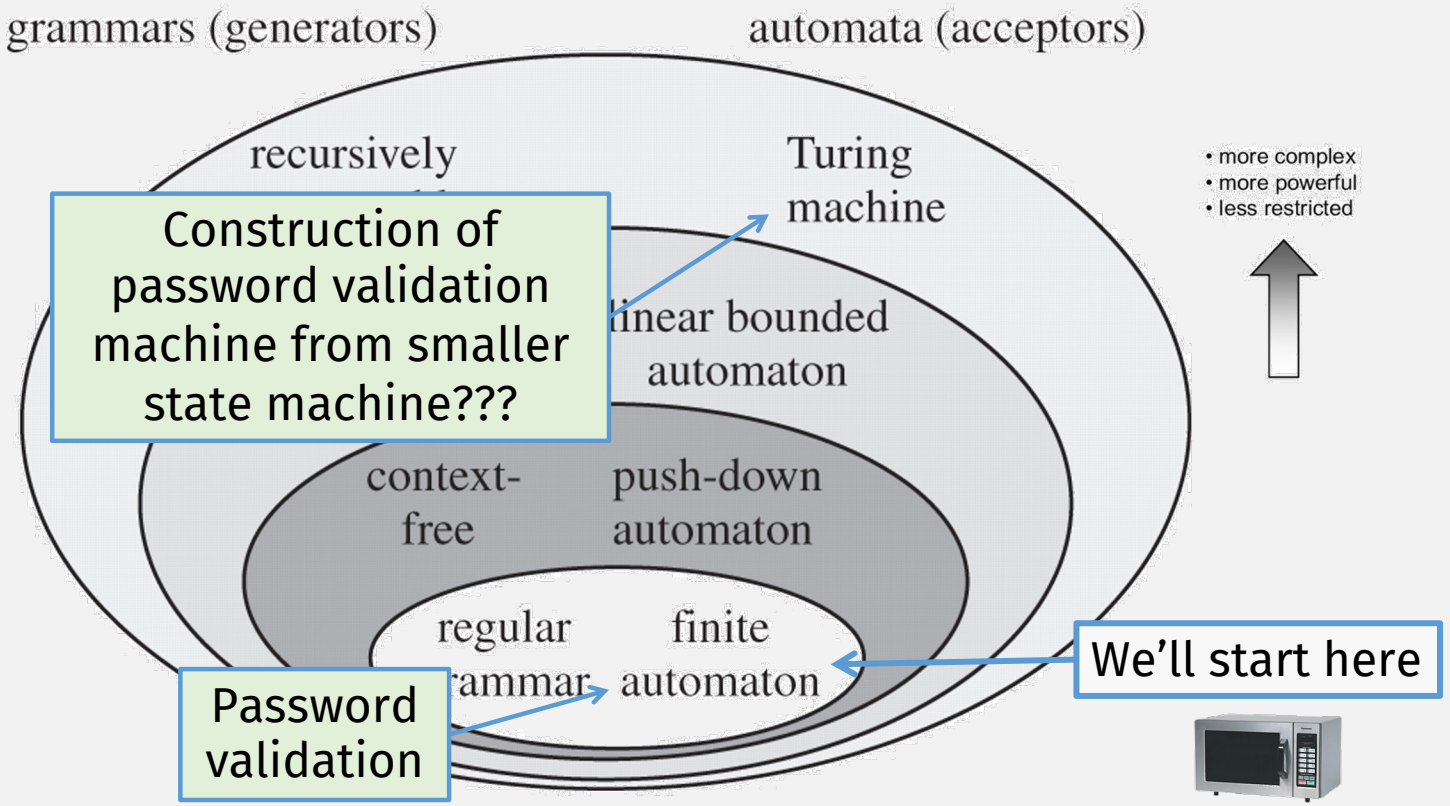
Regular Expressions and Inductive Proofs

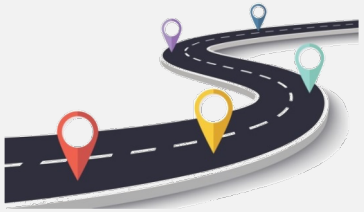
Logistics

- HW3
  - Mostly a repeat of HW1-2 tasks, but for NFAs
- Coding for this class:
  - Forces you to be precise
  - Reinforces that we are studying different kinds of **computation**
    - and meta-computation!
    - Proof by construction = algorithm = computation by a more powerful computer!
    - (see next slide)
  - As computational models get complex, we may be forced to on-paper proofs
- Questions?

Flashback: Levels of Computational *Power*



Last time: Big Picture Road Map

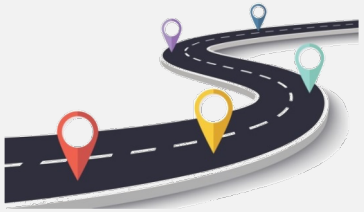


- In this course, we must formally prove the equivalence:
  - Regular Languages ⇔ Regular Expressions Today!

- To do so, we need to prove these ops are closed under reg langs:
  - Union (**done**!)
  - Concatentation (**done**!)
  - Kleene star (**done**!)

- To prove closure properties, we using NFAs:
  - Need NFA ⇔ DFA equivalence theorem (**done**!)

By the end of class today …



- We'll have proven that all these are equivalent:
  - Deterministic Finite Automaton (DFA)
  - Non-deterministic Finite Automaton (NFA)
  - Generalized Non-deterministic Finite Automaton (GNFA)
  - Regular Expressions

- They all represent a regular language!

Last time: Regular Expressions

- **Regular expressions** are widely used by programmers
  - But they can only match regular languages
  - So to *properly* use reg. exps, you must know what is/isn't a regular lang!

RegEx match open tags except XHTML self-contained tags

Asked 11 years, 3 months ago Active 3 months ago Viewed 3.1m times

1647

I need to match all of these opening tags:

```
<p>
<a href="foo">
```

But not these:

```
<br />
<hr class="foo" />
```

6651

4413

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions.

Expressions

Small Expression

$4.23

Regular Expression

/\$\d+\.?\d{2}/

Large Expression

$6.23

U. S. AIR FORCE

PROJECT RAND

RESEARCH MEMORANDUM

REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA

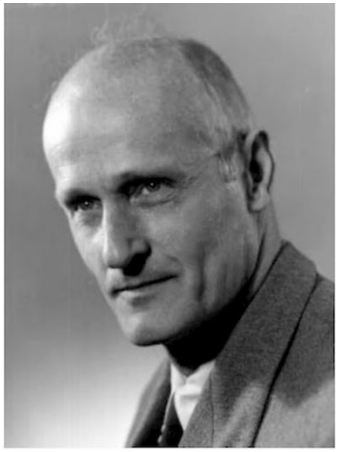
S. C. Kleene

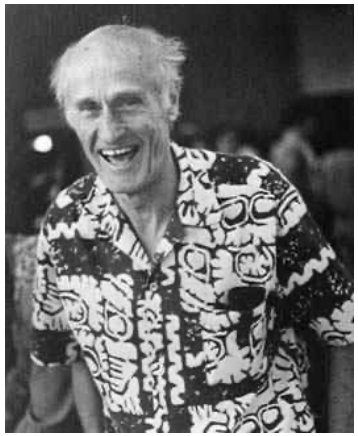
RM-704

15 December 1951

7. Regular Events:

7.1 "Regular events" defined: We shall presently describe a class of events which we will call "regular events." (We would welcome any suggestions as to a more descriptive term.*)





BTW: Is “Regular” the right name?

*There are only two hard things in Computer Science: cache invalidation and **naming things**.*

-- Phil Karlton

Is “Regular” at least a *decent* fit?

What would be a better name?

Regular Expressions, Formal Definition

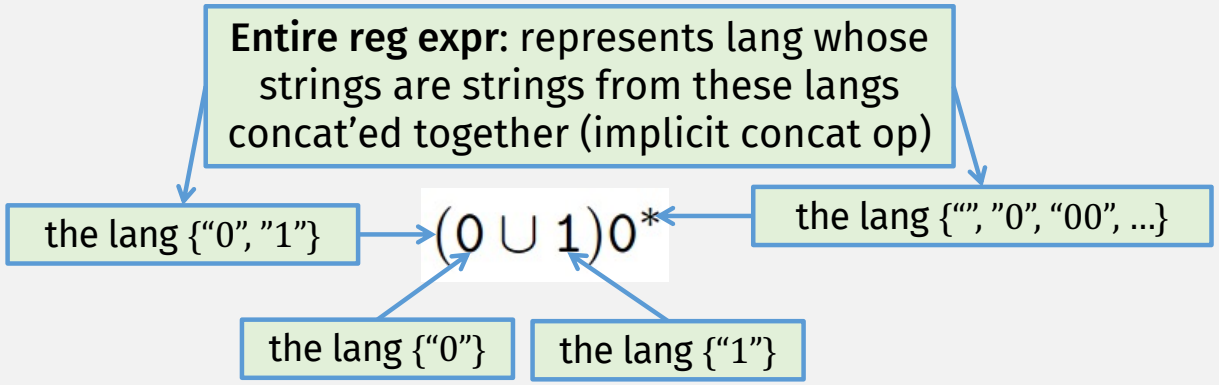
Remember:
A **Regular Expression** represents a (regular) language, i.e., a set of strings!

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

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)
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions, Union of langs
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or Concat of langs
6. (R_1^*), where R_1 is a regular expression. Star of langs

Regular Expression: Concrete Example

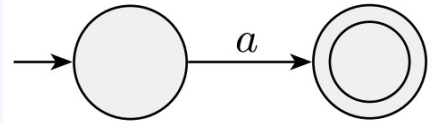


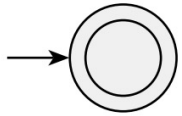
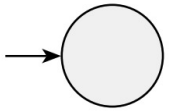
- Operator Precedence:
  - Parens
  - Star
  - Concat (sometimes implicit)
  - Union

Lemma 1.55: Regexp -> NFA

DEFINITION 1.52

Say that 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 r
6. (R_1^*), where R_1 is a regular expression.

Constructions from before!

Exercise: Regular expr over Σ = {0, 1} for ...

- You can use Σ as a shorthand for "the alphabet" if you wish"

- {w | w contains a single 1}
- {w | the length of w is a multiple of 3}
- {w | w contains the string 001 as a substring}

Exercise Pt 2: Let’s try some more

- You can use Σ as a shorthand for "the alphabet" if you wish"

- {w | w starts and ends with the same symbol}
- {w | w = ε }
- {w | every 0 in w is followed by at least one 1}

Thm: A lang is regular **iff** some reg expr describes it

- => If a language is regular, it is described by a reg expr

- <= If a language is described by a reg expr, it is regular
  - Easy!
  - For a given regexp, construct the equiv NFA!
  - Lemma 1.55

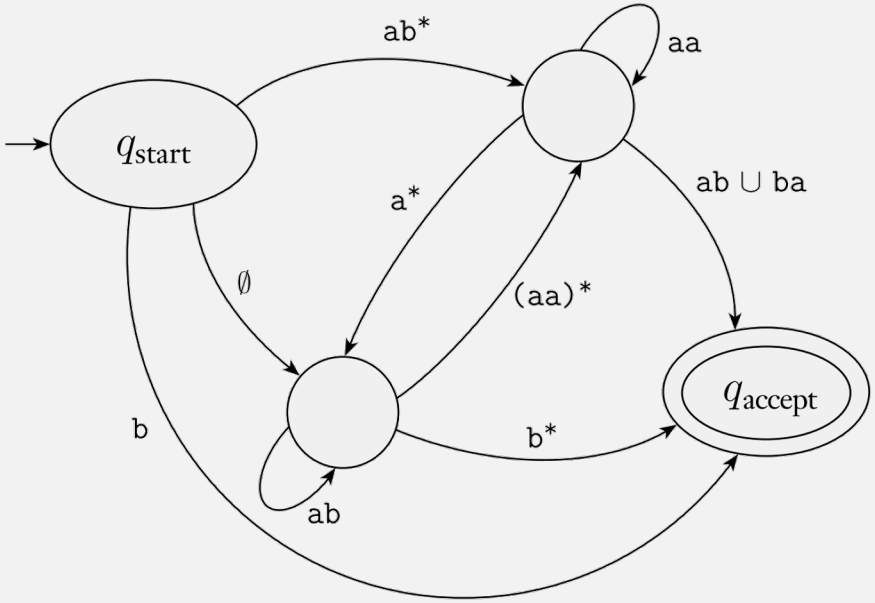
How to show that a lang is regular?

Construct DFA *or* NFA!

Thm: A lang is regular **iff** some reg expr describes it

- => If a language is regular, it is described by a reg expr
  - Hard!
  - Need to convert DFA or NFA to Regular Expression
  - Need something new: a GNFA
- <= If a language is described by a reg expr, it is regular
  - Easy!
  - Construct the NFA! (**Done**)

Generalized NFAs (GNFAs)

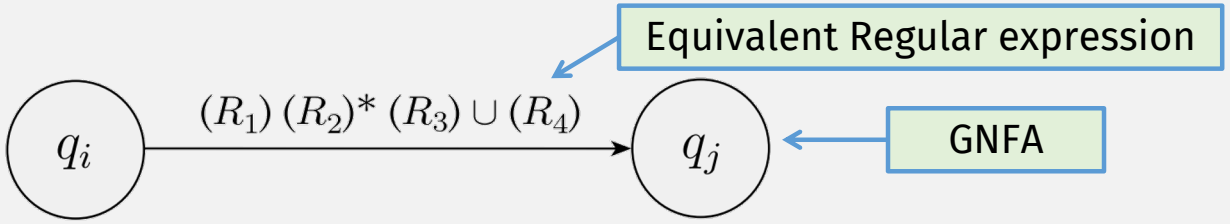


- GNFA = NFA with regular expression transitions

Want to convert GNFAs to Reg Exprs

GNFA->Regexp function

- On GNFA input G:
- If G has 2 states, return the regular expression transition, e.g.:



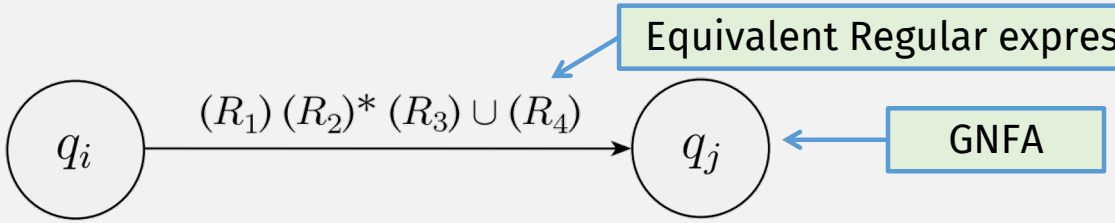
- Else:
  - "Rip out" one state, and "repair", to get G' (has one less state than G)
  - Recursively call GNFA->Regexp(G')

A recursive (function) definition!

GNFA->Regexp function

- On GNFA input G:
- If G has 2 states, return the regular expression transition, e.g.:

Base case



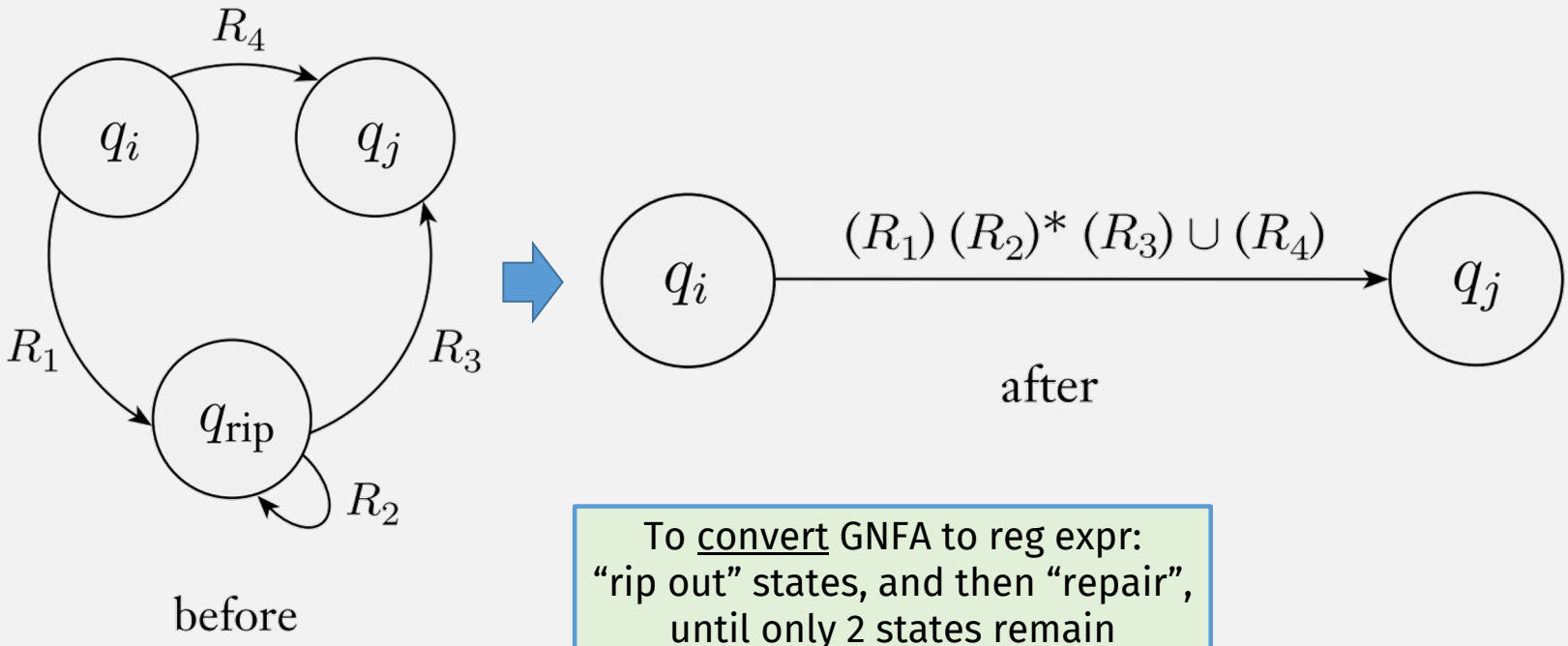
Inductive case

- Else:
  - "Rip out" one state, and "repair", to get G' (has one less state than G)
  - Recursively call GNFA->Regexp(G')

Recursive call is "smaller"

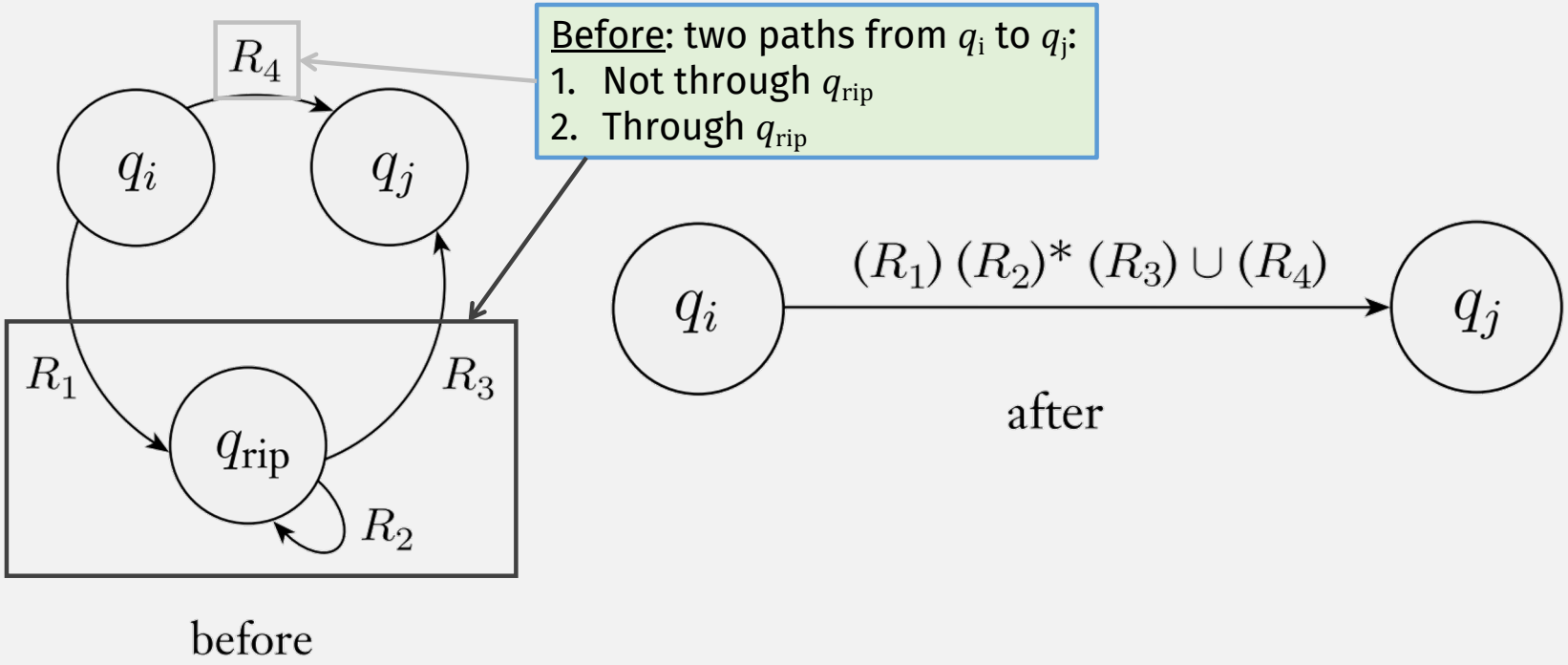
A recursive (function) definition!

GNFA->Regexp function: “Rip/repair” step

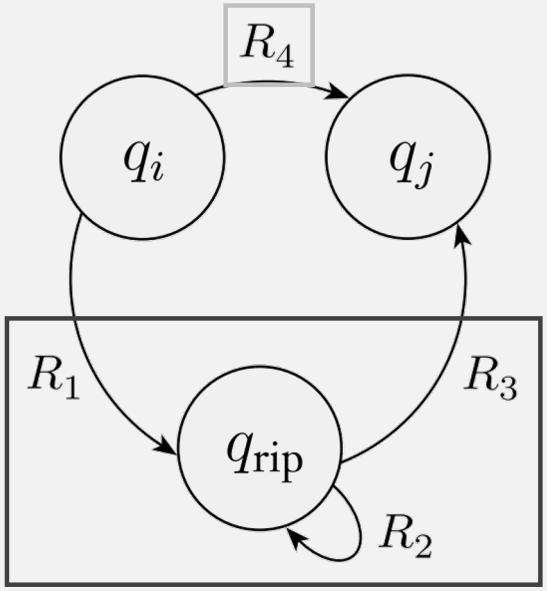


To <u>convert</u> GNFA to reg expr:
“rip out” states, and then “repair”,
until only 2 states remain

GNFA->Regexp function: "Rip/repair" step

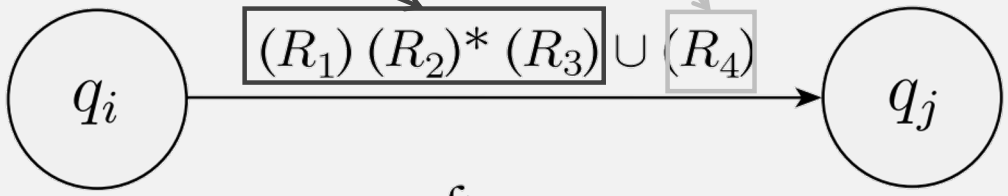


GNFA->Regexp function: "Rip/repair" step



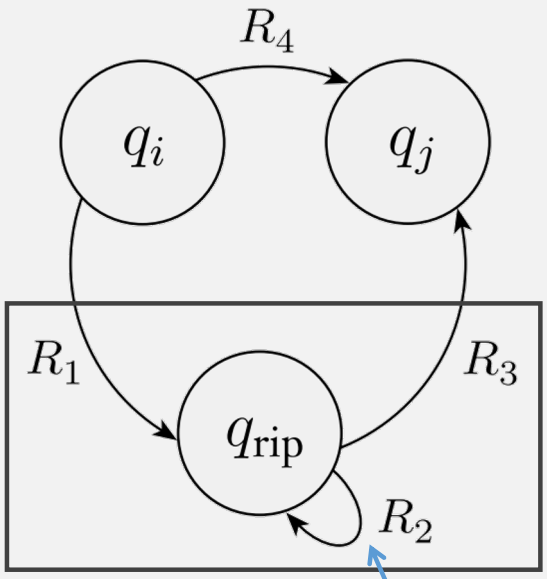
before

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

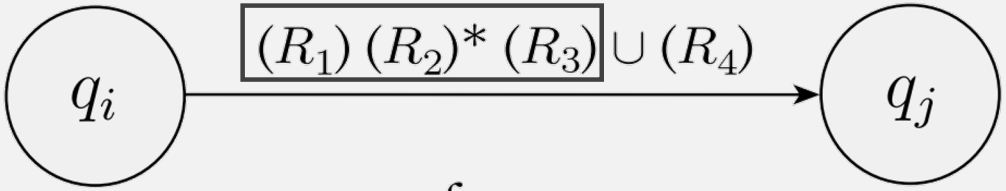


after

GNFA->Regexp function: "Rip/repair" step



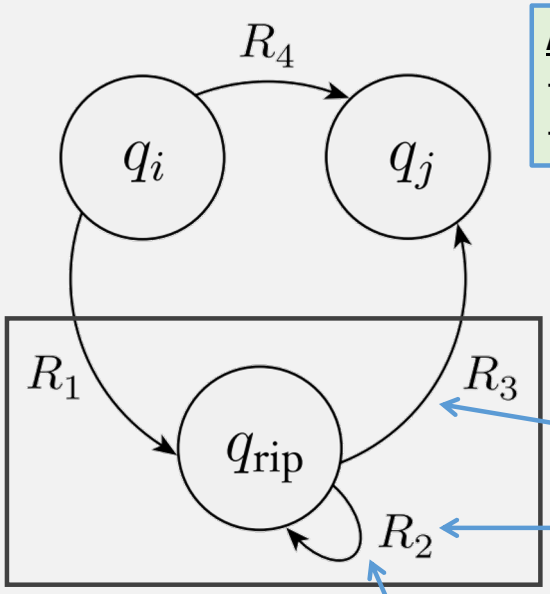
before



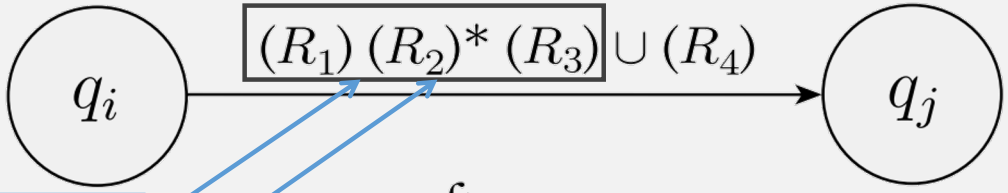
after

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

GNFA->Regexp function: "Rip/repair" step



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



concat

Star operation

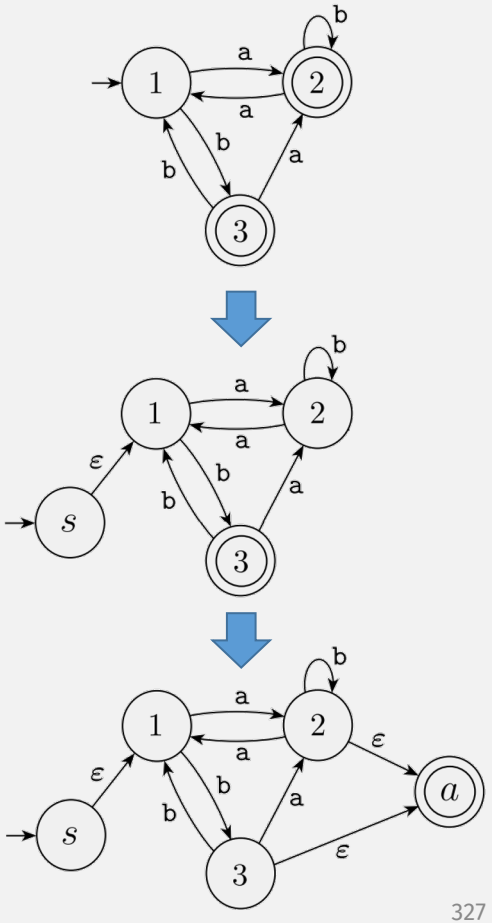
Before:
- path through q_{rip} has 3 transitions
- One is self loop

This is "informal" correctness

This course requires formal correctness, i.e., proof

GNFA->Regexp

- First modifies input machine to have:
  - New start state
    - With no incoming transitions
    - And epsilon transition to old start state
  - New, single accept state
    - With epsilon transitions from old accept states



Need to prove GNFA->Regexp "correct"

- Where "correct" means:

$$\text{LangOf}(G) = \text{LangOf}(\text{GNFA->Regexp}(G))$$

- i.e., GNFA->Regexp must not change the language!

Kinds of Mathematical Proof

- Proof by construction
- Proof by contradiction
- Proof by induction 
  - Use to prove properties of recursive (inductive) defs or functions

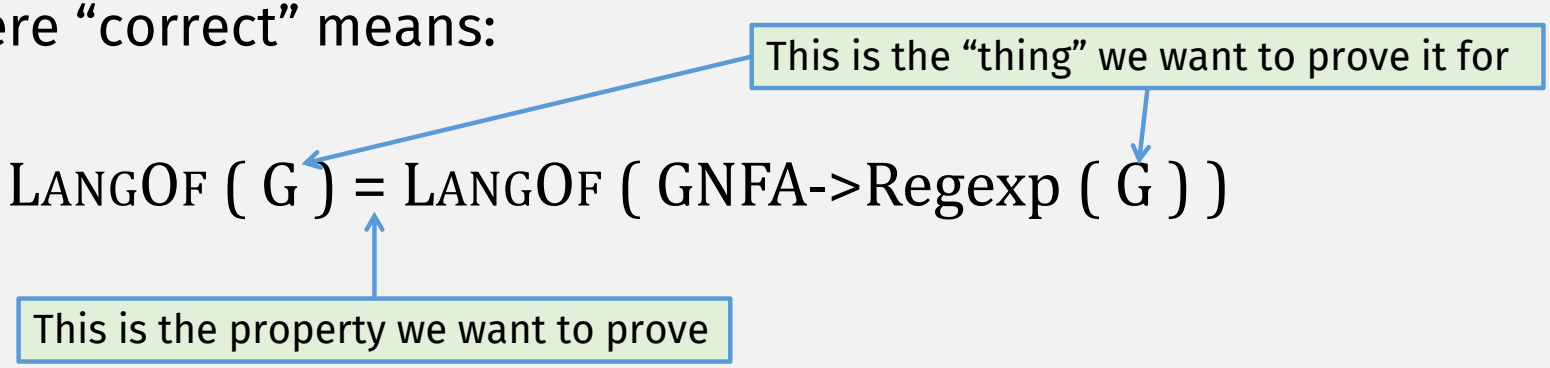
Proof by Induction

- To prove that a **property** P is true for a **thing** x
  - First, prove that P is true for the base case of x (usually easy)
  - Then, prove the induction step:
    - Assume the induction hypothesis (IH):
      - $P(x)$ is true, for some $x_{smaller}$ that smaller than x
    - and use it to prove $P(x)$
    - The *key* is $x_{smaller}$ must be smaller than x

- Why can we assume IH is true???
  - Because we can always start at base case,
  - Then use it to prove for slightly larger case,
  - Then use that to prove for slightly larger case ...

Need to prove GNFA->Regexp "correct"

- Where "correct" means:



- i.e., GNFA->Regexp must not change the language!

GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:

If n = 2: return the reg expr on the transition

Else (G has n > 2 states):

"Rip" out one state to get G'

Recursively Call GNFA->Regexp(G')

➢ Proof (by induction on size of G):

GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of (# states in) G):

GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:

If n = 2: return the reg expr on the transition

Else (G has n > 2 states):

"Rip" out one state to get G'

Recursively Call GNFA->Regexp(G')

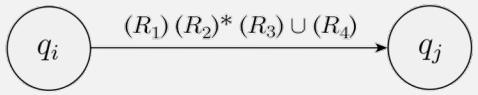
➢ Proof (by induction on size of G):

GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of G):
  - Base case: G has 2 states
    - LANGOF (G) = LANGOF (GNFA->Regexp (G)) is true!

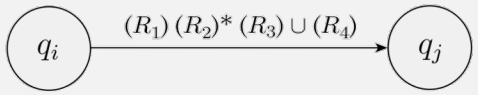


GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of G):



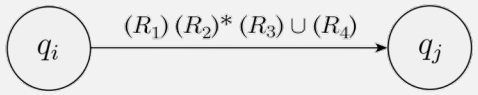
  - Base case: G has 2 states
    - LANGOF (G) = LANGOF (GNFA->Regexp (G)) is true!
  - ➢IH: Assume LANGOF (G') = LANGOF (GNFA->Regexp (G'))
    - For some G' with n-1 states

GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of G):
  - Base case: G has 2 states
    - LANGOF (G) = LANGOF (GNFA->Regexp (G)) is true!
  - IH: Assume LANGOF (G') = LANGOF (GNFA->Regexp (G'))
    - For some G' with n-1 states
  - ➢Induction Step: Prove it's true for G with n states

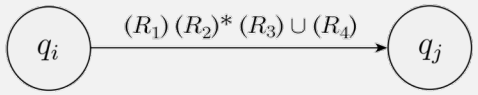


GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of G):
  - Base case: G has 2 states
    - LANGOF (G) = LANGOF (GNFA->Regexp (G)) is true!
  - IH: Assume LANGOF (G') = LANGOF (GNFA->Regexp (G'))
    - For some G' with n-1 states
  - ➢Induction Step: Prove it's true for G with n states
    - After "rip" step, we have exactly a GNFA with n-1 states
    - And we know LANGOF (G') = LANGOF (GNFA->Regexp (G')) from the IH!

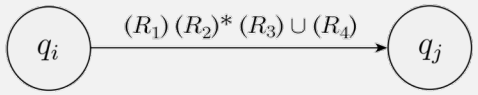


GNFA->Regexp is correct

LANGOF (G)
=
LANGOF (GNFA->Regexp (G))

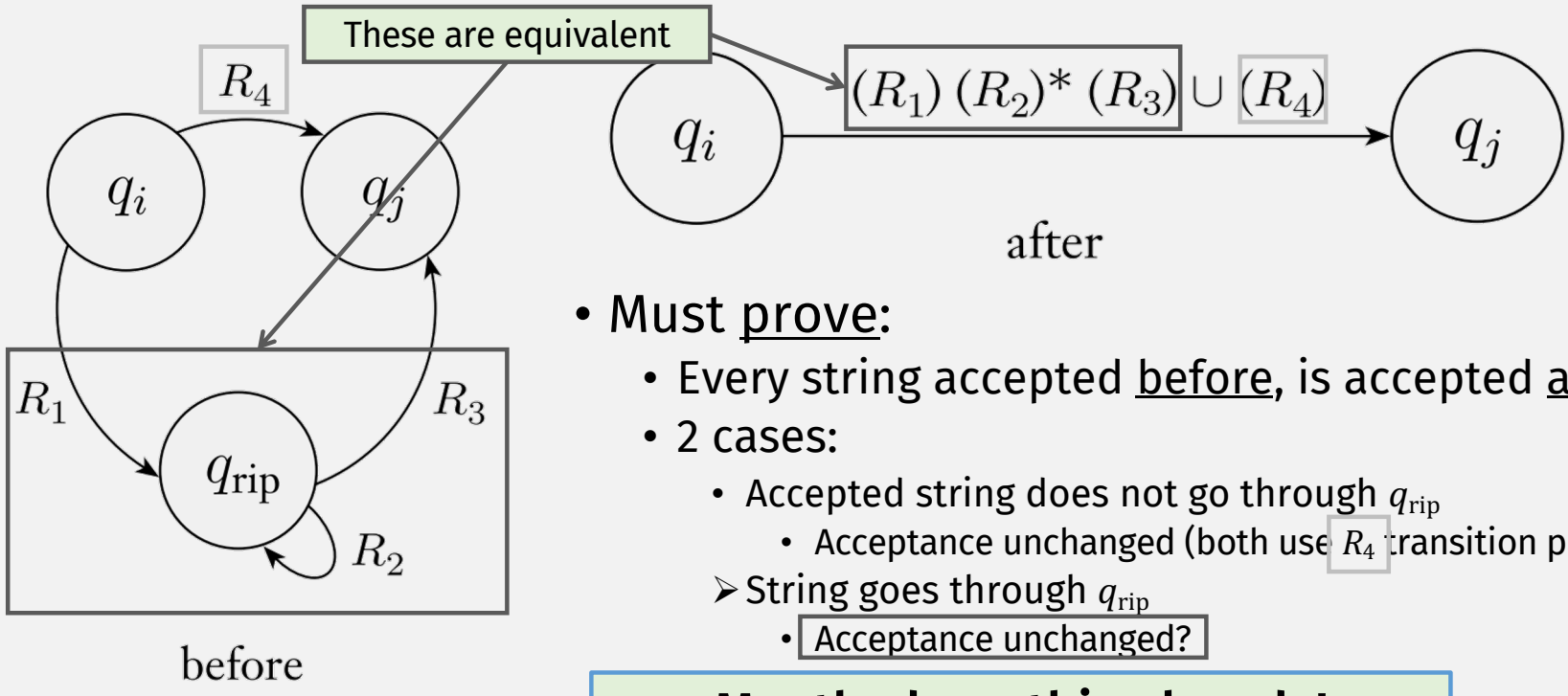
Def: GNFA->Regexp: input G is a GNFA with n states:
If n = 2: return the reg expr on the transition
Else (G has n > 2 states):
"Rip" out one state to get G'
Recursively Call GNFA->Regexp(G')

- Proof (by induction on size of G):



  - Base case: G has 2 states
    - LANGOF (G) = LANGOF (GNFA->Regexp (G)) is true!
  - IH: Assume LANGOF (G') = LANGOF (GNFA->Regexp (G'))
    - For some G' with n-1 states
  - Induction Step: Prove it's true for G with n states
    - After "rip" step, we have exactly a GNFA with n-1 states
    - And we know LANGOF (G') = LANGOF (GNFA->Regexp (G')) from the IH!
    - ➢ To go from G to G': need to prove correctness of "rip" step

GNFA->Regexp: "rip" step correctness



- Must <u>prove</u>:
  - Every string accepted <u>before</u>, is accepted <u>after</u>
  - 2 cases:
    - Accepted string does not go through q_{rip}
      - Acceptance unchanged (both use R_4 transition part)
    - ➢ String goes through q_{rip}
      - Acceptance unchanged?

Mostly done this already!
Just need to state more formally

Thm: A lang is regular **iff** some reg expr describes it

- => If a language is regular, it is described by a reg expr
  - Hard!
  - Need to convert DFA or NFA to Regular Expression
  - Use GNFA->Regexp to convert GNFA to regular expression**! (Done!)**
- <= If a language is described by a reg expr, it is regular
  - Easy!
  - Construct the NFA! (**Done**)

Now we may confidently use regular expressions to represent regular langs.

JFLAP Demo

Not quite our algorithm, so a little extra attention

Check-in Quiz

On gradescope