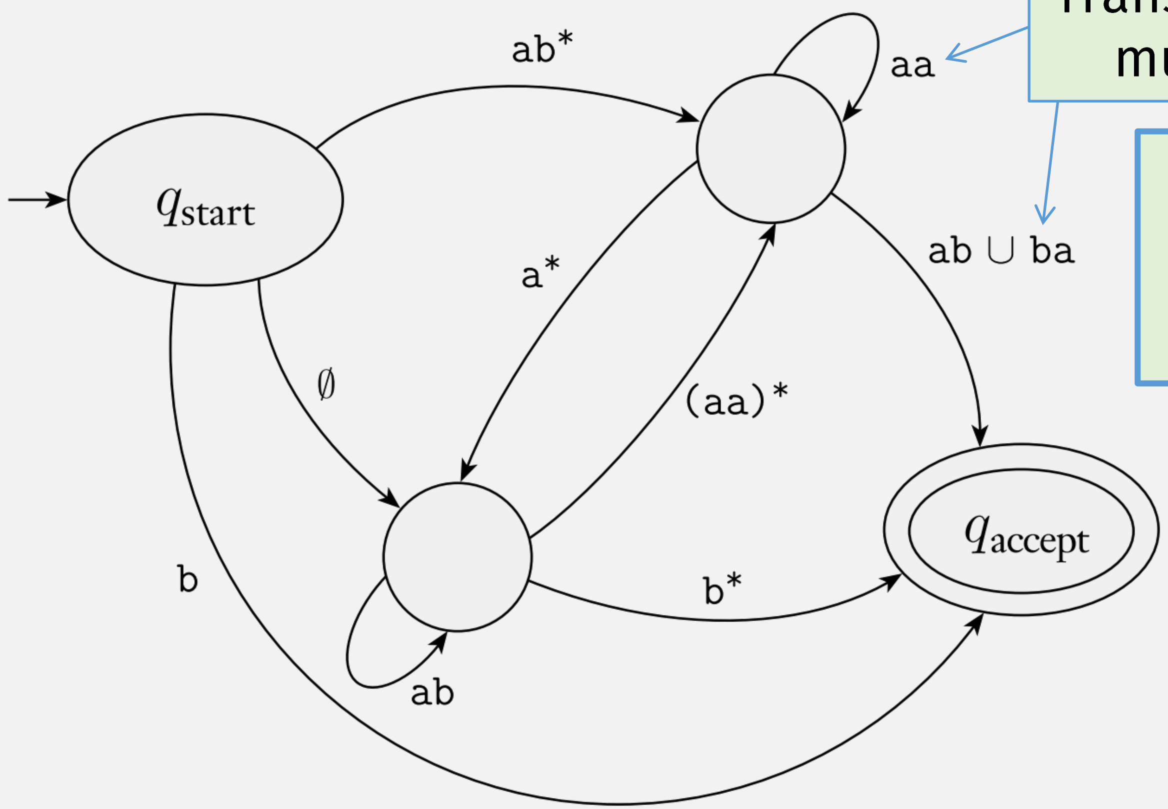
- Key step: **Convert an ~~DFA~~ or NFA → equivalent Regular Expression**
- First, **we need another kind of finite automata: a GNFA**

⇐ 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

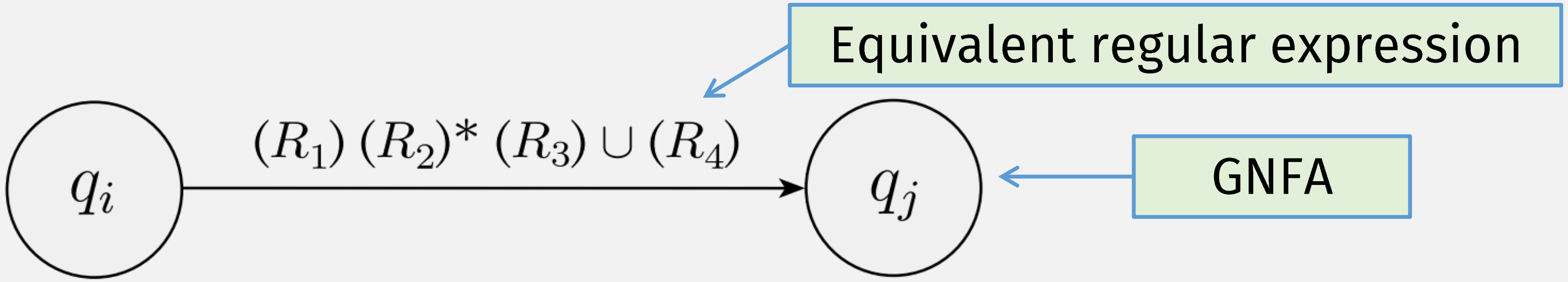
Goal: convert **GNFAs** to <u>equivalent</u> **Regular Exprs**

- GNFA = NFA with regular expression transitions

GNFA→RegExpr function

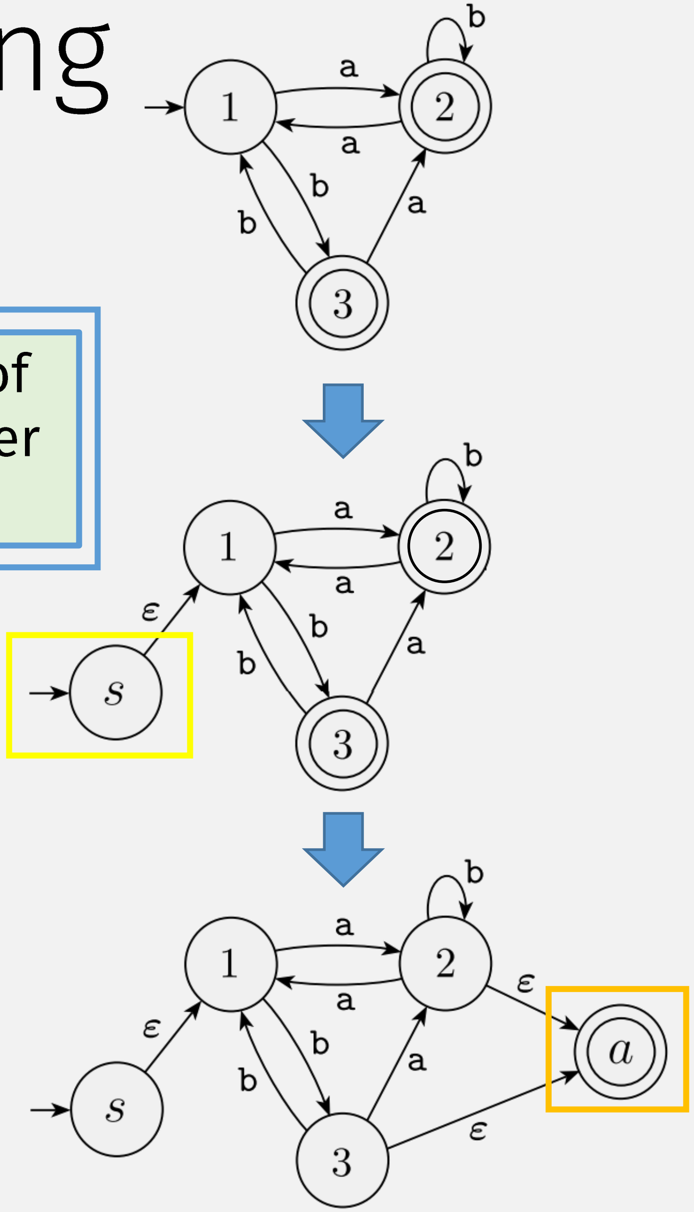
On **GNFA** <u>input</u> G:

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



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

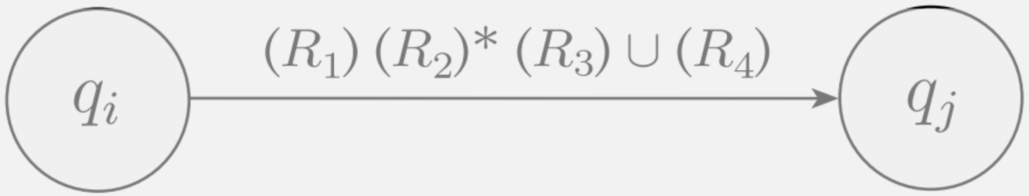
Modified machine always has 2+ states:
- <u>New start state</u>
- <u>New accept state</u>

GNFA→RegExpr function (recursive)

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

Base Case

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

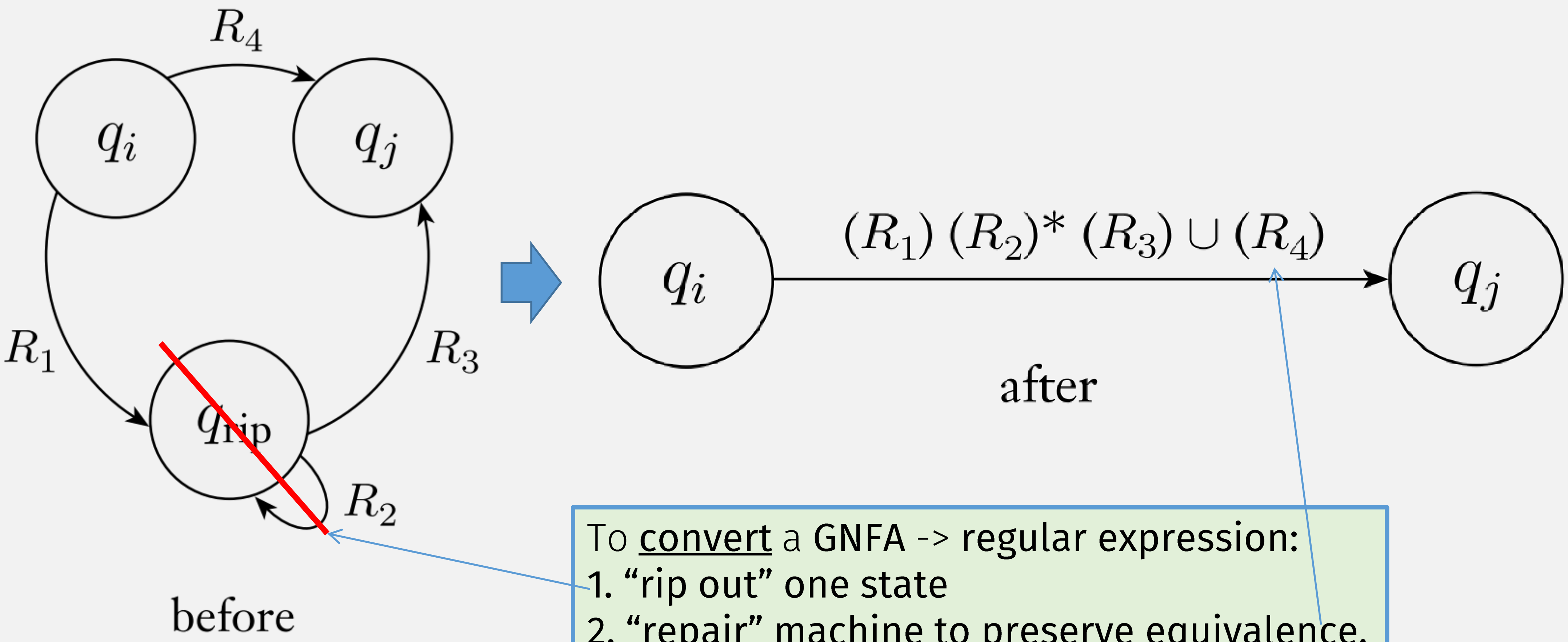


Recursive Case

- 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

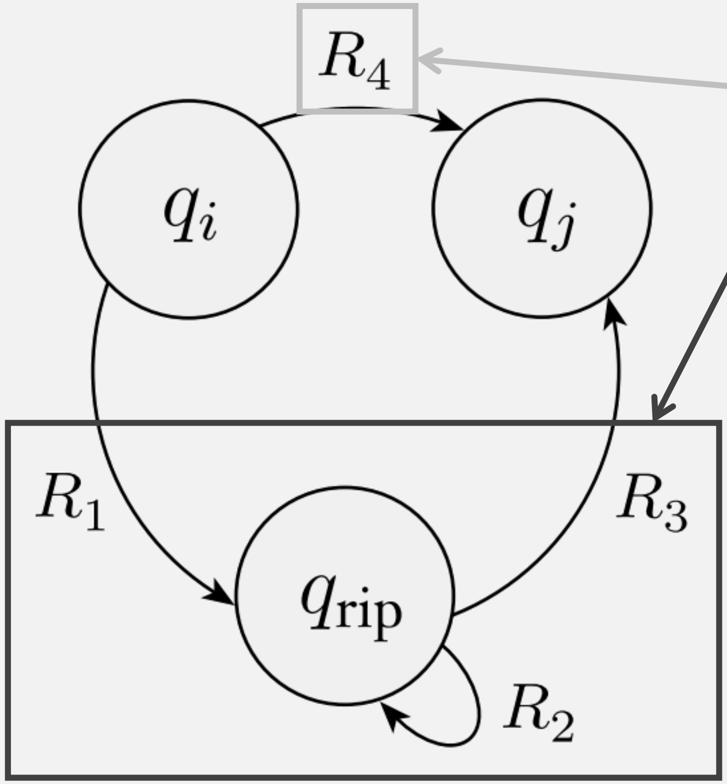


To convert a GNFA -> regular expression:
1. "rip out" one state
2. "repair" machine to preserve equivalence,
3. repeat until only 2 states remain

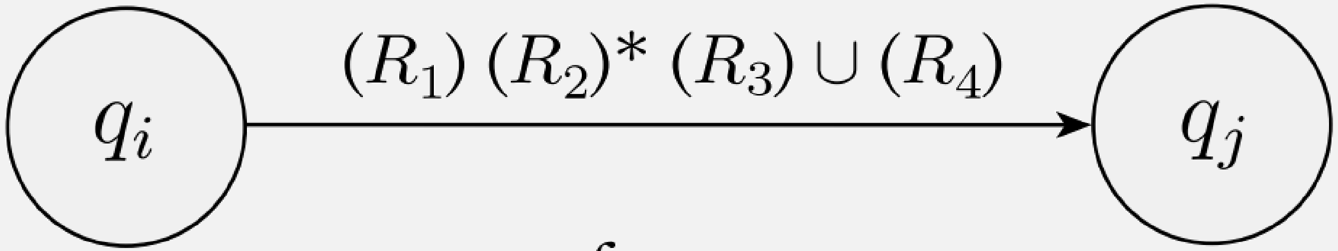
GNFA→RegExpr: “Rip/Repair” step

<u>Before</u>: two paths from q_i to q_j:

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



before

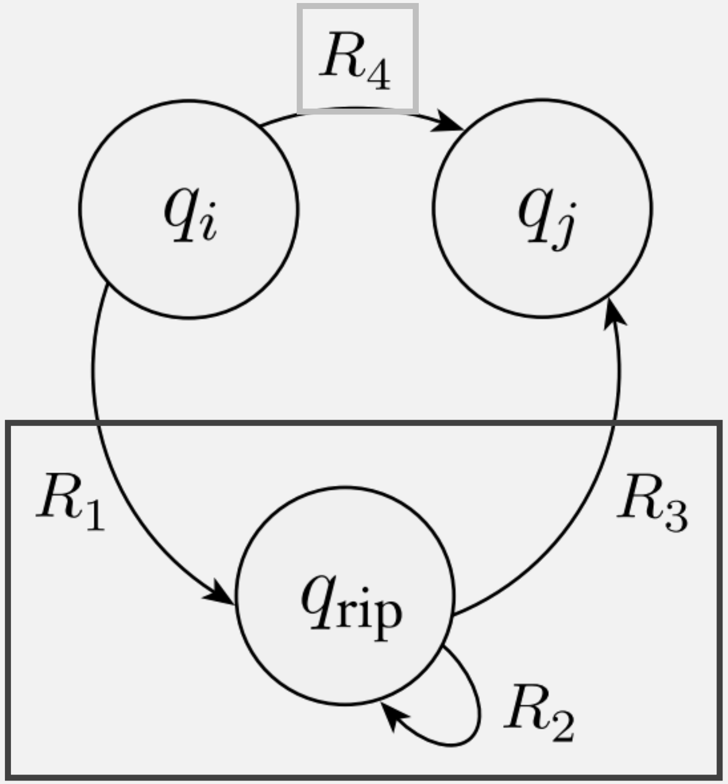


after

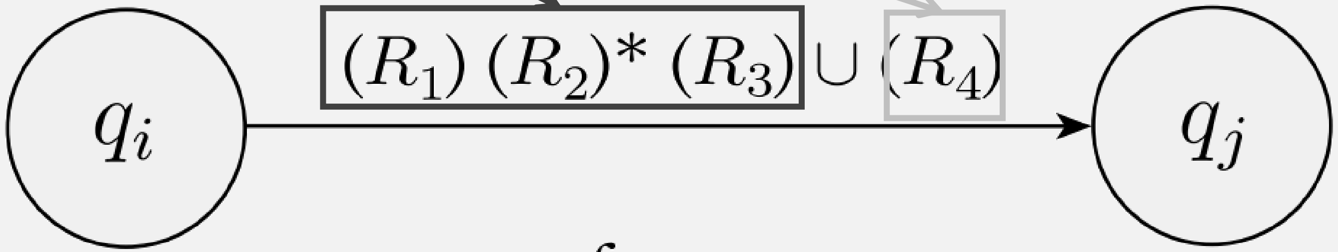
To <u>convert</u> a GNFA -> regular expression:
1. “rip out” one state
2. “repair” machine to preserve equivalence,
3. repeat until only 2 states remain

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}



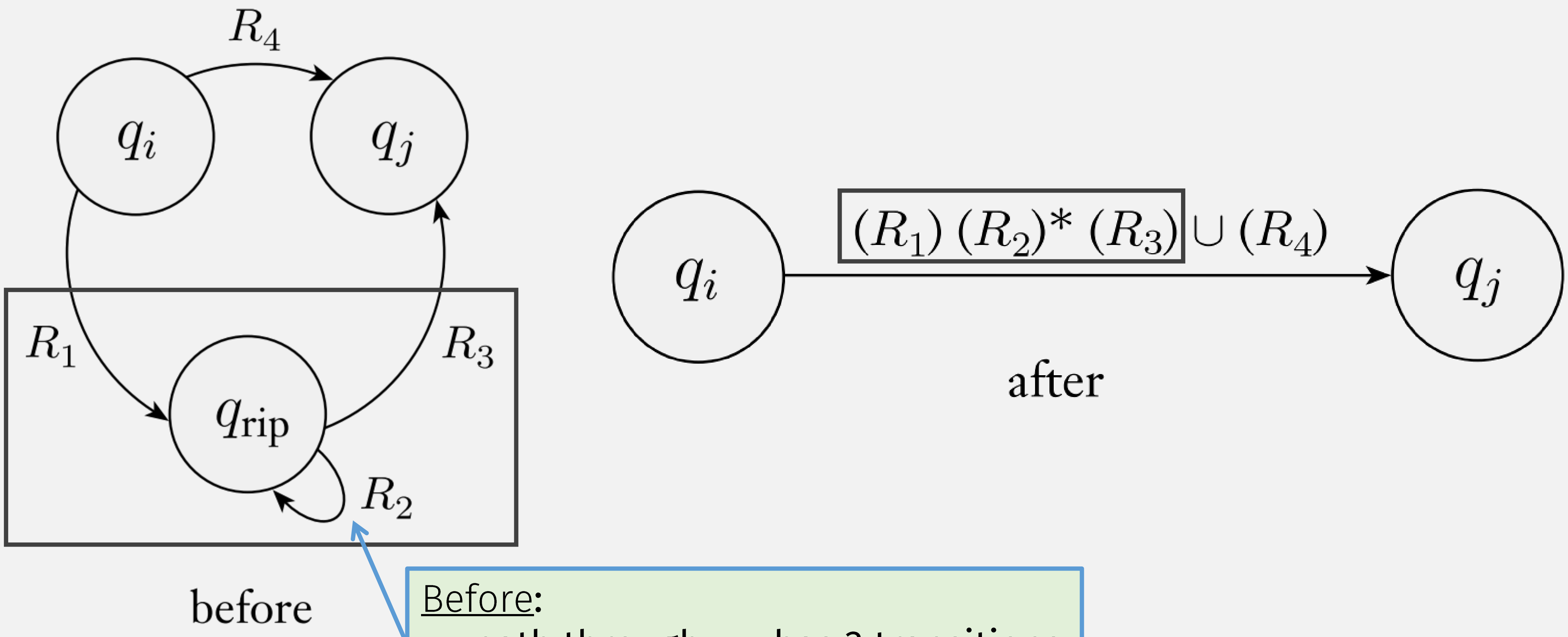
before



after

To <u>convert</u> a GNFA -> regular expression:
1. “rip out” one state
2. “repair” machine to preserve equivalence,
3. repeat until only 2 states remain

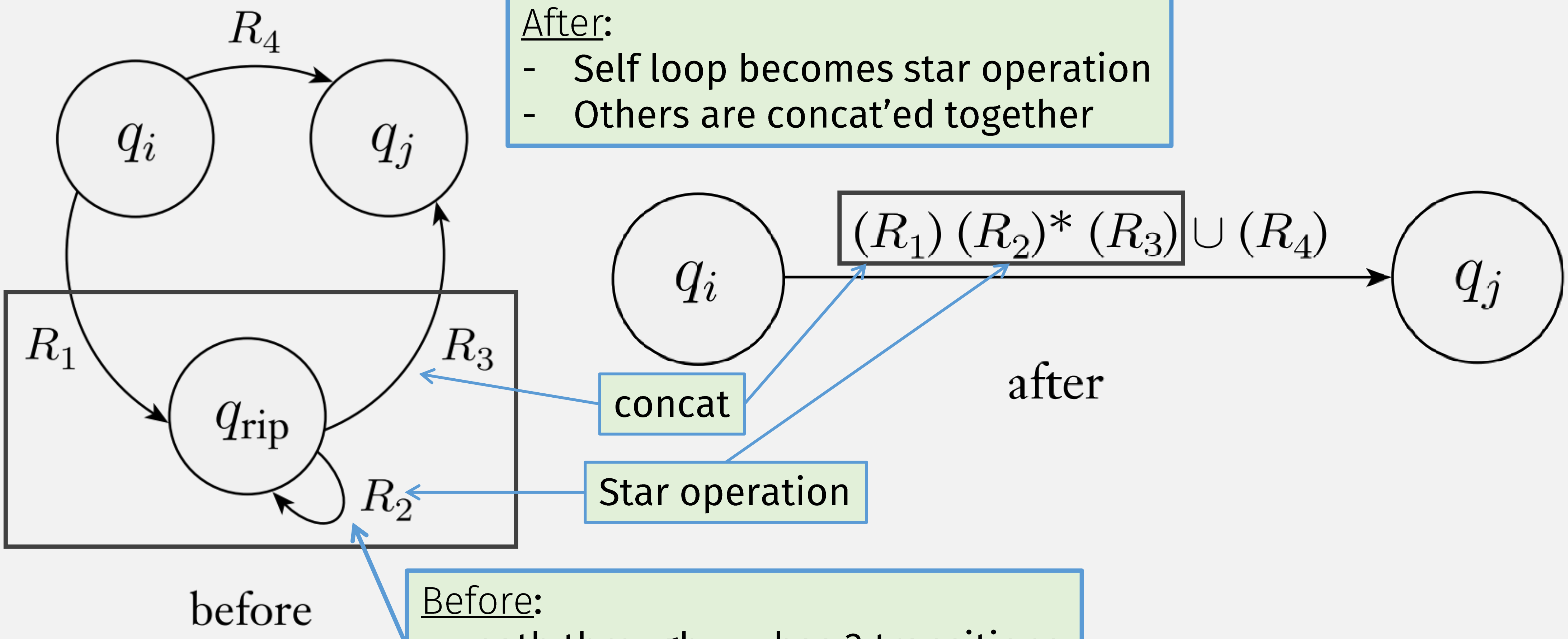
GNFA→RegExpr: "Rip/Repair" step



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 some examples)

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

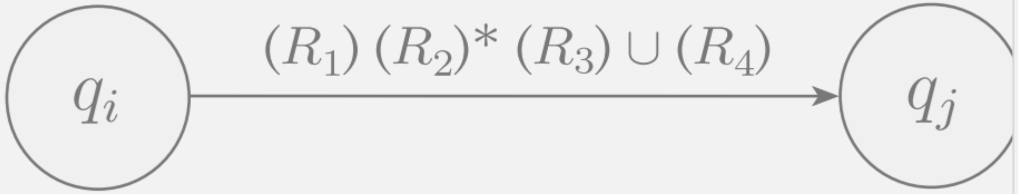
☑ • Convert regular expression → equiv NFA!

GNFA→RegExpr function (recursive)

On GNFA input G:

Base Case

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



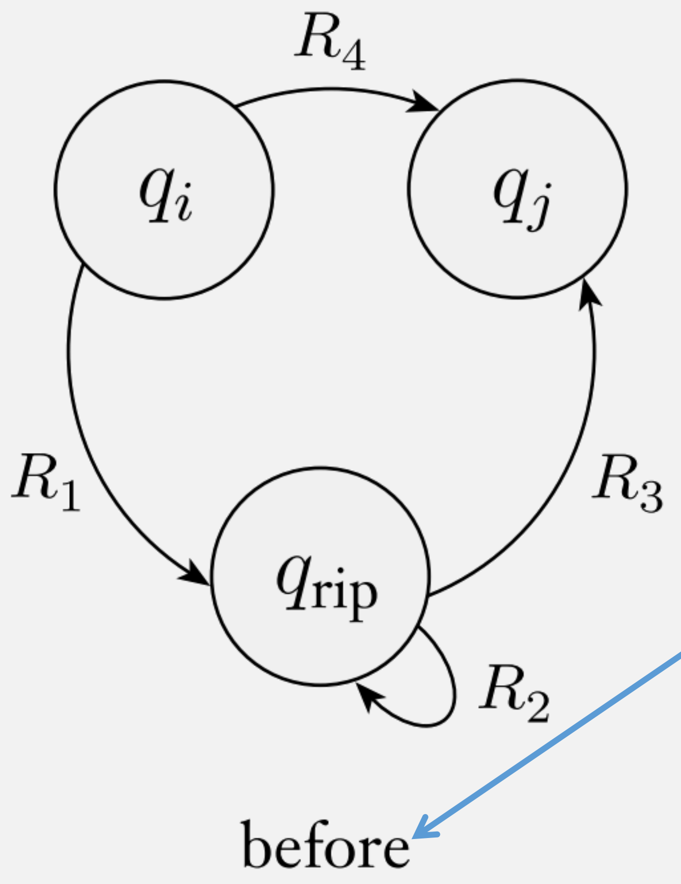
This time, let's really prove equivalence! (we previously "proved" it with some examples)

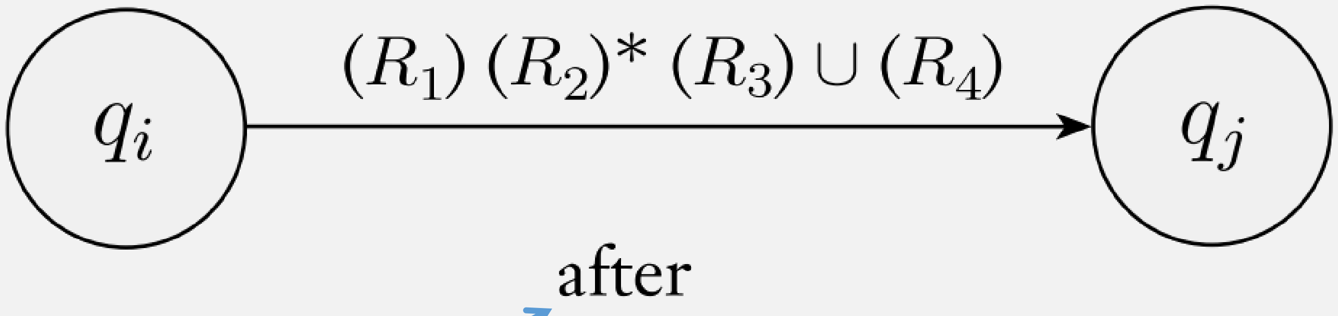
Recursive Case

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

First, show this step preserves equivalence

GNFA→RegExpr: Rip/Repair Correctness



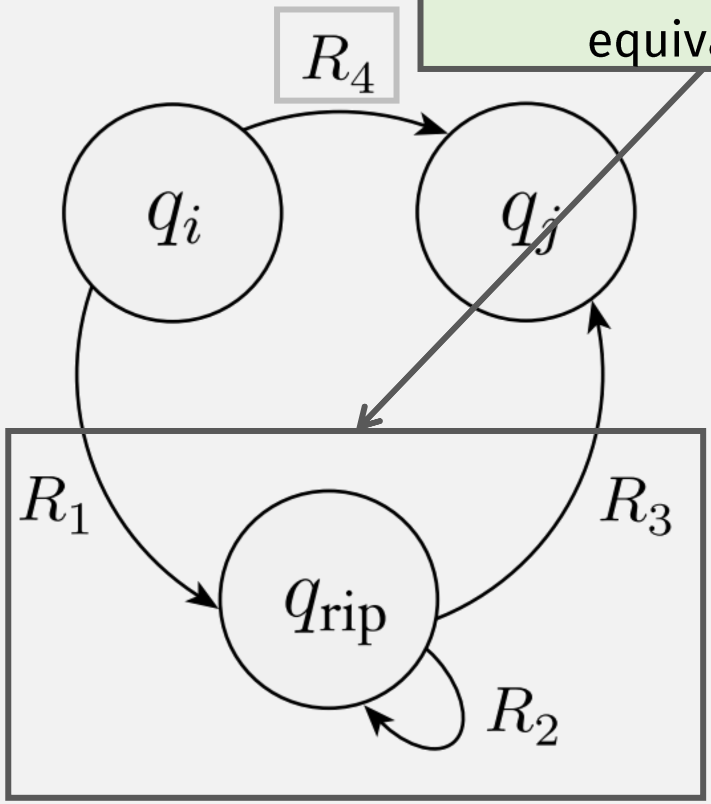


Must show these are <u>equivalent</u>

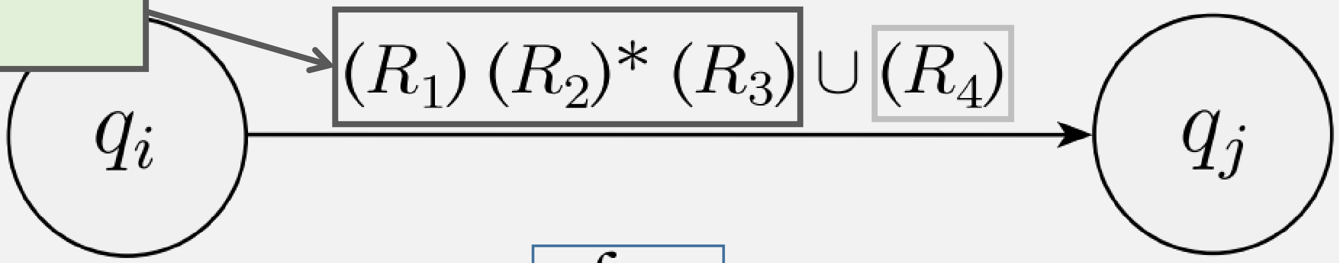
Equivalent =
same language =
accepts the same strings

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

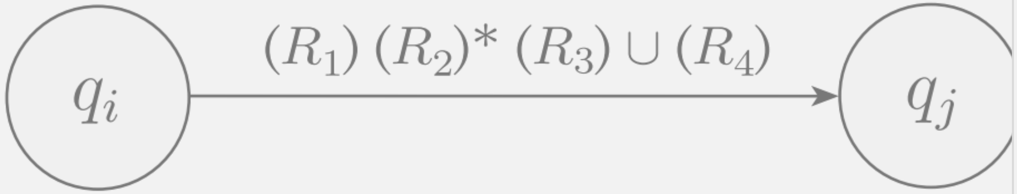
`GNFA→RegExpr` function (recursive)

Now we prove the whole function preserves equivalence

On GNFA input G:

Base Case

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



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

Recursive Case

First, show this step preserves equivalence ☑

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

GNFA→RegExpr Equivalence

- **Equivalent** = the **language does not change** (same 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)

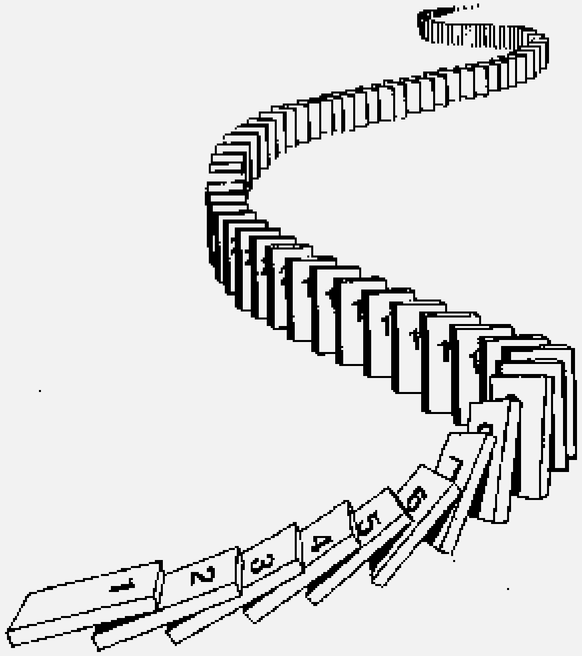
Language could be infinite set of strings!

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

Recursion!

Inductive Proofs

(Proofs involving recursion)



Kinds of Mathematical Proof

- **Deductive proof** (from before)
  - Start with: **assumptions, axioms,** and **definitions**
  - Prove: **news conclusions by making logical inferences** (e.g., **modus ponens**)

- **Proof by induction** (i.e., **"a proof involving recursion"**) (now)
  - Same as above …
  - But: **use this when proving something that is recursively defined**

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

Proof by Induction

(cases match a recursive definition)

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

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

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

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

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

A <u>valid recursive definition</u> has:
- **base case** and
- **recursive case** (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: $P_t = PM^t - Y\left(\frac{M^t - 1}{M - 1}\right)$

Proof: by **induction** on natural number t

An proof by induction exactly follows the recursive definition (here, natural numbers) that the induction is "on"

Base Case, $t = 0$:

- Goal: **Show $P_0 = P$** (amount owed at start = loan amount)
- Proof of Goal:

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

Plug in $t = 0$

Simplify, to get to goal statement

A Natural Number is:

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

Proof By Induction Example (Sipser Ch 0)

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

A Natural Number is:

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

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

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

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

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

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

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

- Proof of Goal:

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

"Connect together" known definitions and statements

Plug in IH for P_k

Simplify, to get to goal statement

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** exactly follows the **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 (number) the induction is "on"

Proof by Induction: CS 622 Example

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

Condition for **GNFA→RegExpr** function to be "correct", i.e., the languages must be equivalent

- 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

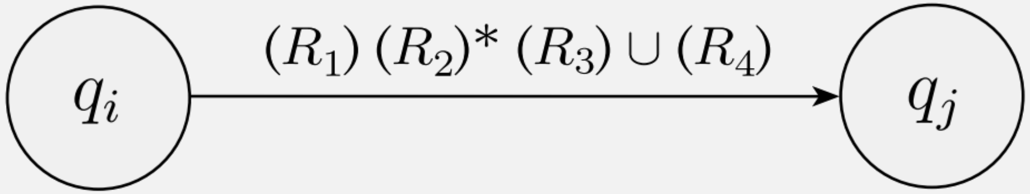
Now we are really proving equivalence!
(previously, we "proved" equivalence with a table of examples)

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

Proof by Induction: CS 622 Example

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

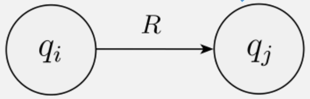
Recursively defined "thing"

Plug in

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?

Statements	Justifications
1. LANGOF ($q_i \xrightarrow{R} q_j$) = LANGOF (R)	1. Definition of GNFA
2. GNFA→RegExpr($q_i \xrightarrow{R} q_j$) = R	2. Definition of GNFA→RegExpr
Goal: LANGOF ($q_i \xrightarrow{R} q_j$) = LANGOF (GNFA→RegExpr($q_i \xrightarrow{R} q_j$))	3. From (1) and (2)

Plug in R

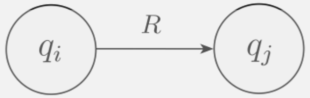
Don't forget to write out Statements / Justifications !

Proof by Induction: CS 622 Example

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

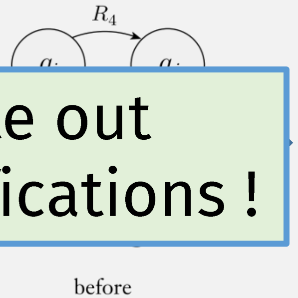
Proof: by Induction on # of states in *G*

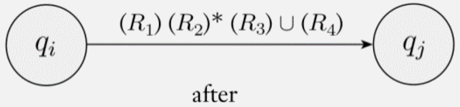
☑ 1. Prove *Statement* is true for base case — *G* has 2 states



2. Prove *Statement* is true for recursive case: *G* has > 2 states
   - Assume the **induction hypothesis** (IH):
     - *Statement* is true for smaller *G'*
   - Use it to prove *Statement* is true for larger *G*
     - Show that going from *G* to *G'* preserves *Statement*

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





Don't forget to write out Statements / Justifications !

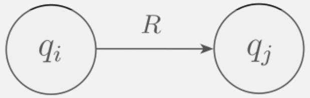
Show that "rip/repair" step converts *G* to smaller, equivalent *G'*

Proof by Induction: CS 622 Example

Statement to prove: LangOf (G) = LangOf (**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 **induction hypothesis** (IH):
  - *Statement* is true for smaller G'
- Use it to prove *Statement* is true for larger G
  - Show that going from G to G' preserves *Statement*

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. Correctness of Rip/Repair step (prev)
3. GNFA→RegExpr(G) = GNFA→RegExpr(G')	3. Def of GNFA→RegExpr
Goal 4. LangOf (G) = LangOf (GNFA→RegExpr(G))	4. From (1), (2), and (3)

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!

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.