Regular Expressions (1.3)

COT 4210 Discrete Structures II
Summer 2025
Department of Computer Science
Dr. Steinberg

Mathematical Expressions

You're familiar with mathematical expressions in general:

$$1 + 2 / 4$$

- This expression generates a number, based on its operators, operands and their precedents
- We can use the regular operations to generate *languages*, using languages as operands and representations of the regular operations as operators

$$(0 \cup 1)0^*$$

- This expression generates a language – the language consisting of all strings beginning with a 0 or a 1, and followed by any number (including zero) of zeroes

Regular Expressions: Important Note

- You’ve probably worked with regular expressions before
- Variations include:
    - POSIX Basic regular expressions
    - POSIX Extended regular expressions
    - GNU regular expressions
    - Perl regular expressions
    - Microsoft regular expressions

Regular Expressions: Important Note

- **FORGET EVERYTHING YOU KNOW FROM ALL OF THESE!**
  - The regular expressions we are about to work with are much, *much* more basic than any of the above
  - In particular, some of the above types of “regular expressions” are actually significantly more powerful than theoretical regular expressions!
  - **DO NOT ASSUME** that a language is regular because you can recognize it with a real-world regex engine

Regular Expressions Generally

In general, for a regular expression describing a language over the alphabet Σ, we write:

- A symbol to represent the language containing the string consisting of itself
- ($a \cup b$) to represent either of symbols *a* or *b*
- $a \circ b$ or just *ab* to represent symbol *a* concatenated with symbol *b*
- Σ to represent any symbol from Σ
- a^* to represent zero or more occurrences of *a*
- Σ^* to represent zero or more occurrences of any symbol from Σ

We extend all of these as normal – we can union, concatenate and star-close any regular expressions with each other

- Absent parentheses:
  - Star closure has precedence over concatenation
  - Concatenation has precedence over union

Definition: Regular Expressions

- R is a **regular expression** over the alphabet Σ if it is:
  1. a for some $a \in \Sigma$
  2. λ
  3. $\varnothing$
  4. $(R_1 \cup R_2)$ where R_1 and R_2 are both regular expressions
     - Note: $R_1 \cup R_2$ can also be written as $R_1 + R_2$
  5. $(R_1 \circ R_2)$ where R_1 and R_2 are both regular expressions
  6. $(R_1{}^*)$ where R_1 is a regular expression
- 1-3 represent the languages $\{a\}$, $\{\lambda\}$ and the empty language, respectively
- 4-6 represent the union, concatenation and star closure of the language(s) described by the regular expression operand(s)

Regular Expression Examples

(Board work: Example 1.53 and others)

Regular Expressions Describe Regular Languages

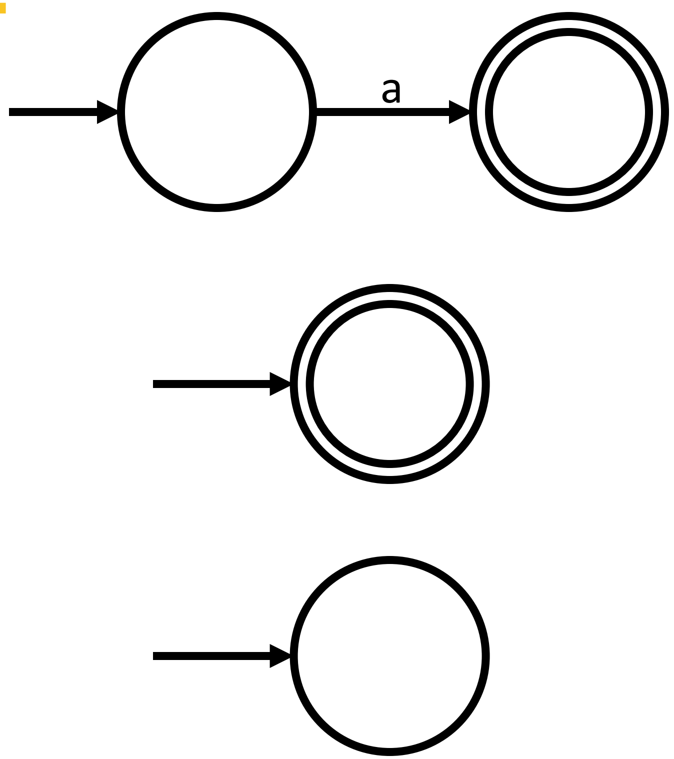
Regular Expression Equivalence

- You've already guessed that regular expressions describe all and only the regular languages
  - Now we're going to prove it
- This is *not* a proof we can do on the board
  - It's far too complicated
  - We're going to step through it in slide format
- **You will *not* be expected to do a proof this complex yourself in this class.**
- We'll split the set equivalence proof as normal, and prove that:
  - If a language is described by a regular expression, then that language is regular
  - If a language is regular, it is described by a regular expression

Equivalence Direction 1

- Consider a language *A* described by a regular expression *R*. It suffices to show that there is an NFA recognizing *A*.
- Given the definition of regular expressions R can take one of six forms. It suffices in turn to show that NFAs can recognize languages in each of them.

Equivalence Direction 1



NFA_a

NFA_λ

$NFA_{no.}$

Consider each case of the definition of regular expressions

1. $R = a$ for some $a \in \Sigma$
2. $R = \lambda$
3. $R = \emptyset$
4. $R = R_1 \, U \, R_2$
5. $R = R_1 R_2$
6. $R = R_1^*$

…and for 4-6, we just use the same constructions from the regular class closure proofs

Equivalence Direction 2

- Now we need to show that all regular languages can be described by regular expressions
  - It suffices to show that for every DFA recognizing a language, there is a regular expression that describes the same language
  - As usual, all we need to do is prove that regular expression exists
- This is harder
  - Actually not *that* much harder – but a lot less direct
- To prove this we actually define a new type of automaton: the **generalized nondeterministic finite automaton**

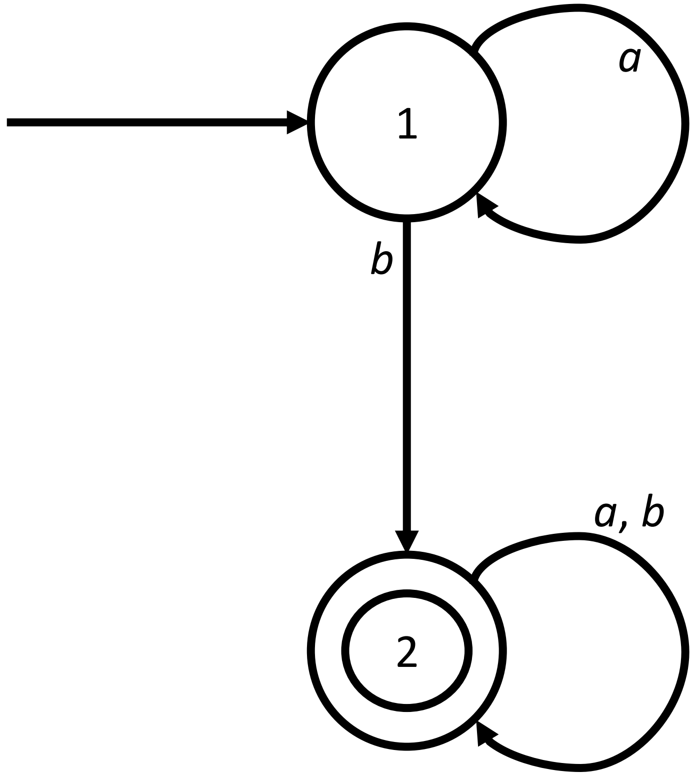
GNFAs

GNFAs generally

- A GNFA is a special kind of NFA that uses regular expressions as its *transition alphabet*
  - A GNFA has a single start state and a single accept state
  - Nothing can transition *into* the start state, and nothing can transition *out of* the accept state
- First, we convert our DFA to a GNFA
  - This is the easy part
- We then convert that GNFA to a regular expression by *state ripping* and *repair*
  - One by one, we remove states from the GNFA, or *rip* the states out
  - After each rip, we expand the expressions on the transitions surrounding the removed state, so that the GNFA still recognizes the same language
- We know we're done when there are only two states left—the start and accept states
  - …and the transition regular expression between them has to be the regular expression recognizing the original language

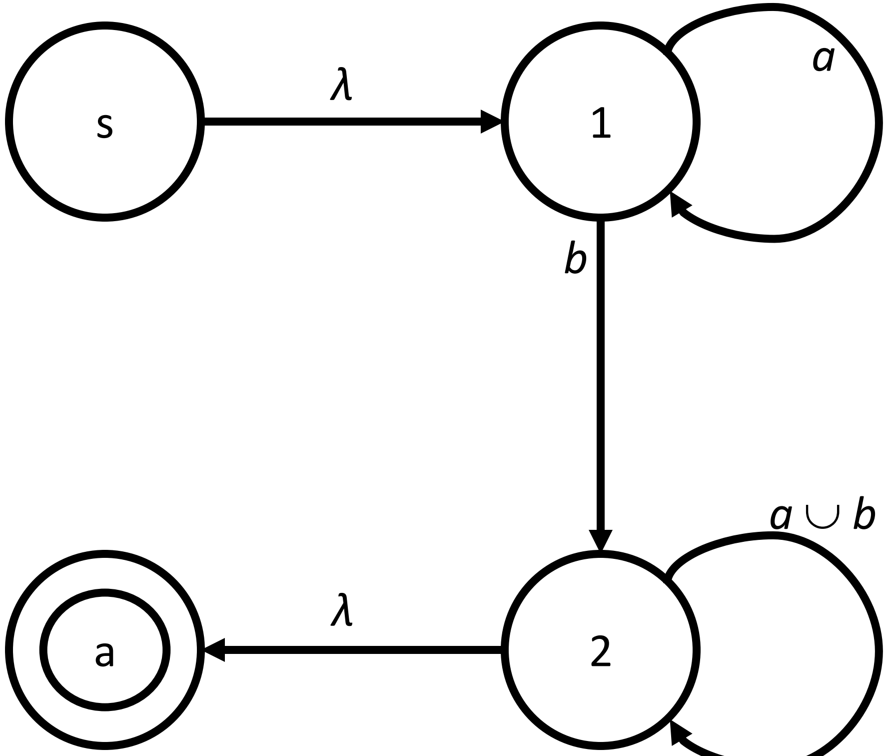
Making the GNFA

- First, we:
  - Add specific start and accept states
  - Add an empty-string transition from the start state to the old start state
  - Add empty transitions from the old accept states to the accept state
  - Convert all the multiple-symbol transitions to use the union operator



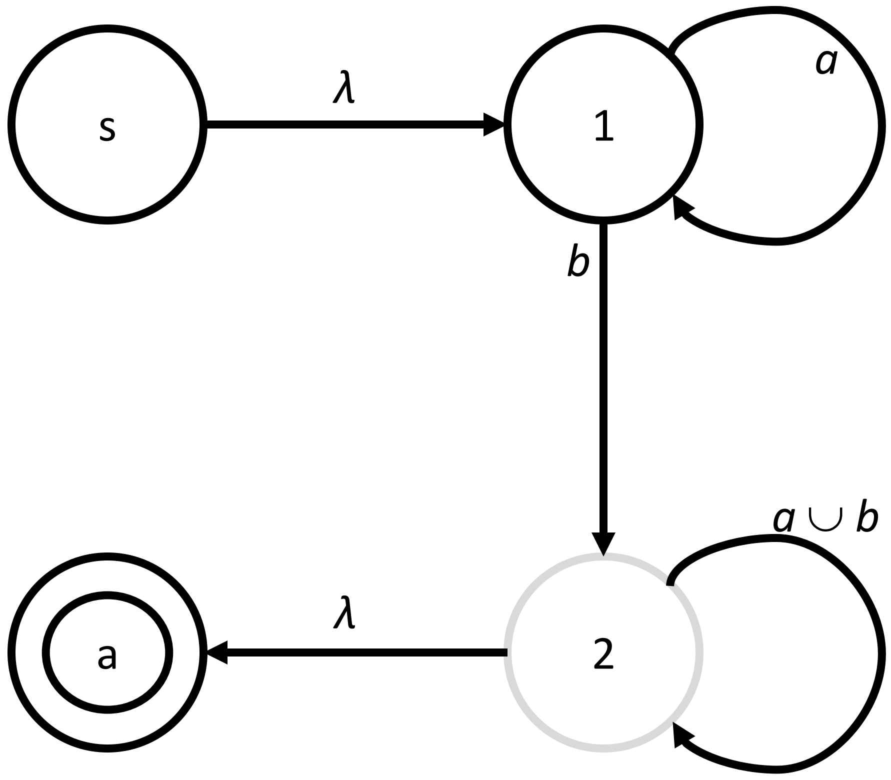
Making the GNFA

- First, we:
  - Add specific start and accept states
  - Add an empty-string transition from the start state to the old start state
  - Add empty transitions from the old accept states to the accept state
  - Convert all the multiple-symbol transitions to use the union operator



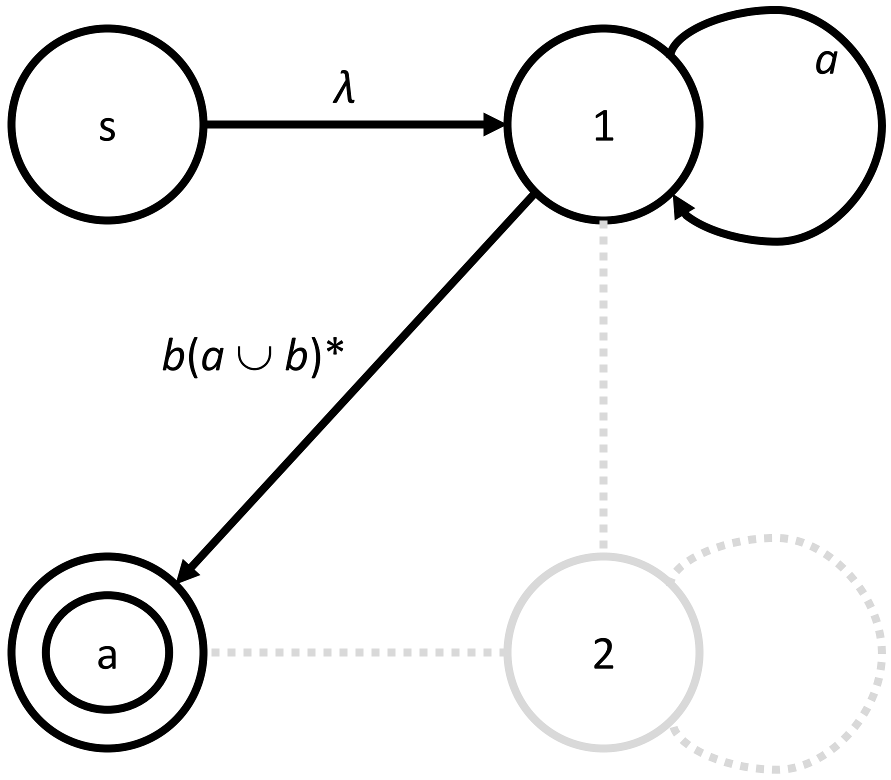
Making the GNFA

- Now we rip out a state
    - It actually doesn't matter which
- 1 transitioned to the accept state *through* 2, so…
    - We need to repair that transition



Making the GNFA

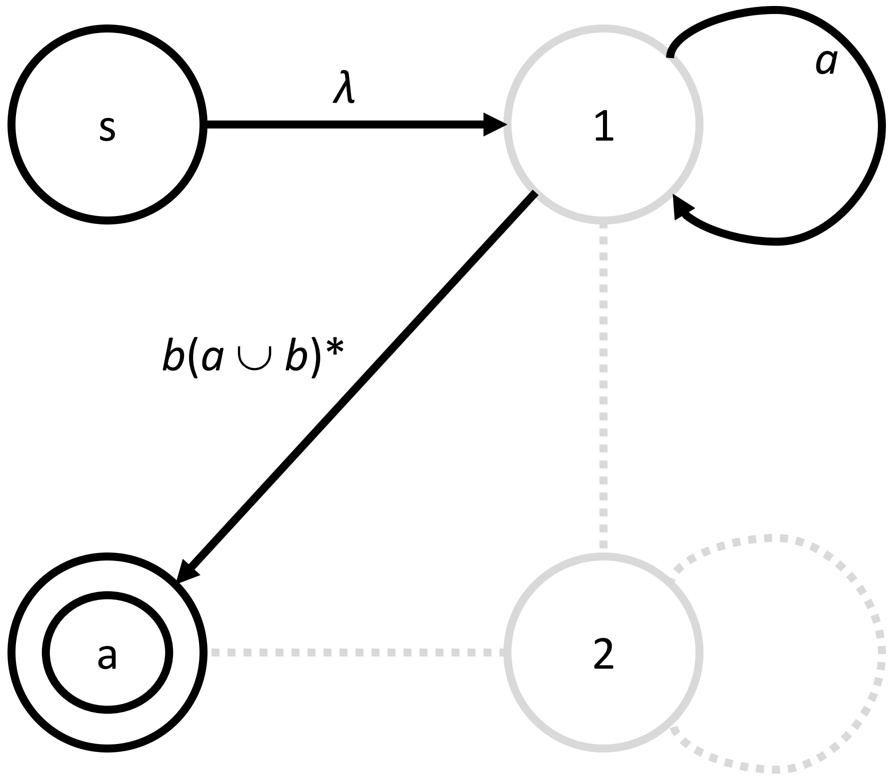
- Now we rip out a state
  - It actually doesn't matter which
- 1 transitioned to the accept state *through* 2, so…
  - We need to repair that transition
  - The concatenation is obvious
  - Can you see why we need the star closure?



Making the GNFA

- Now we rip out a state
  - It actually doesn't matter which
- 1 transitioned to the accept state *through* 2, so…
  - We need to repair that transition
  - The concatenation is obvious
  - Can you see why we need the star closure?

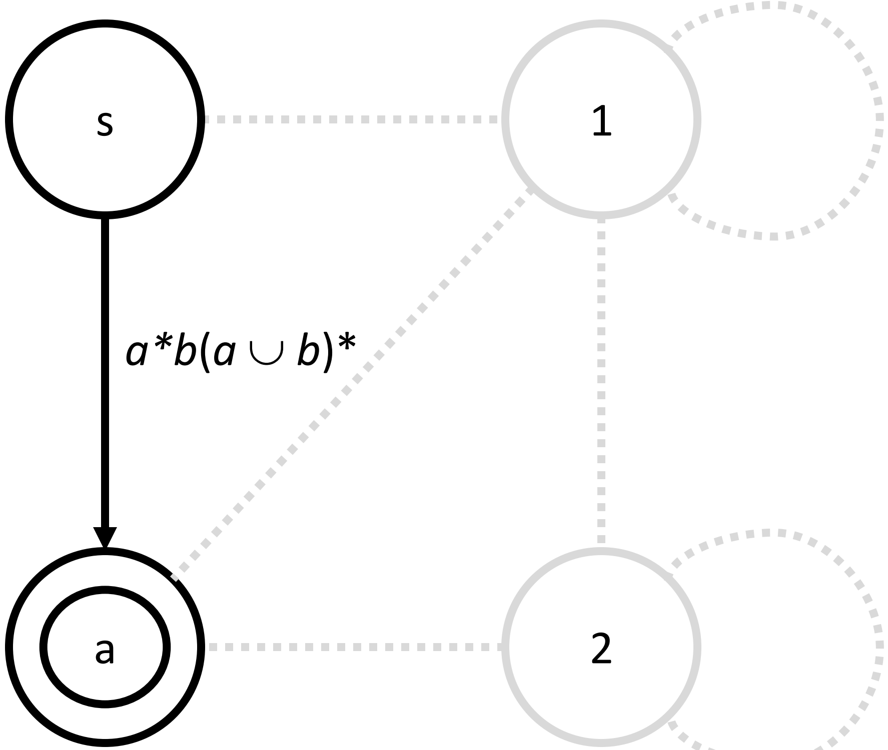
One more state, and we're done



Making the GNFA

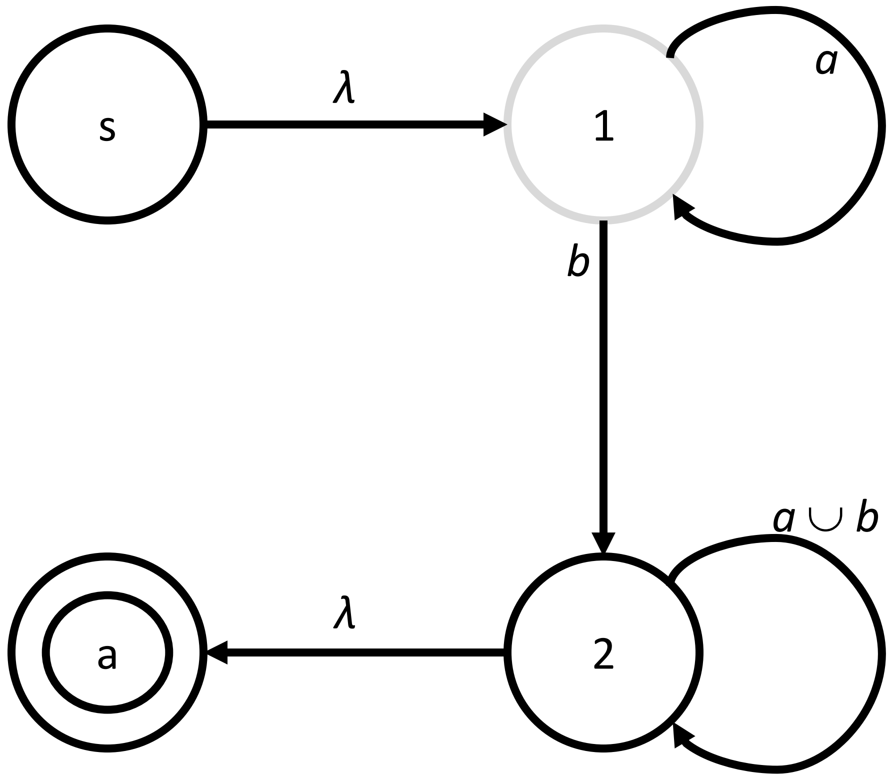
- Now we rip out a state
  - It actually doesn't matter which
- 1 transitioned to the accept state *through* 2, so…
  - We need to repair that transition
  - The concatenation is obvious
  - Can you see why we need the star closure?

One more state, and we're done



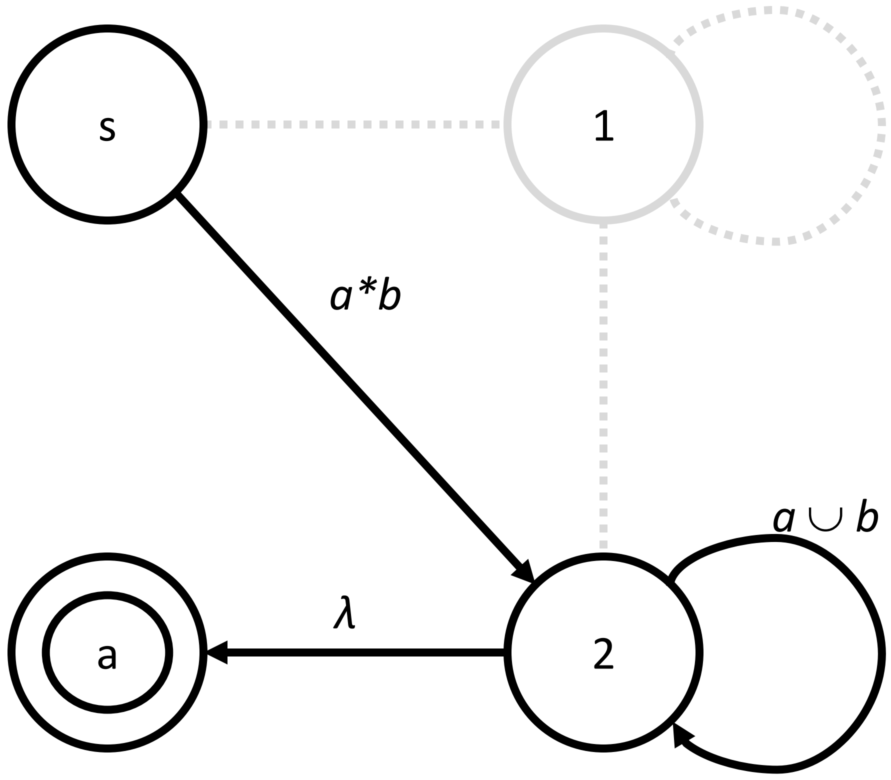
Making the GNFA

- By the way, this also works just fine the other direction



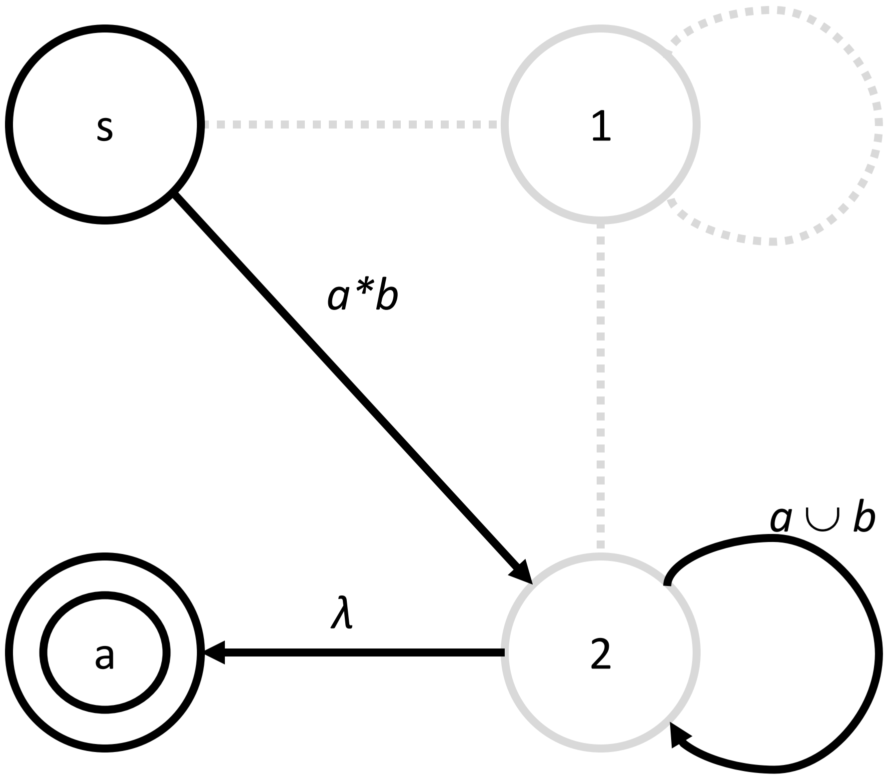
Making the GNFA

- By the way, this also works just fine the other direction



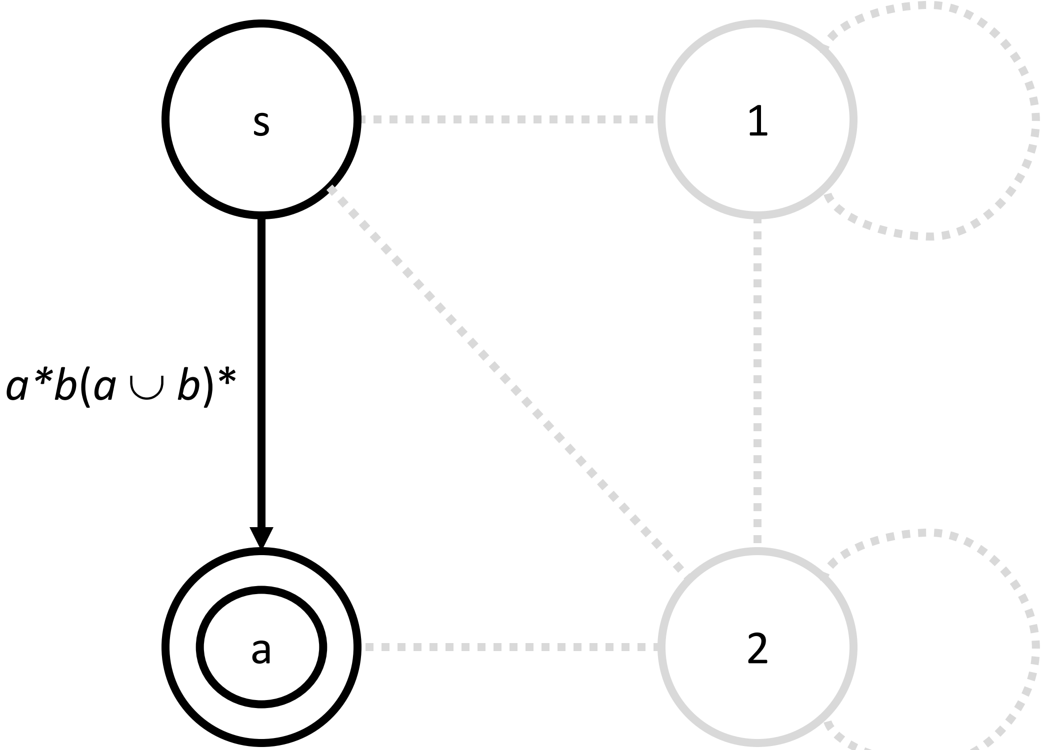
Making the GNFA

- By the way, this also works just fine the other direction



Making the GNFA

- By the way, this also works just fine the other direction



Taking Inventory

- By now we have a decent sense of how the GNFA conversion works
  - We also probably have "warm fuzzy feelings" about describing DFAs' languages with regexes we create using GNFAs
  - …and if we can do that with DFAs, we can with NFAs
- To close the box on the proof, we need to do two things:
  - Figure out how to **reliably** rip and repair – present an algorithm to consistently reduce a GNFA to a single regular expression
  - Pull together our findings into (semi-)formal reasoning

Reliable Ripping and Repair

- Ripping is the easy part: Just pick a state q_r that isn't the start or accept state
- Repair is the hard part. Consider every *pair of states* q_a and q_b so that:
  - q_a can transition to q_r on regular expression R_{ar}
  - q_r can transition to q_b on regular expression R_{rb}
  - (If the transition can go the other way too, it counts as two pairs)
- **Since we aren't picking the start or accept state, it is both necessary and sufficient to repair every such transition**
- Three cases to consider:
  - q_a can always transition to q_b on regular expression $(R_{ar})(R_{rb})$
  - If q_r has a self-loop on R_r then we concatenate with (R_r^*) to get $(R_{ar})(R_r)^*(R_{rb})$
  - And finally, if q_a can transition to q_b on regex R_{ab} without q_r involved, we union with (R_{ab}) to get:

$$\boldsymbol{(R_{ar})(R_r)^*(R_{rb})\cup(R_{ab})}$$

Definition: Generalized Nondeterministic Finite Automaton

- A GNFA is a 5-tuple $G = \{Q, \Sigma, \delta, q_s, q_f\}$ where:
  - Q is the set of states,
  - Σ is the input alphabet,
  - $\delta: (Q - \{q_a\}) \times (Q - \{q_s\}) \rightarrow \boldsymbol{R}$ (with $\boldsymbol{R}$ as the set of all regular expressions over Σ) is the transition function,
  - q_s is the start state, and
  - q_f is the (single) accept state
- A GNFA accepts string w on Σ if:
  - $w = w_1 w_2 \ldots w_k$, and…
  - …state sequence $q_0 q_1 \ldots q_k$ exists, so that $q_0 = q_s$ and $q_k = q_f$, and…
  - $w_i \in L(\delta(q_{i-1}, q_i))$ for i from 1 to k

Recursive Conversion

Let RIP(G) be a function that accepts a GNFA $G = \{Q, \Sigma, \delta, q_s, q_f\}$. It returns $G_R = \{Q_R, \Sigma, \delta_R, q_s, q_f\}$ so that:

- $Q_R = Q - \{q_r\}$ for some $q_r \notin \{q_s, q_f\}$, and
- For every $q_a \in Q_R - \{q_f\}$ and $q_b \in Q_R - \{q_s\}$,

$$\boldsymbol{\delta_R(q_a, q_b) = (R_{ar})(R_r)^*(R_{rb}) \cup (R_{ab})}$$

where: $R_{ar} = \delta(q_a, q_r)$ $R_{rb} = \delta(q_r, q_b)$ $R_r = \delta(q_r, q_r)$ $R_{ab} = \delta(q_a, q_b)$

Now Let CONVERT(G) be a function that accepts a GNFA $G = \{Q, \Sigma, \delta, q_s, q_f\}$. It returns:

- The regular expression $\delta(q_s, q_f)$ if $|Q| = 2$, and
- CONVERT(RIP(G)) otherwise.

A Little Convincing

- We can show RIP(G) is equivalent to G:
  - If G accepts w, then G enters states $q_s, q_1, q_2, \dots, q_f$
    - If none of these are q_r, obviously RIP(G) accepts w
    - If q_r **does** appear, then let the states before and after it be q_a and q_b, and our construction shows that δ_R provides a regular expression transition between them equivalent to all transitions through q_r
  - If RIP(G) accepts w, then RIP(G) enters states $q_s, q_1, q_2, \dots, q_f$
    - If none of the transitions previously involved q_r, obviously G accepts w
    - If a transition **did** previously involve q_r, we just reverse our construction to observe that G can make the same transition through q_r
  - G and RIP(G) each accept everything the other does; therefore, they are equivalent.

Cleaning Up

We have everything we need to show that regular expressions describe the same languages.

Let’s go ahead and do it.

Lemma: DFAs to Regular Expressions

Since we can make a GNFA out of any DFA, to show that a DFA's language can be described by a regular expression, it suffices to show that for a GNFA $G = \{Q, \Sigma, \delta, q_s, q_f\}$, CONVERT($G$) returns a regular expression describing L(G).

- **Proof:** Induction on $|Q|$.
- **Basis:** $|Q| = 2$. Then G has a singular transition from q_s, q_f for strings described by a regular expression $\delta(q_s, q_f) = R$, which CONVERT(G) returns as desired.
- **Induction Hypothesis:** Assume that for any $G_k = \{Q_k, \Sigma, \delta_k, q_{sk}, q_{fk}\}$ with $|Q_k| < |Q|$, CONVERT returns a regular expression describing L(G_k).
- **Induction:** Consider RIP(G) $= \{Q_R, \Sigma, \delta_R, q_s, q_f\}$.
  - By definition of RIP(G), $|Q_R|=|Q| - 1$.
  - Therefore, by the induction hypothesis, CONVERT(RIP(G)) returns a regular expression describing L(RIP(G)).
  - We have already shown that L(RIP(G)) = L(G).
  - CONVERT(RIP(G)) returns a regular expression describing L(G), as desired.

Regular Expression/Regular Language Equivalence

We split the set equivalence proof as normal. We need to prove two things:

If a language is regular, it is described by a regular expression

- Handled by last slide's lemma.

If a language is described by a regular expression, then that language is regular

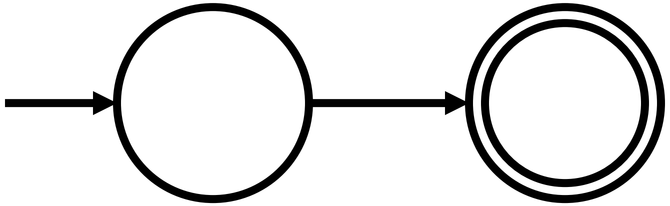
- Recall the definition of regular languages, which we show here to the right.

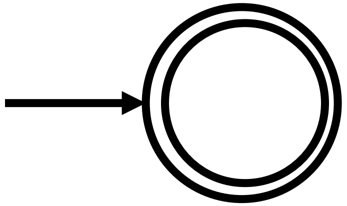
R is a **regular expression** over the alphabet Σ if it is:

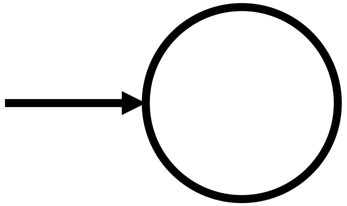
1. a for some $a \in \Sigma$
2. λ
3. $\varnothing$
4. $(R_1 \cup R_2)$ where R_1 and R_2 are both regular expressions
5. $(R_1 \circ R_2)$ where R_1 and R_2 are both regular expressions
6. $(R_1{}^*)$ where R_1 is a regular expression

- 1-3 represent the languages $\{a\}$, $\{\lambda\}$ and the empty language, respectively
- 4-6 represent the union, concatenation and star closure of the language(s) described by the regular expression operand(s)

Expression-to-Language Equivalence







These DFAs are enough to handle the basic cases of regular expressions…

1. $R = a$ for some $a \in \Sigma$
2. $R = \lambda$
3. $R = \emptyset$
4. $R = R_1 \; U \; R_2$
5. $R = R_1 R_2$
6. $R = R_1^*$

…and for 4-6, we just re-use the constructions from our regular class closure proofs.

Conversion Examples

(Board work: Examples 1.56, 1.58, 1.68)

Acknowledgement

Some Notes and content come from Dr. Gerber, Dr. Hughes, and Mr. Guha's COT4210 class and the Sipser Textbook, *Introduction to the Theory of Computation*, 3rd ed., 2013