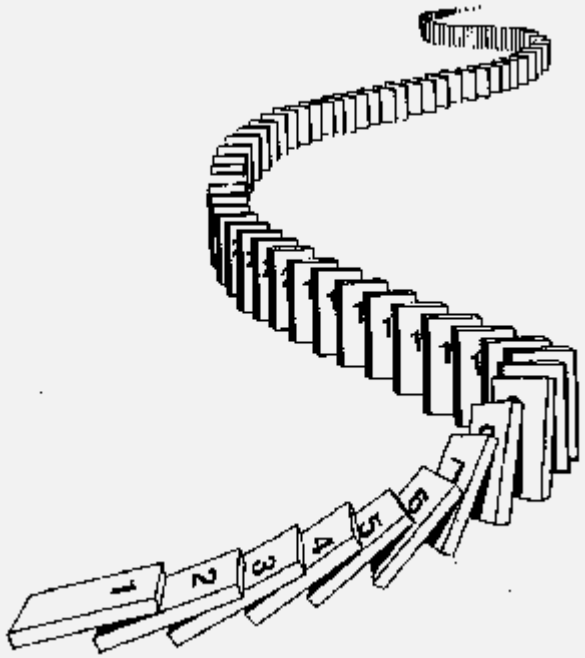
CS 420

Inductive Proofs

(recursive)

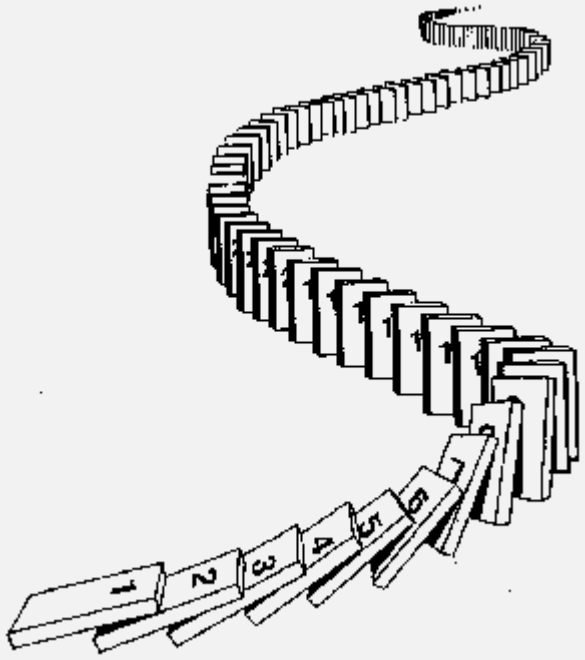
Thursday, October 15, 2026

UMass Boston Computer Science



Announcements

- HW 3
  - Grades out
- HW 4
  - ~~Due (extended): Thu 10/15 12:30pm~~
- HW 5
  - Out: Thu 10/15
  - Due: Thu 10/22 12:30pm



L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

(some) R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

(You should be able to sketch out the other steps of the full proof on your own)

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

Key step: convert regular expression → equivalent NFA!

How to show that a language is regular?

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

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

(sketch)

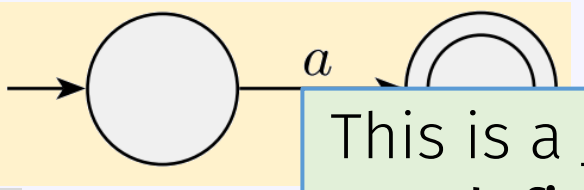
RegExpr→NFA

(this function must also be recursive!)

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

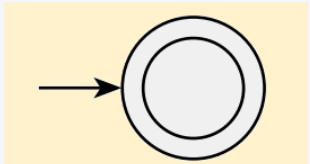
1. a for some a in
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and
5. $(R_1 \circ R_2)$, where R_1 and
6. (R_1^*),

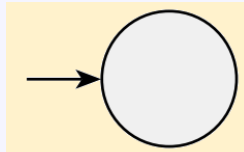
… then combine results of recursive calls

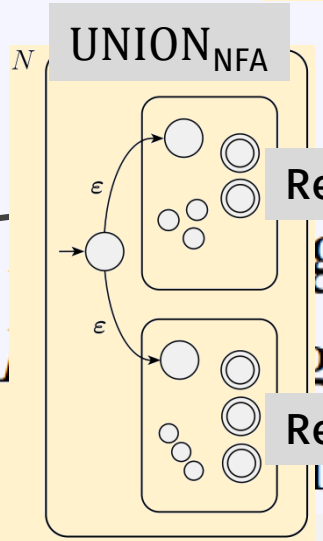


This is a recursive definition

Recursive calls on recursive parts …



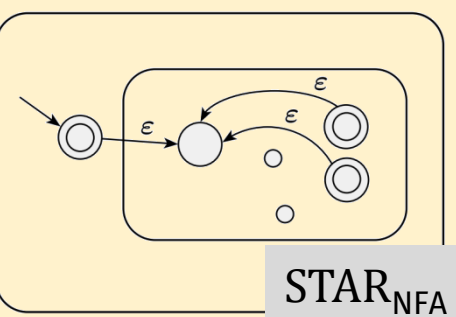




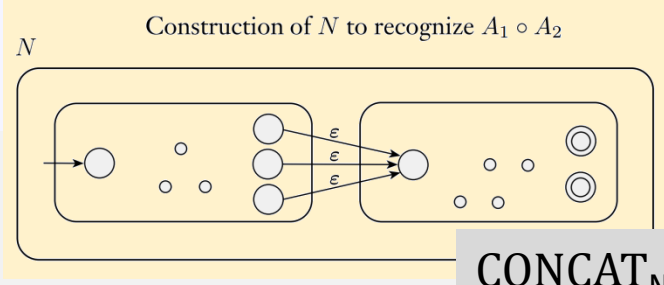
UNION$_{NFA}$

RegExpr→NFA(R_1)

RegExpr→NFA(R_2)



STAR$_{NFA}$



CONCAT$_{NFA}$

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

GNFA

Key step: Convert an ~~DFA~~ or NFA → equivalent Regular Expression

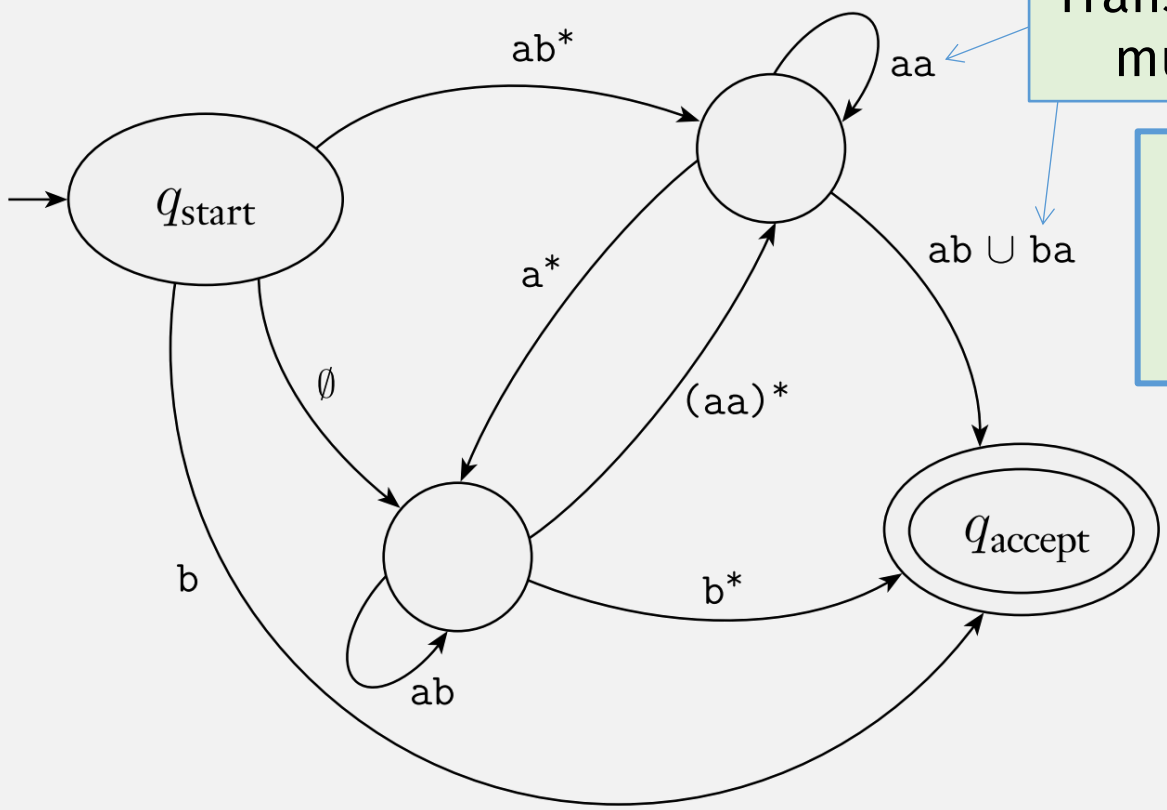
First, we need another kind of finite automata: a **GNFA**

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

☑ Key step: convert regular expression → equivalent NFA!

(You should be able to sketch out the other steps of the full proof on your own)

Generalized NFAs (GNFAs)



Transition can read multiple chars

plain NFA
= GNFA with single char regular expr transitions

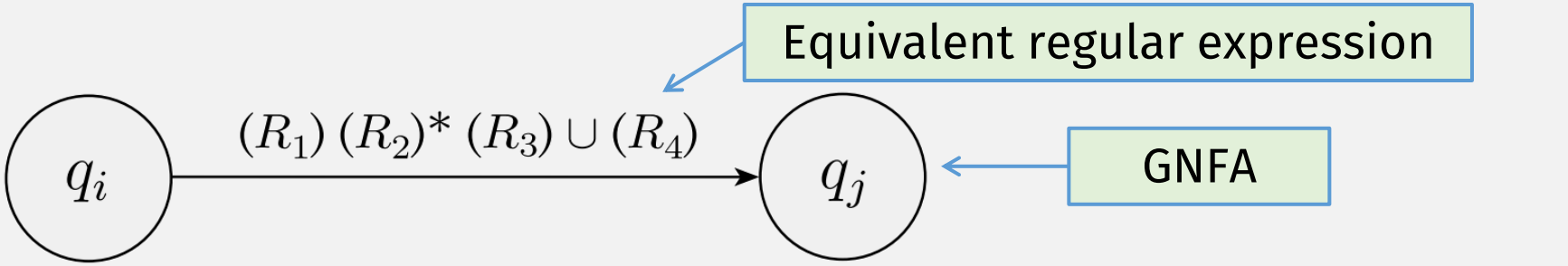
Goal: convert **GNFAs** to equivalent **Regular Exprs**

- GNFA = NFA with regular expression transitions

GNFA→RegExpr function : (sketch)

On **GNFA** input 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

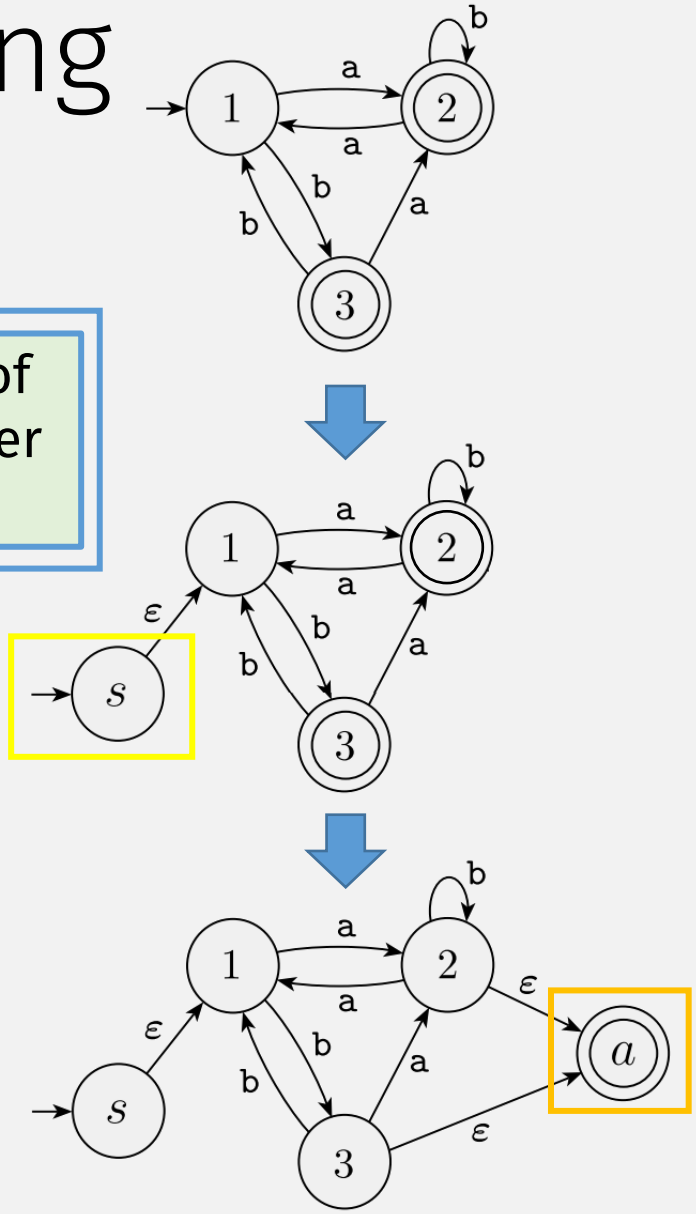
- Modify input machine to have:

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

- 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:
- New start state
- New accept state

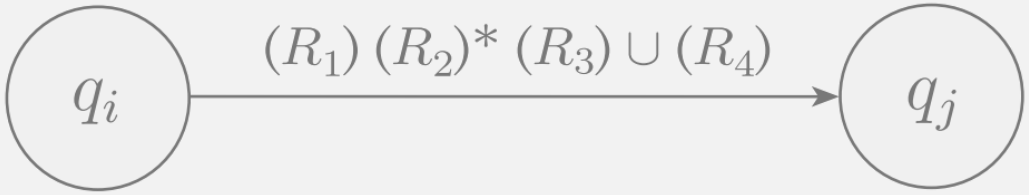


GNFA→RegExpr function (recursive)

On **GNFA** input G:

Base Case

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

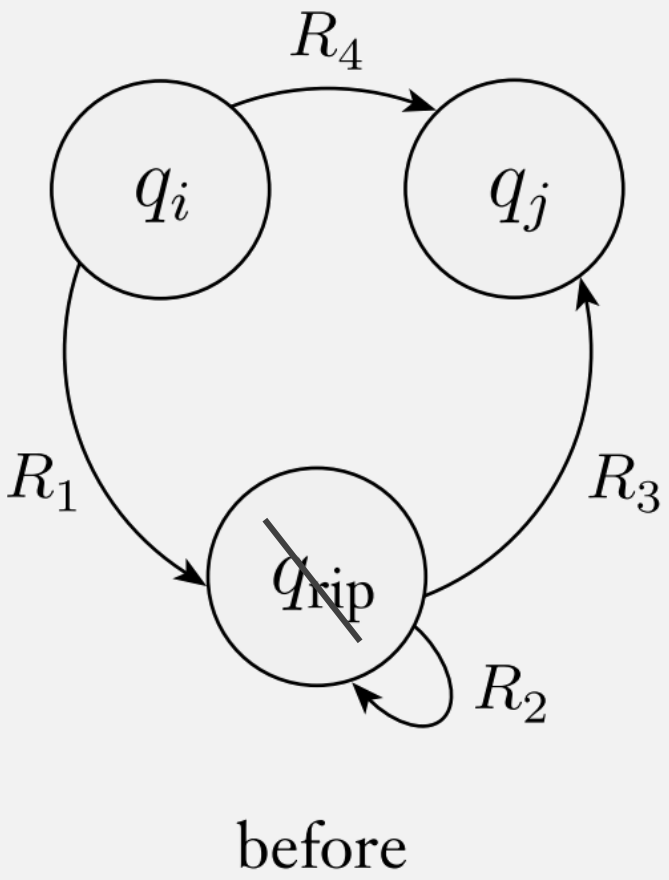


Recursive Case

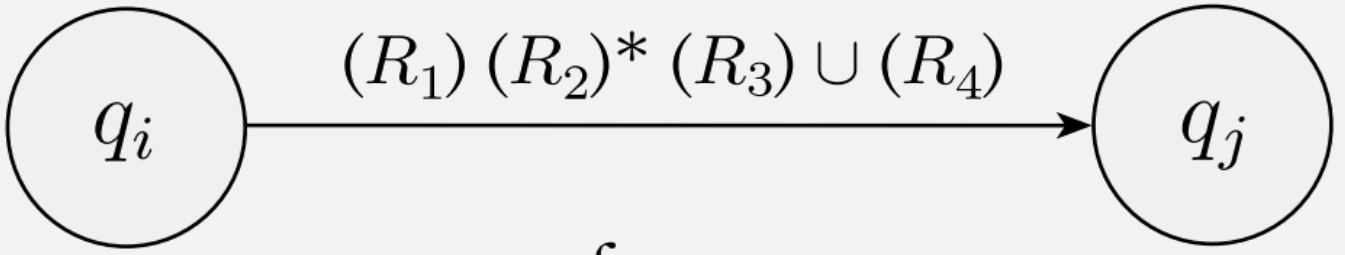
- Else:
  - "Rip out" one state
  - "Repair" the machine to get an equivalent GNFA G'
  - Recursively call **GNFA→RegExpr**(G')

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

GNFA→RegExpr: “Rip/Repair” step



before

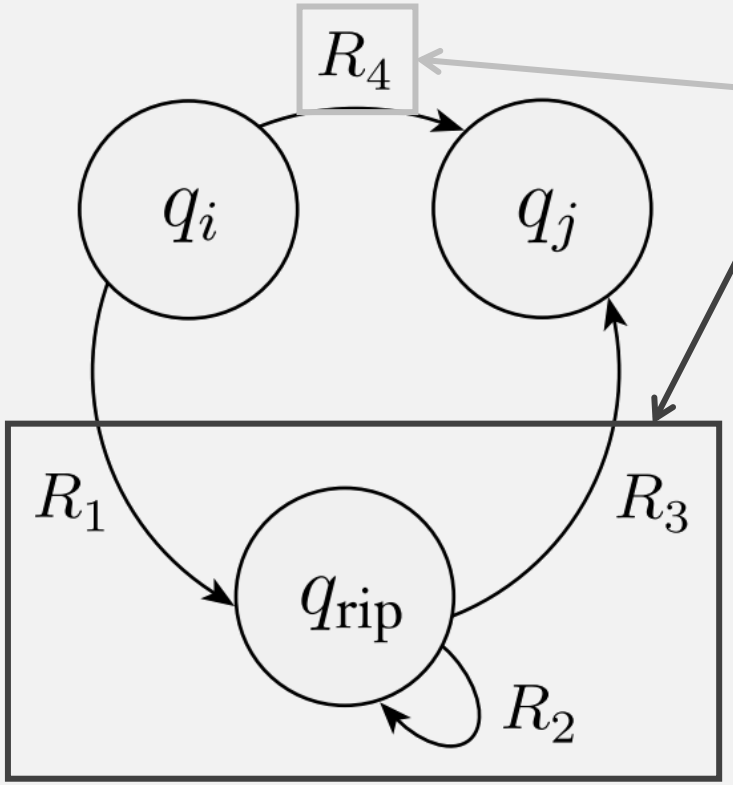


after

To convert GNFA -> regular expression:

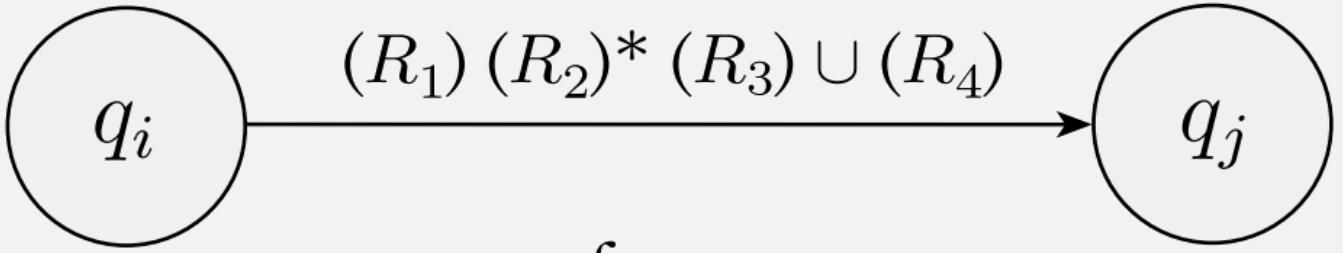
1. “rip out” a state,
2. “repair” so machine is equivalent
3. repeat until only 2 states remain

GNFA→RegExpr: "Rip/Repair" step



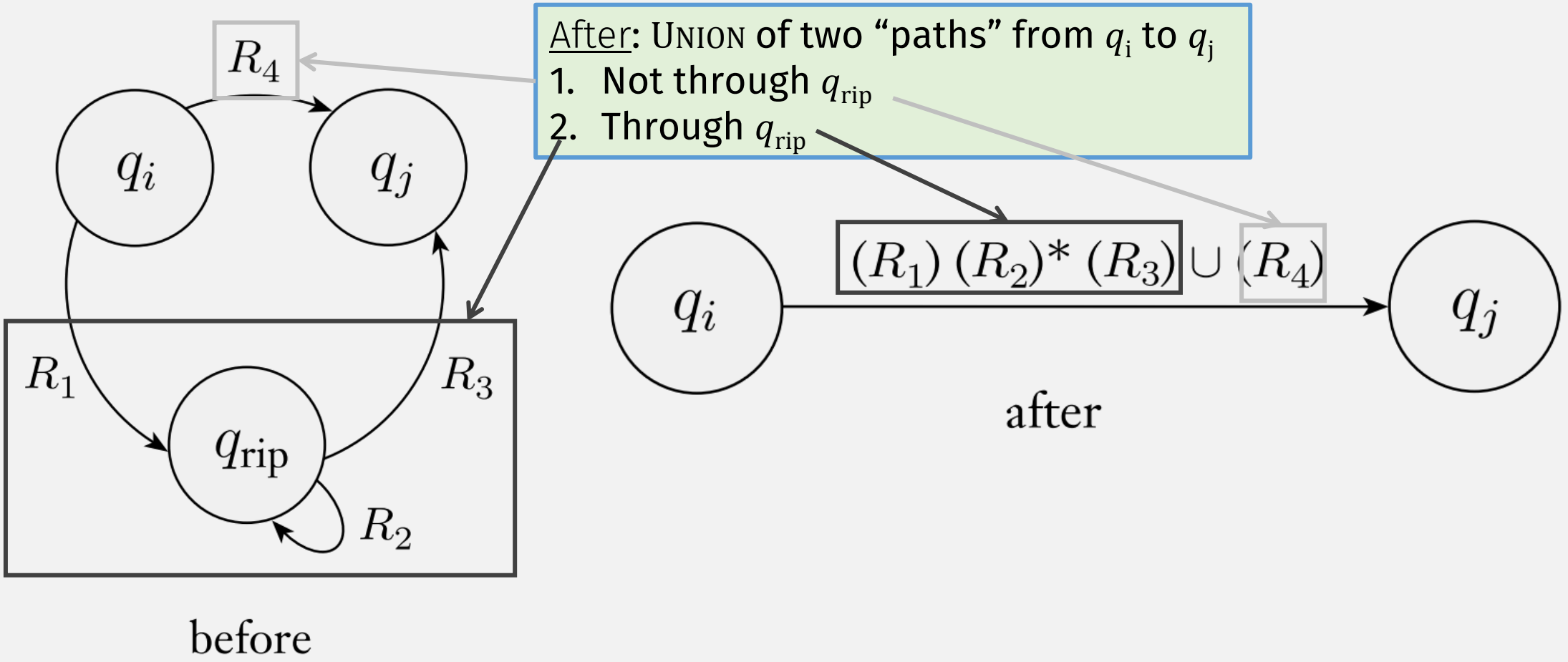
before

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

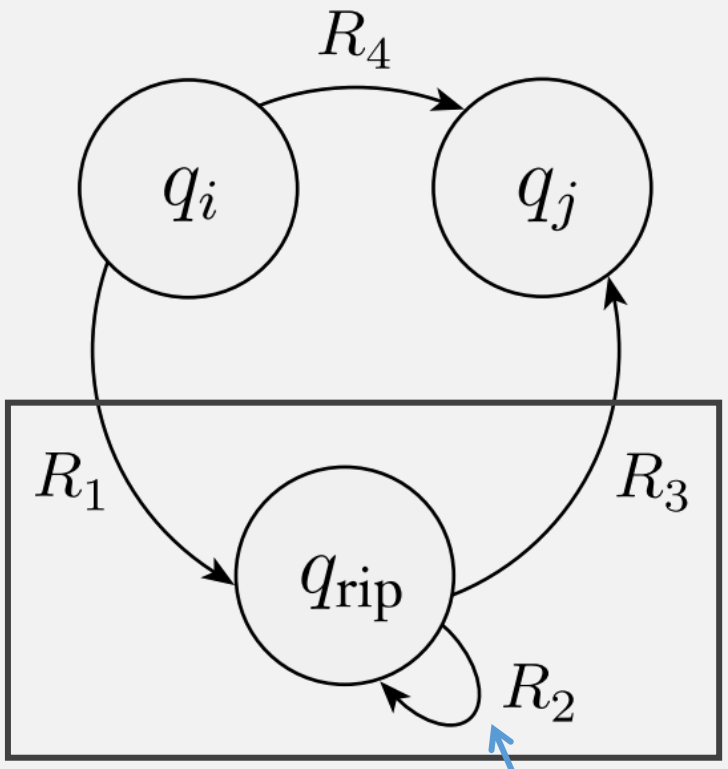


after

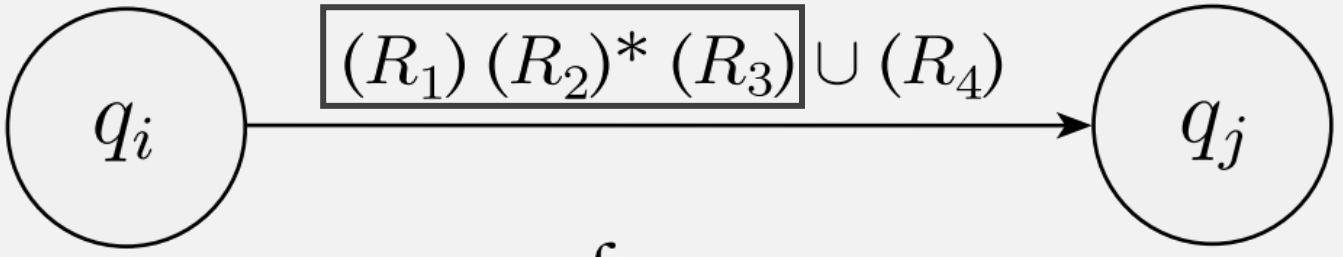
GNFA→RegExpr: "Rip/Repair" step



GNFA→RegExpr: “Rip/Repair” step



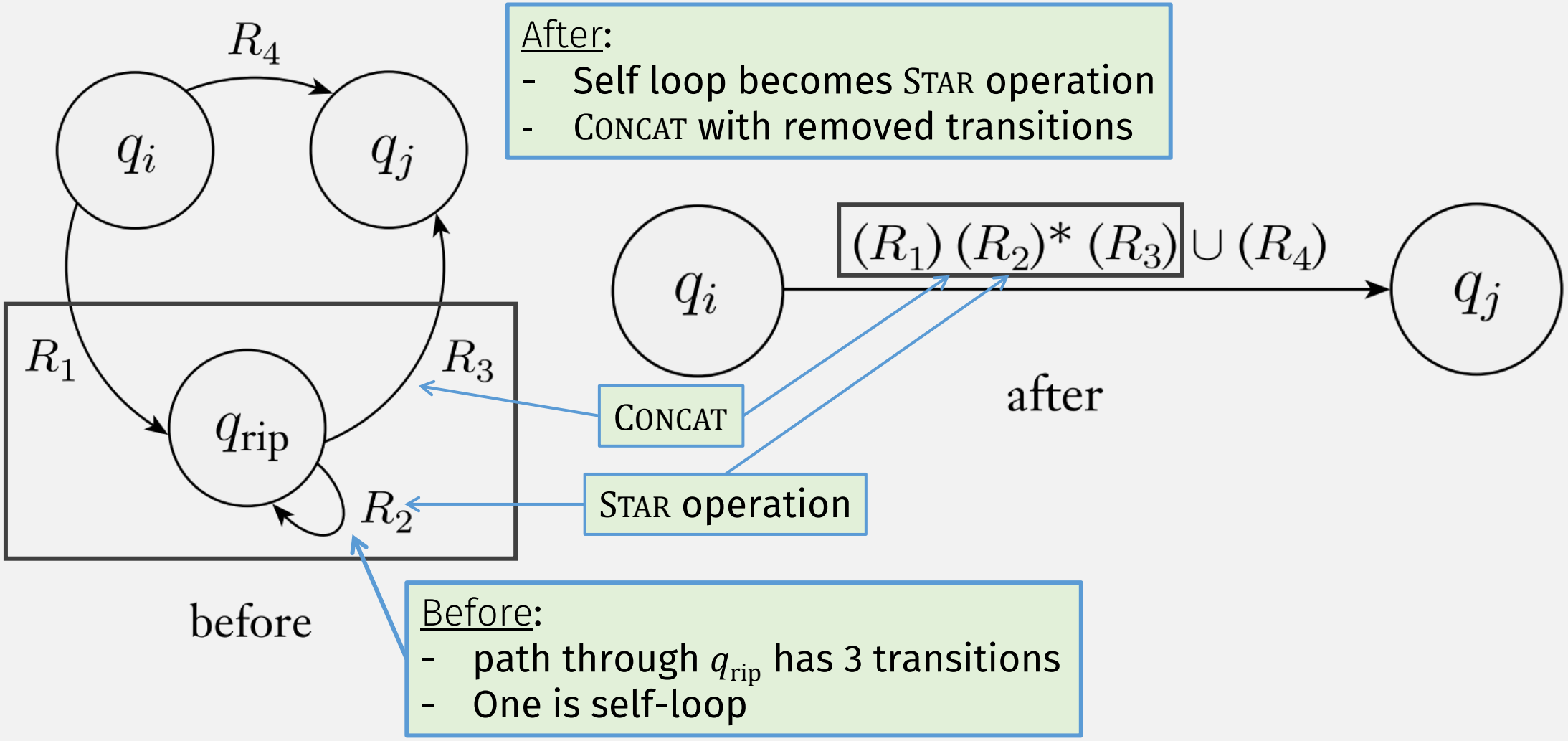
before



after

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

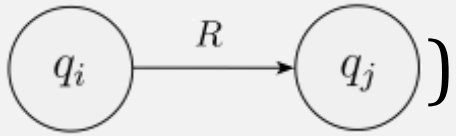
GNFA→RegExpr: "Rip/Repair" step



GNFA→RegExpr : GNFA → Reg Expr (more formally)

Base Case

(= 2 states)

GNFA→RegExpr (

) = R

$\text{Convert}_{\text{GNFA}}\text{BaseCase}$

Recursive Case

GNFA→RegExpr (G) = **GNFA→RegExpr** ("Rip/Repair" (G))

(> 2 states)

$\text{Convert}_{\text{GNFA}}\text{RecCase}$

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

Key step: Convert a GNFA → equivalent Regular Expression

☑ ???

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

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

☑ Key step: convert regular expression → equivalent NFA!

GNFA→RegExpr Correctness

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

Statement to Prove:

???

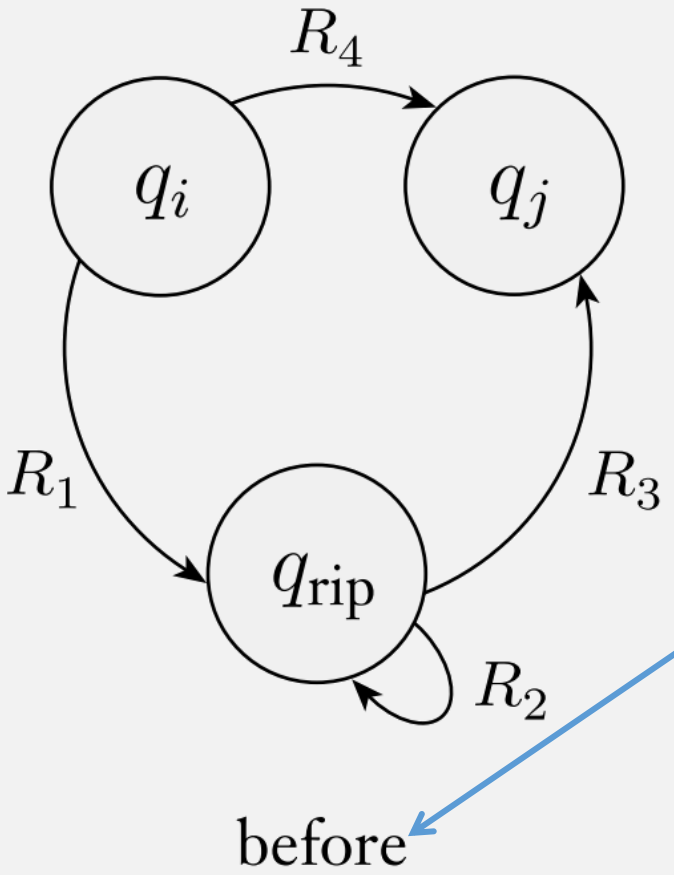
$$\text{LANGOF}(G) = \text{LANGOF}(R)$$

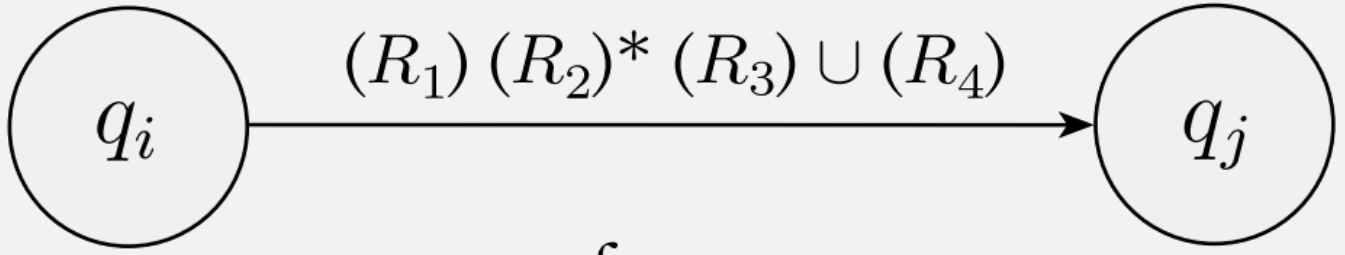
We are ready to **really prove equivalence!**
(we previously "proved" it with some examples)

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

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

GNFA→RegExpr: Rip/Repair Correctness





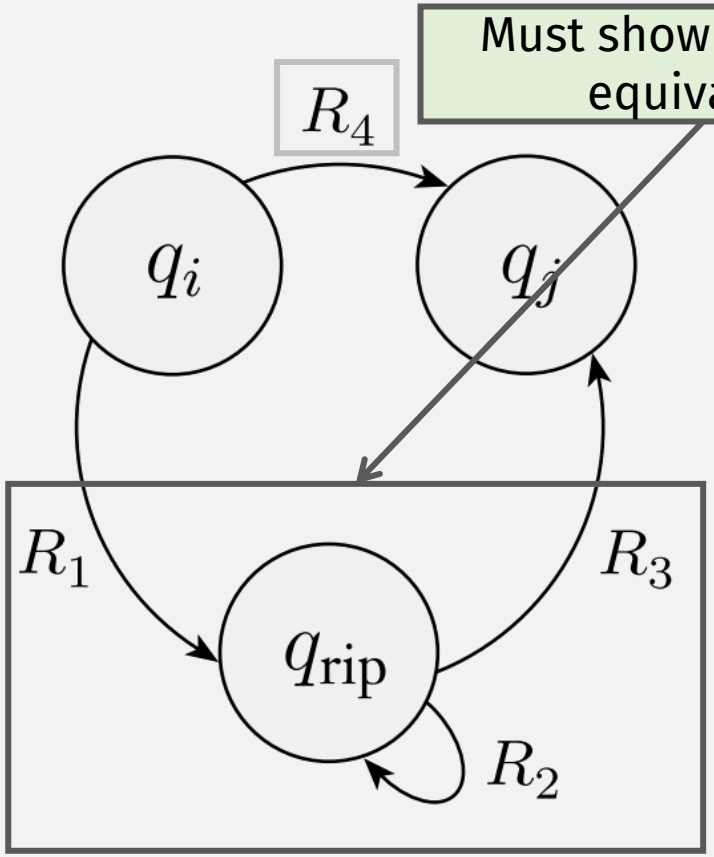
after

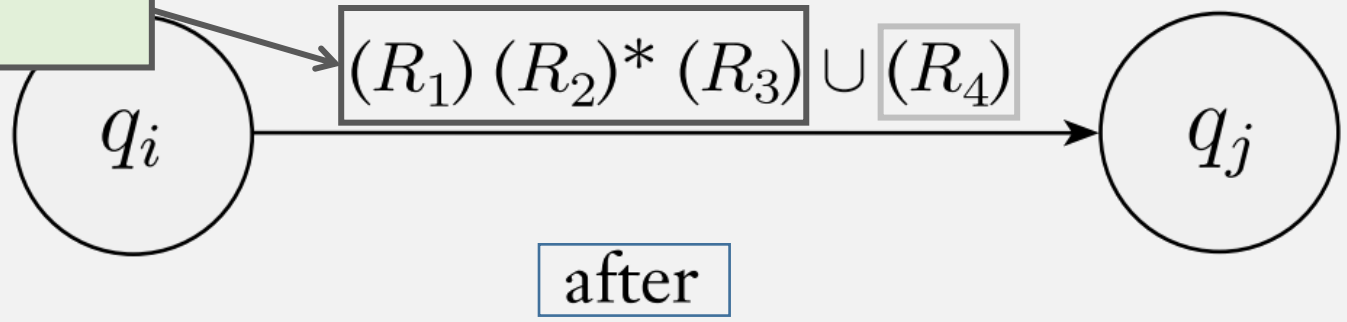
Must show these are equivalent

Equivalent =
same language =
accepts the same strings

RipRepairCorrect $\text{LANGOF}(G) = \text{LANGOF}(\text{Rip/Repair}(G))$

GNFA→RegExpr: Rip/Repair Correctness





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

(Similar to non-concrete Examples Table!)

GNFA→RegExpr Equivalence

- **Equivalent** = the **language does not change** (i.e., same set of 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 show **equivalence** for a possibly **infinite set of strings**?)

Recursion!

Proof by Induction

Previously:

Recursive function

- Use it when: defining 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 $k + 1$, where k is a **Natural Number**

Self-reference

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

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

A valid recursive definition 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)

Where:
amt owed in month t+1 =
amt owed in month t * interest M – amt paid Y

Loan_0 $P_0 = P$ (owed at start = amt borrowed)

Loan_{t+1} $P_{t+1} = P_tM - Y$

Proof By Induction Example (Sipser Ch 0)

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

Inductive proofs must have a name!

let LoanProof = mk-Inductive-proof (on t IS-A Natural Number)

(proof of)

A proof by induction follows the cases of the recursive definition that the induction is "on"

Need 2 subproofs, one for each case!

Use data def that proof is "on" to write "main" proof template

t = ...	**Statements**	**Justifications**
$t = 0$	$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right)$	BaseCaseProof (todo!)
$t = k + 1$	$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right)$	**APPLY** IndCaseProof **TO** … LoanProof(k) (proof of stmt for $t = k$)

Plug in $t = 0$

Plug in $t = k+1$

A Natural Number is:

- 0
- Or $k + 1$, where k is a Natural Number

Recursion in data def = recursion in proof!

let BaseCaseProof =

Statements

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

(Statement to Prove)

Justifications

1. **BY** REFL (= is **reflexive**, meaning x = x for all x)
2. **APPLY** SIMPL **TO** Proof1 (simplify)
3. **SUBST** Loan_0 **INTO** Proof2

Loan_0: $P_0 = P$



Proof By Induction Example (Sipser Ch 0)

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

let LoanProof = **mk-Inductive-proof** (**on** t IS-A Natural Number)

t = ...	**Statements**	**Justifications**
$t = 0$	$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right)$	BaseCaseProof ☑
$t = k + 1$	$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right)$	**APPLY** IndCaseProof **TO** ... LoanProof(k) (todo!)

A Natural Number is:

- 0 ☑
- Or $k + 1$, where k is a Natural Number

let IndCaseProof = **mk-Inductive-Case-proof** (**with** IH = proof of $P_k = PM^k - Y\left(\frac{M^k - 1}{M - 1}\right)$):

Plug in k

Statements

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

(simplify)

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

(Statement to Prove)

Justifications

1. SUBST IH INTO LOAN_{t+1}

Loan_{t+1}: $P_{k+1} = P_kM - Y$

2. **APPLY** SIMPL **TO** Proof1



Proof By Induction Example (Sipser Ch 0)

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

let LoanProof = **mk-Inductive-proof** (**on** t IS-A Natural Number)

t = ...	**Statements**	**Justifications**
$t = 0$	$P_0 = PM^0 - Y\left(\frac{M^0 - 1}{M - 1}\right)$	BaseCaseProof ☑
$t = k + 1$	$P_{k+1} = PM^{k+1} - Y\left(\frac{M^{k+1} - 1}{M - 1}\right)$	**APPLY** IndCaseProof **TO** LoanProof(k) ☑

A Natural Number is:

- 0 ☑
- Or $k + 1$, where k is a Natural Number ☑

Submit in gradescope

Lecture Supplement: 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: $(z \neq 1)$

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

Hints:

- **Must choose which value** the **induction is "on"** (could be more than one option)
- **Use data def of "on" value** to **write recursive proof template**
  - E.g., same number of cases
  - Recursive on the same part
- **Use proper proof constructors** `mk-Inductive-proof`
- **Recursive proof must have name** (makes self-reference possible)
- **Subproofs must have proper names** and **also use proper constructors** (if necessary)

`mk-Inductive-Case-proof`

Proof by Induction: CS 420 Example

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

- 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"

let GNFAProof = **mk-Inductive-proof** (**on** # of states in G IS-A NatNum>1): (Proof of)

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

Statements	Justifications
G has 2 states: LANGOF ($q_i \xrightarrow{R} q_j$) = LANGOF (**GNFA→RegExpr**($q_i \xrightarrow{R} q_j$))	GNFABaseCaseProof Todo!
G has k+1 states: LANGOF (G) = LANGOF (**GNFA→RegExpr**(G))	**APPLY** GNFAIndCaseProof **TO** GNFAProof(G') where G' IS-A **GNFA** with k states

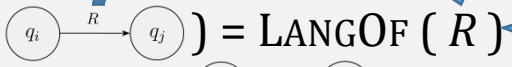
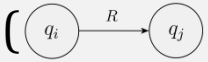
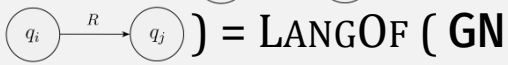
(Modified) Recursive definition:

A "NatNum>1" is:

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

Recursion in data def = recursion in proof!

let GNFABaseCaseProof =

Statements	**Justifications**
1. LangOf () = LangOf (R)	**1. BY** DefGNFACompute
2. **GNFA→RegExpr**() = R	2. Convert$_{\texttt{GNFA}}$BaseCase
3. LangOf () = LangOf (**GNFA→RegExpr**())	**3. SUBST** Proof2 **INTO** Proof1

Plug in R

(stmt to prove)

Proof by Induction: CS 420 Example

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

let GNFAProof = **mk-Inductive-proof** (**on** # of states in G IS-A NatNum>1):

Statements

Justifications

G has 2 states

LANGOF () = LANGOF (**GNFA→RegExpr**( ))

GNFABaseCaseProof ☑

A "NatNum>1" is:

- 2 ☑
- Or $k + 1$, where k is a "NatNum>1"

G has k+1 states

LANGOF (G) = LANGOF (**GNFA→RegExpr**(G))

APPLY GNFAIndCaseProof **TO** GNFAProof(G')

Todo! where G' IS-A **GNFA** with k states

Where G' = Rip/Repair(G) --- is smaller than G

let GNFAIndCaseProof = **mk-Inductive-Case-proof** (**with** IH = proof of LangOf (G') = LangOf (**GNFA→RegExpr**(G'))):

Statements	**Justifications**
1. LangOf (G) = LangOf (G')	1. RipRepairCorrect
2. GNFA→RegExpr(G) = **GNFA→RegExpr**(G')	2. Convert$_{\text{GNFA}}$RecCase
3. LangOf (G) = LangOf (**GNFA→RegExpr**(G'))	**3. SUBST** Proof1 **INTO** IH
4. LangOf (G) = LangOf (**GNFA→RegExpr**(G))	**4. SUBST** Proof2 **INTO** Proof3

(stmt to prove)

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND R **describes** L

Need 2 proofs:

⇒ IF L IS-A **Regular** Language THEN R IS-A **Regular** Expression AND R **describes** L

☑ Key step: Convert a GNFA → equivalent Regular Expression

⇐ IF R IS-A **Regular** Expression AND R **describes** L THEN L IS-A **Regular** Language

☑ Key step: convert regular expression → equivalent NFA!

■

Now: we can use regular expressions to represent regular langs!

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

So a regular language has these equivalent representations:

- DFA
- NFA
- Regular Expression

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.

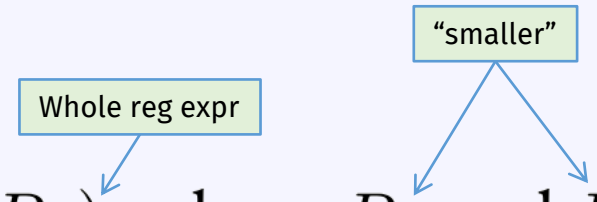
Proof by Induction

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,
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.



Thm: for Regular Langs Closed Under Reverse

Example string: rev **abc** = **cba**

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

For any language A, let **rev** A = { **rev** w | $w \in A$ }

Example language:

rev { **a**, **ab**, **abc** } = { **a**, **ba**, **cba** }

Theorem:

IF A IS-A **Regular** Language THEN rev A IS-A **Regular** Language

Want to use: regular expression of A!

IF A IS-A **Regular** Language THEN rev(A) IS-A **Regular** Language

Assume IF part ...

`mk-IFTHEN-proof (with` ASSUMPTION = proof of A IS-A **Regular** Language):

Statements

1. R IS-A **Regular** Expression AND R **describes** A

   LangOf(R) = A

Equivalent thm

IF R IS-A **Regular** Expression THEN rev(LangOf(R)) IS-A **Regular** Language

10. rev(A) IS-A **Regular** Language

    rev(LangOf(R)) IS-A **Regular** Language

... prove THEN part

Justifications

1. **APPLY** RegLangIFFRegEx ⇒ **TO** ASSUMPTION

   L IS-A **Regular** Language — RegLangIFFRegEx

   IF AND ONLY IF

   R IS-A **Regular** Expression AND R **describes** L

   R **describes** L means LangOf(R) = L

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

An inductive proof on a regular expression must follow this recursive structure!

IF R IS-A **Regular** Expression THEN rev(LangOf(R)) IS-A **Regular** Language

Write "main" proof template first

let RecRevProof = **`mk-Inductive-proof`** (**on** proof of R IS-A **Regular** Expression):

	Statements	**Justifications**
$R = a$	1. IF a IS-A **Regular** Expression THEN rev(LangOf(a)) IS-A **Regular** Language	1. BaseCaseProof$_a$
$R = \varepsilon$	2. IF ε IS-A **Regular** Expression THEN rev(LangOf(ε)) IS-A **Regular** Language	2. BaseCaseProof$_\varepsilon$
$R = \emptyset$	3. IF $\emptyset$ IS-A **Regular** Expression THEN rev(LangOf($\emptyset$)) IS-A **Regular** Language	3. BaseCaseProof$_\emptyset$
$R = R_1 \cup R_2$	4. IF $R_1 \cup R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \cup R_2$)) IS-A **Regular** Language	4. APPLY RecCaseProof$_\cup$ TO RecRevProof(R_1), RecRevProof(R_2)
$R = R_1 \circ R_2$	5. IF $R_1 \circ R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \circ R_2$)) IS-A **Regular** Language	5. APPLY RecCaseProofo TO RecRevProof(R_1), RecRevProof(R_2)
$R = R_1^*$	6. IF R_1^* IS-A **Regular** Expression THEN rev(LangOf(R_1^*)) IS-A **Regular** Language	6. APPLY RecCaseProof* TO RecRevProof(R_1)

6 IFTHEN Proof TODOs!

Recursion in data def = recursion in proof!

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.

An inductive proof on a regular expression must follow this recursive structure!

IF a IS-A **Regular** Expression
THEN rev LangOf(a) IS-A **Regular** Language

let BaseCaseProof$_a$ =

IFTHEN proof!

mk-IFTHEN-proof (**with** ASSUMPTION = proof of a IS-A **Regular** Expression):

Statements	**Justifications**
1. LangOf(a) = {a}	1. DefRegExCompute1
2. rev {a} = {a}	2. DefRev
3. rev LangOf(a) = {a}	**3. SUBST** Proof1 **INTO** Proof2
4. {a} IS-A **Regular** Language	**4. APPLY** RegLangIFFRegEx ⇐ **TO** **mk-AND-proof** ASSUMPTION, Proof1
5. rev LangOf(a) IS-A **Regular** Language	**5. SUBST** Proof3 **INTO** Proof4

L IS-A **Regular** Language RegLangIFFRegEx

IF AND ONLY IF

R IS-A **Regular** Expression AND LangOf(R) = L

IF R IS-A **Regular** Expression THEN rev(LangOf(R)) IS-A **Regular** Language

"main" proof template

let RecRevProof = **`mk-Inductive-proof`** (**on** proof of R IS-A **Regular** Expression):

	Statements	Justifications
$R = a$	1. IF a IS-A **Regular** Expression THEN rev(LangOf(a)) IS-A **Regular** Language	1. BaseCaseProof_a ☑
$R = \varepsilon$	2. IF ε IS-A **Regular** Expression THEN rev(LangOf(ε)) IS-A **Regular** Language	2. $\text{BaseCaseProof}_\varepsilon$
$R = \emptyset$	3. IF $\emptyset$ IS-A **Regular** Expression THEN rev(LangOf($\emptyset$)) IS-A **Regular** Language	3. $\text{BaseCaseProof}_\emptyset$
$R = R_1 \cup R_2$	4. IF $R_1 \cup R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \cup R_2$)) IS-A **Regular** Language	4. APPLY $\text{RecCaseProof}_\cup$ TO RecRevProof(R_1), RecRevProof(R_2)
$R = R_1 \circ R_2$	5. IF $R_1 \circ R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \circ R_2$)) IS-A **Regular** Language	5. APPLY RecCaseProofo TO RecRevProof(R_1), RecRevProof(R_2)
$R = R_1^*$	6. IF R_1^* IS-A **Regular** Expression THEN rev(LangOf(R_1^*)) IS-A **Regular** Language	6. APPLY RecCaseProof* TO RecRevProof(R_1)

5 IFTHEN Proof TODOs!

IF R_1 ∪ R_2 IS-A **Regular** Expression
THEN rev LANGOF(R_1 ∪ R_2) IS-A **Regular** Language

let RecCaseProof$_U$ =

mk-Inductive-Case-proof (with
IH1 = proof of IF R_1 IS-A **Regular** Expression THEN rev LANGOF(R_1) IS-A **Regular** Language,
IH2 = proof of IF R_2 IS-A **Regular** Expression THEN rev LangOf(R_2) IS-A **Regular** Language):
mk-IFTHEN-proof (with ASSUMPTION = proof of R_1 ∪ R_2 IS-A **Regular** Expression):

Still an IFTHEN proof!

Statements	**Justifications**
1. R_1 IS-A **Regular** Expression	**1. FIRST APPLY** DefRegEx4 TO ASSUMPTION
2. R_2 IS-A **Regular** Expression	**2. SECOND APPLY** DefRegEx4 TO ASSUMPTION
3. rev LANGOF(R_1) IS-A **Regular** Language	**3. APPLY** IH1 **TO** Proof1
4. rev LANGOF(R_2) IS-A **Regular** Language	**4. APPLY** IH2 **TO** Proof2
5. rev LANGOF(R_1) ∪ rev LANGOF(R_2) IS-A **Regular** Language	**5. APPLY** RegLangsClosedUnion **TO mk-AND-proof** Proof3 Proof4
6. rev LANGOF(R_1) ∪ rev LANGOF(R_2) = rev (LANGOF(R_1) ∪ LANGOF(R_2))	6. RevUnionCommute Todo!
7. LANGOF(R_1 ∪ R_1) = LANGOF(R_1) ∪ LANGOF(R_2)	7. DefRegExCompute4
8. rev LANGOF(R_1) ∪ rev LANGOF(R_2) = rev LANGOF(R_1 ∪ R_1)	8. SUBST Proof7 INTO Proof6
9. rev LANGOF(R_1 ∪ R_1) IS-A **Regular** Language	9. SUBST Proof8 INTO Proof5

IF R IS-A **Regular** Expression THEN rev(LangOf(R)) IS-A **Regular** Language

"main" proof template

let RecRevProof = **`mk-Inductive-proof`** (**on** proof of R IS-A **Regular** Expression):

	Statements	Justifications
$R = a$	1. IF a IS-A **Regular** Expression THEN rev(LangOf(a)) IS-A **Regular** Language	1. BaseCaseProof_a ☑
$R = \varepsilon$	2. IF ε IS-A **Regular** Expression THEN rev(LangOf(ε)) IS-A **Regular** Language	2. $\text{BaseCaseProof}_\varepsilon$
$R = \emptyset$	3. IF $\emptyset$ IS-A **Regular** Expression THEN rev(LangOf($\emptyset$)) IS-A **Regular** Language	3. $\text{BaseCaseProof}_\emptyset$
$R = R_1 \cup R_2$	4. IF $R_1 \cup R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \cup R_2$)) IS-A **Regular** Language	4. APPLY $\text{RecCaseProof}_\cup$ TO RecRevProof(R_1), RecRevProof(R_2) ☑
$R = R_1 \circ R_2$	5. IF $R_1 \circ R_2$ IS-A **Regular** Expression THEN rev(LangOf($R_1 \circ R_2$)) IS-A **Regular** Language	5. APPLY RecCaseProofo TO RecRevProof(R_1), RecRevProof(R_2)
$R = R_1^*$	6. IF R_1^* IS-A **Regular** Expression THEN rev(LangOf(R_1^*)) IS-A **Regular** Language	6. APPLY RecCaseProof* TO RecRevProof(R_1)

4 IFTHEN Proof TODOs!

Remaining cases will use similar reasoning

Next Time

Non-Regular Languages?

- Are there **languages** that are **not regular languages?**

- How can we **prove** that a **language** is **not a regular language?**

