Lecture 3

COT4210 DISCRETE STRUCTURES

DR. MATTHEW B. GERBER

5/26/2016

PORTIONS FROM SIPSER, *INTRODUCTION TO THE THEORY OF COMPUTATION*, 3RD ED., 2013

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

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. ε
3. $\emptyset$
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 $\{a\}$, $\{\varepsilon\}$ 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)

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:

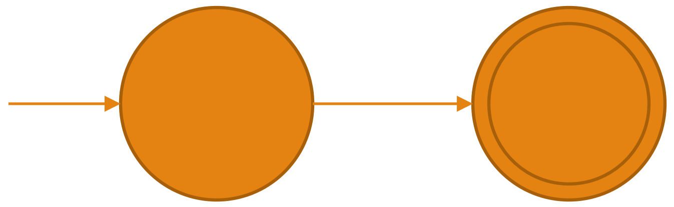
- If a language is described by a regular expression, then that language is regular
- If a language is regular, it is described by a regular expression

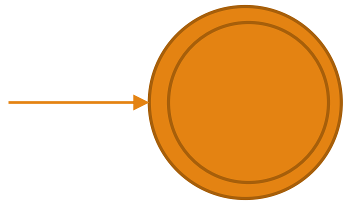
Equivalence Direction 1

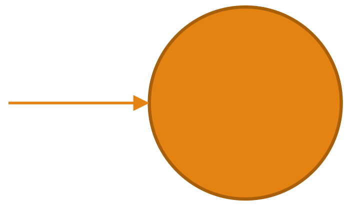
Consider a language *A* described by a regular expression *R*. It suffices to show that there is an NFA recognizing *A*.

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







Consider each case of the definition of regular expressions

1. $R = a$ for some $a \in \Sigma$
2. $R = \varepsilon$
3. $R = \emptyset$

…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 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

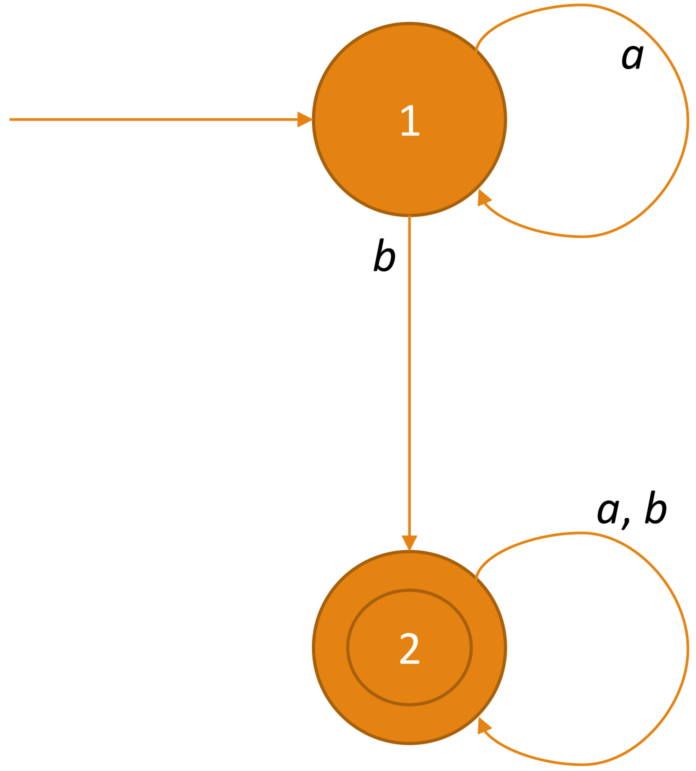
We know we're done when there are only two states left—the start and accept states

- …and the transition regular expression between them has to be the regular expression recognizing the original language

Making the GNFA

First, we:

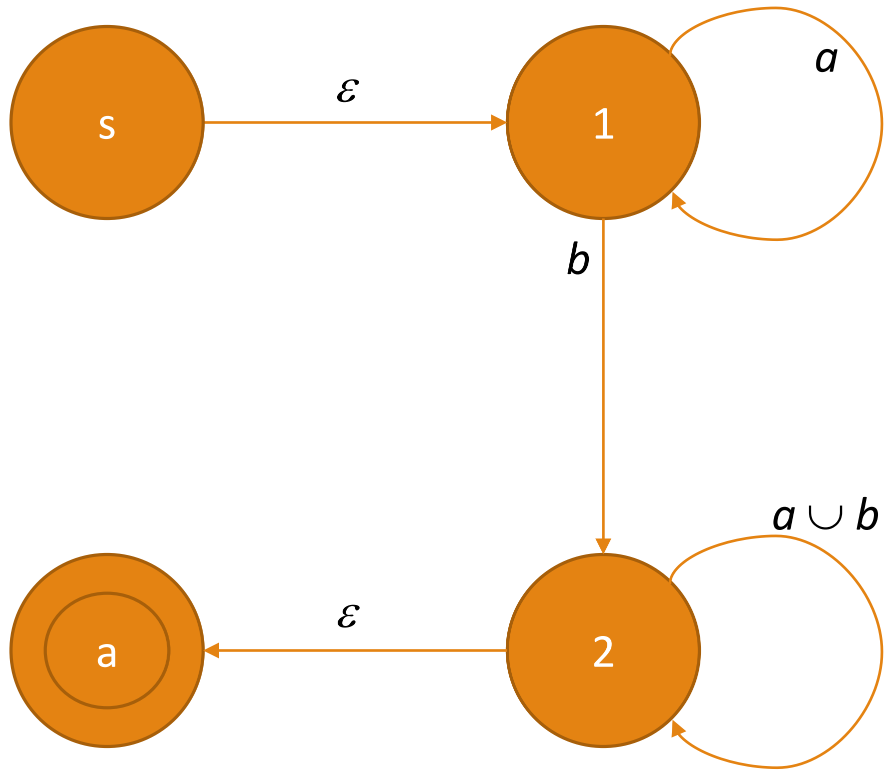
- Add specific start and accept states
- Add an empty-string transition from the start state to the old start state
- Add empty transitions from the old accept states to the accept state
- Convert all the multiple-symbol transitions to use the union operator



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



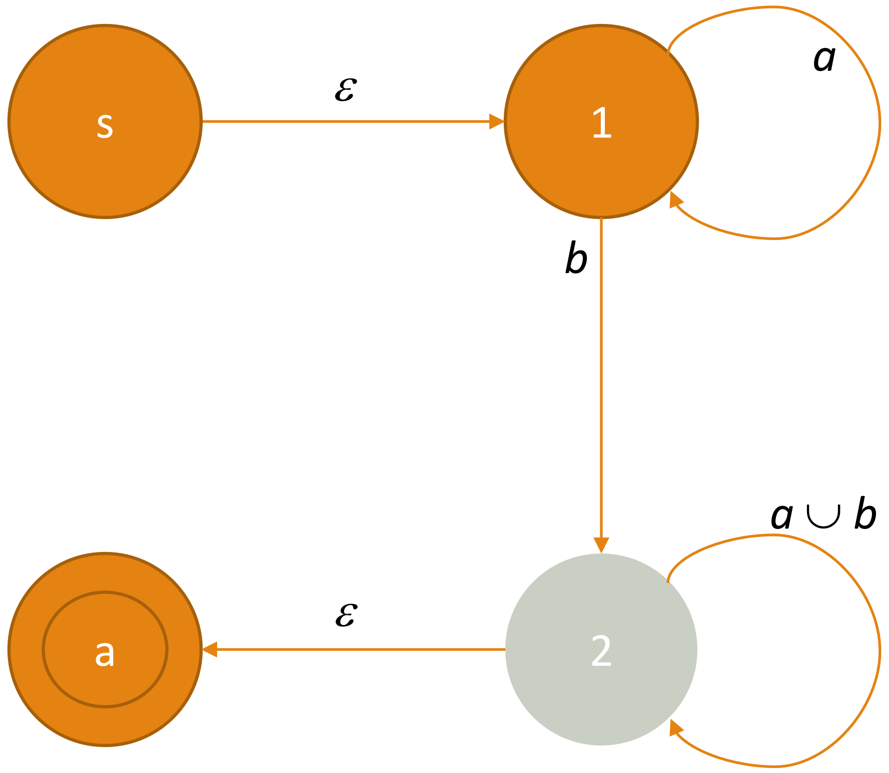
Making the GNFA

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



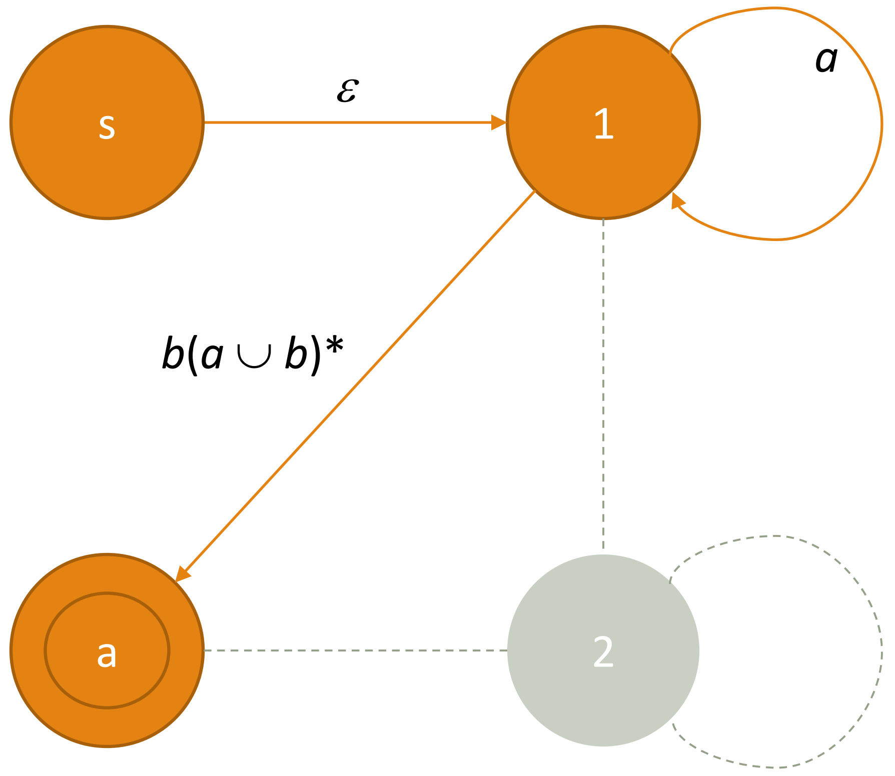
Making the GNFA

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

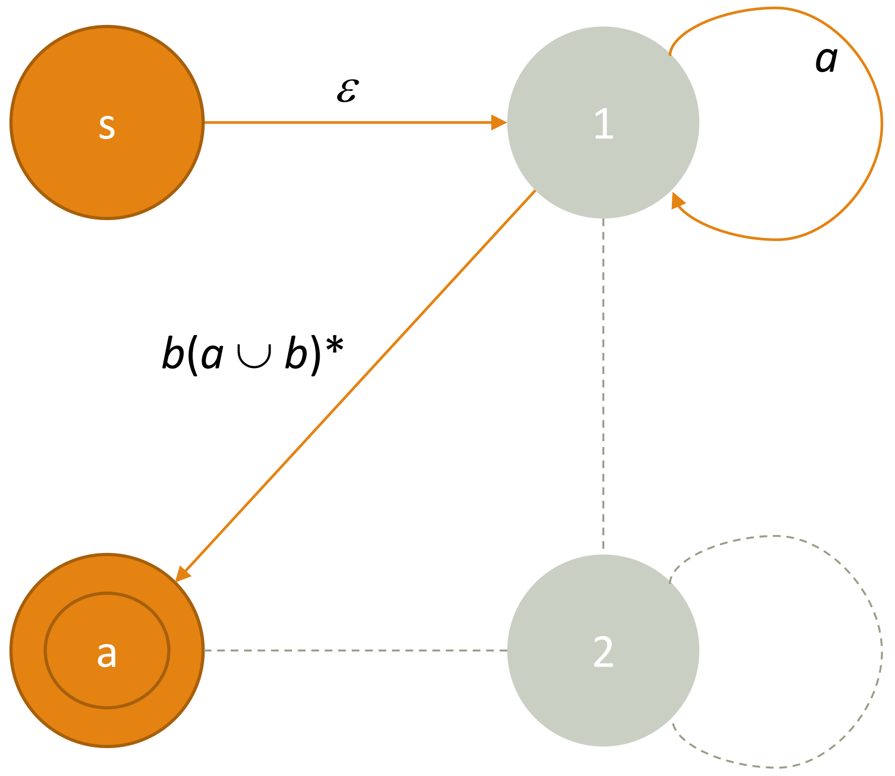
Now we rip out a state

- 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

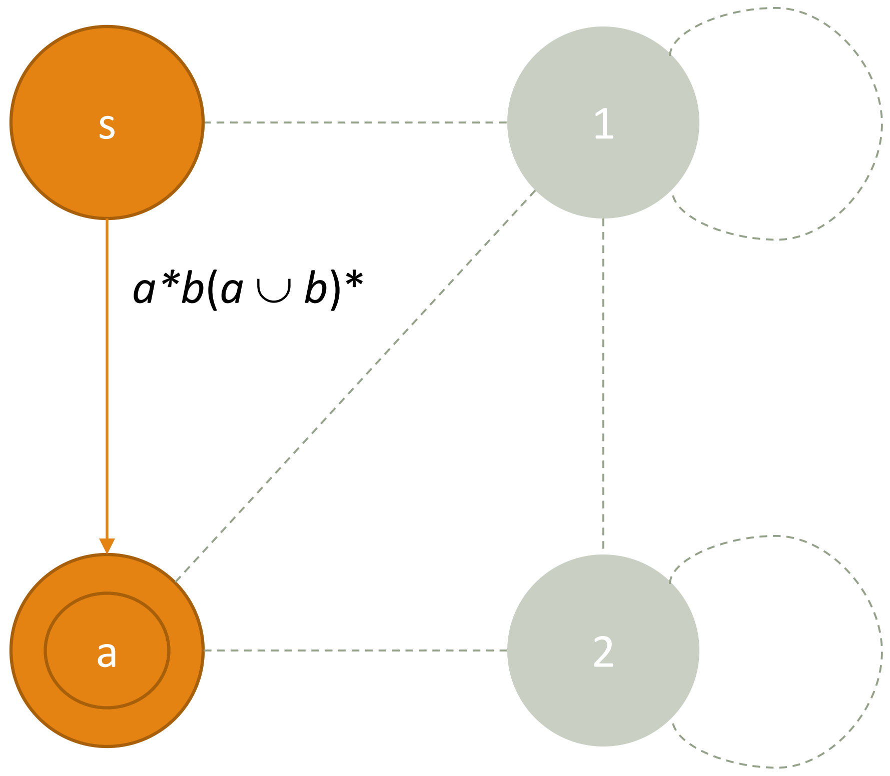
Now we rip out a state

- It actually doesn't matter which

1 transitioned to the accept state *through* 2, so…

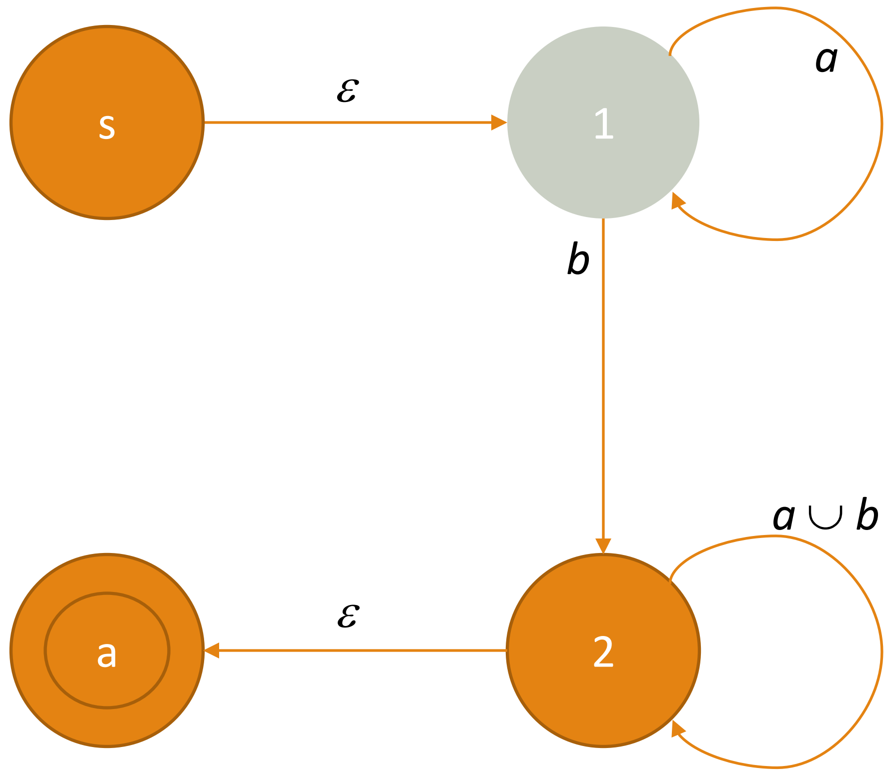
- We need to repair that transition
- The concatenation is obvious
- Can you see why we need the star closure?

One more state, and we're done



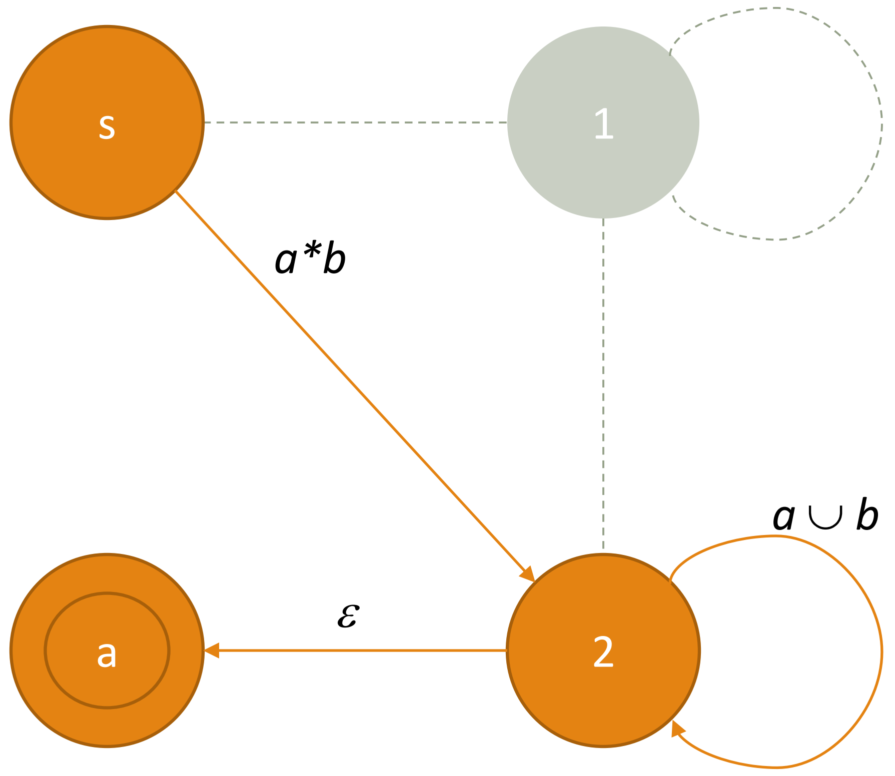
Making the GNFA

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



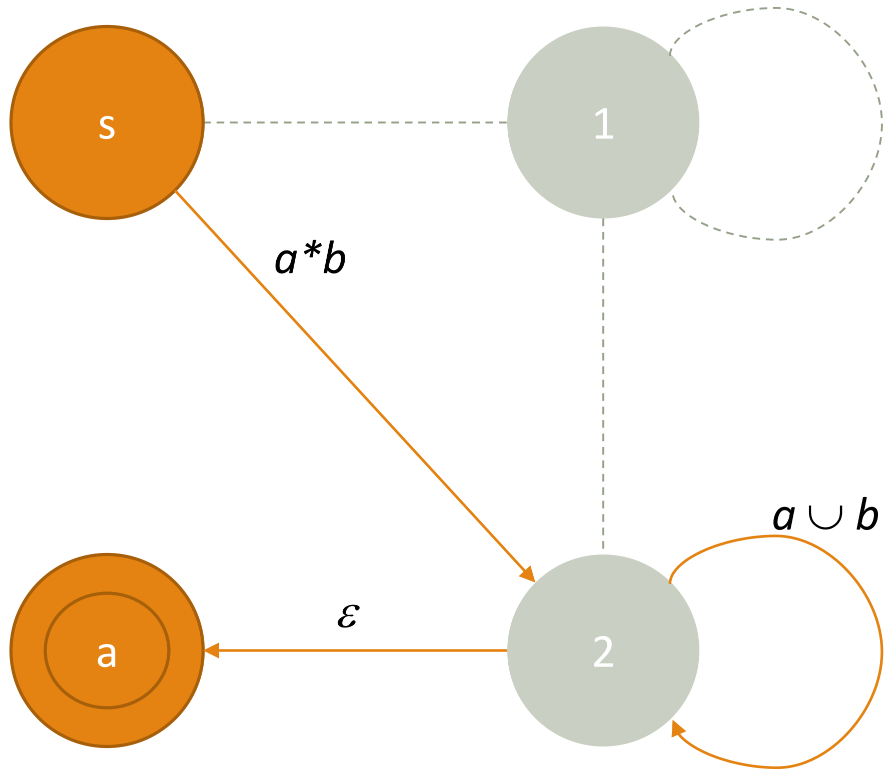
Making the GNFA

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



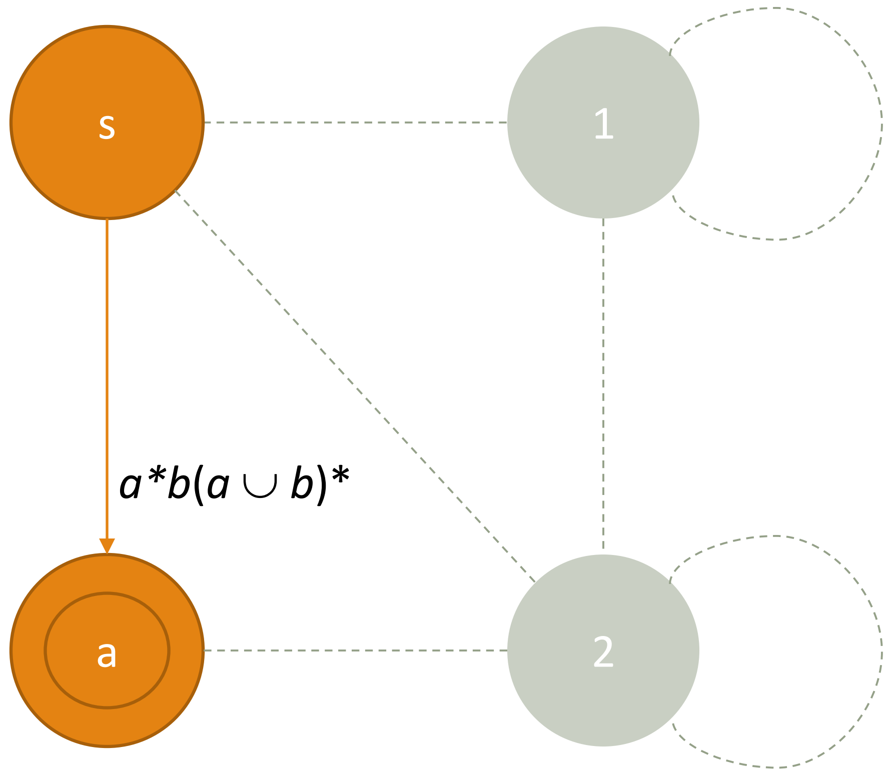
Making the GNFA

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



More on GNFAs

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

Next Time: GNFAs Formally and Non-Regular Languages