CSE396 Spr2026 Lecture Thu. Week 4: Completing the Cycle of Regular Languages

Kleene's Equivalence Theorem gives us multiple characterizations of the **class** of regular languages (over any designated alphabet Σ). We will denote classes of languages in bold green sans-serif capitals. So we call it **REG**. To complete the cycle, we resort to a characterization that mashes together the ideas of "regular epression" and "finite automaton."

Generalized NFAs (GNFAs)

I view these as mathematical bookkeeping devices, not as "real" as NFAs, let alone DFAs. The meaning of an arc from a state p to a state q is "all ways we know so far to get from state p to state q, in however many steps." By getting from state p to state q we mean a string processed along the way, so our notation $L_{p,q} = \{w \in \Sigma^* : N \text{ can process } w \text{ from } p \text{ to } q\}$ comes into play. We will see that this language is always regular "in the final analysis." Moreover, insofar as ϵ or one single character a or b (etc.) "is-a" basic regular expression, we start off with arcs labeled by regular expressions with an NFA anyway. And a loop or edge labeled a, b could really be the non-basic regular expression $a \cup b$ anyway. So let us generalize arcs to any regular expressions over the alphabet Σ, letting Regexp(Σ) stand for that.

Definition: A generalized NFA (GNFA) is a 5-tuple $G = (Q, \Sigma, \delta, s, F)$ where Q, Σ, s, F are the same as in an NFA but now $\delta \subseteq (Q \times \text{Regexp}(\Sigma)) \times Q$.

Definition: A sequence $\vec{c} = [q_0, u_1, q_1][q_1, u_2, q_2][q_2, u_3, q_3] \cdots [q_{t-2}, u_{t-1}q_{t-1}][q_{t-1}, u_t, q_t]$ is a **valid computation** if for each i, $1 \leq i \leq t$, there is an instruction $(q_{i-1}, \alpha, q_i) \in \delta$ such that the string u_i matches the regular expression α. The **string processed** by the computation is the string $u = u_1 \cdot u_2 \cdot u_3 \cdots u_{t-1} \cdot u_t$, which might be shorter or longer than t. Then we say the GNFA G can process u from q_0 to q_t and write $u \in L_{q_0,q_t}$ (with G understood). Then as before,

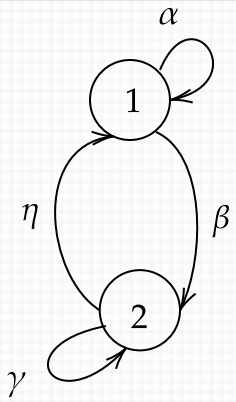
$$L(G) = \bigcup_{f \in F} L_{s,f}.$$

Simply and stupidly, if we have a regular expression β, then we can make a GNFA G such that $L(G) = L(\beta)$ with just two states and one arc. If β falls into the stare case, i.e., if $\beta = \alpha^*$ for some regular expression α, then we need just one state and one self-loop:

The text structures its proof of the FA-to-regexp step with a "preamble" (of adding extra start and final states) so that the left-hand machine G results at the end. If the start state is also the only accepting state in the given FA (whether DFA or NFA), you can actually avoid the preamble and get the one-state

machine G_0 at right as the "ultimate base case." **However**, we can often save lots of messy work by using the *general* two-state machine as the base case.

The idea can be put across intuitively even when we use abstract regular expressions $\alpha, \beta, \gamma, \eta$ (alpha, beta, gamma, **eta**).



$$L_{1,1} = (\alpha + \beta\gamma^*\eta)^*$$

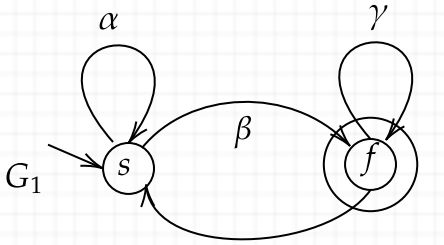
$$L_{1,2} = L_{1,1} \cdot \beta\gamma^* = (\alpha + \beta\gamma^*\eta)^*\beta\gamma^*$$

$$L_{2,2} = (\gamma + \eta\alpha^*\beta)^*$$

$$L_{2,1} = L_{2,2} \cdot \eta\alpha^* = (\gamma + \eta\alpha^*\beta)^*\eta\alpha^*$$

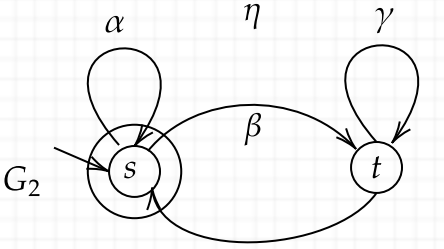
$L_{1,2}$ also equals $\alpha^*\beta \cdot L_{2,2} = \alpha^*\beta(\gamma + \eta\alpha^*\beta)^*$.

The three base cases that arise depend on whether s is accepting and whether there is an accepting state other than s. If there are *two or more* accepting states different from s, that's the only time you need to do the text's extra step of adding a new final state f and ϵ-arcs from all the old final states. (And you *never* have to do the step of adding a new start state.)

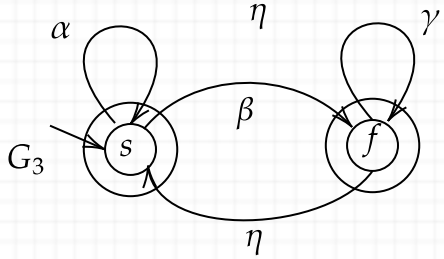


$$L(G_1) = L_{sf} = (\alpha + \beta\gamma^*\eta)^*\beta\gamma^* = \alpha^*\beta(\gamma + \eta\alpha^*\beta)^*$$

η is pronounced "ate-a" in the US, "eat-a" in the UK.



$$L(G_2) = L_{ss} = (\alpha + \beta\gamma^*\eta)^*$$



$$L(G_3) = L_{ss} \cup L_{sf}$$

Note that in G_2, we called the second state t rather than f because it is not accepting. We could boil G_2 down all the way to G_0 by "eliminating" the non-accepting state t, but there is no need. The virtue of understanding the general two-state cases is that it helps with the general case of eliminating a non-accepting state q.

Note that if $\alpha, \beta, \gamma, \eta$ are all basic regular expressions, then we get an NFA. Well, if (say) $\eta = \emptyset$, then that's the same as the back-arc from f to s not existing at all, since $\emptyset$ allows no way to pass through. The definition of accepting computation then "drops down" top being what we defined before. Thus we can say that a DFA or NFA "**Is-A**" GNFA too.

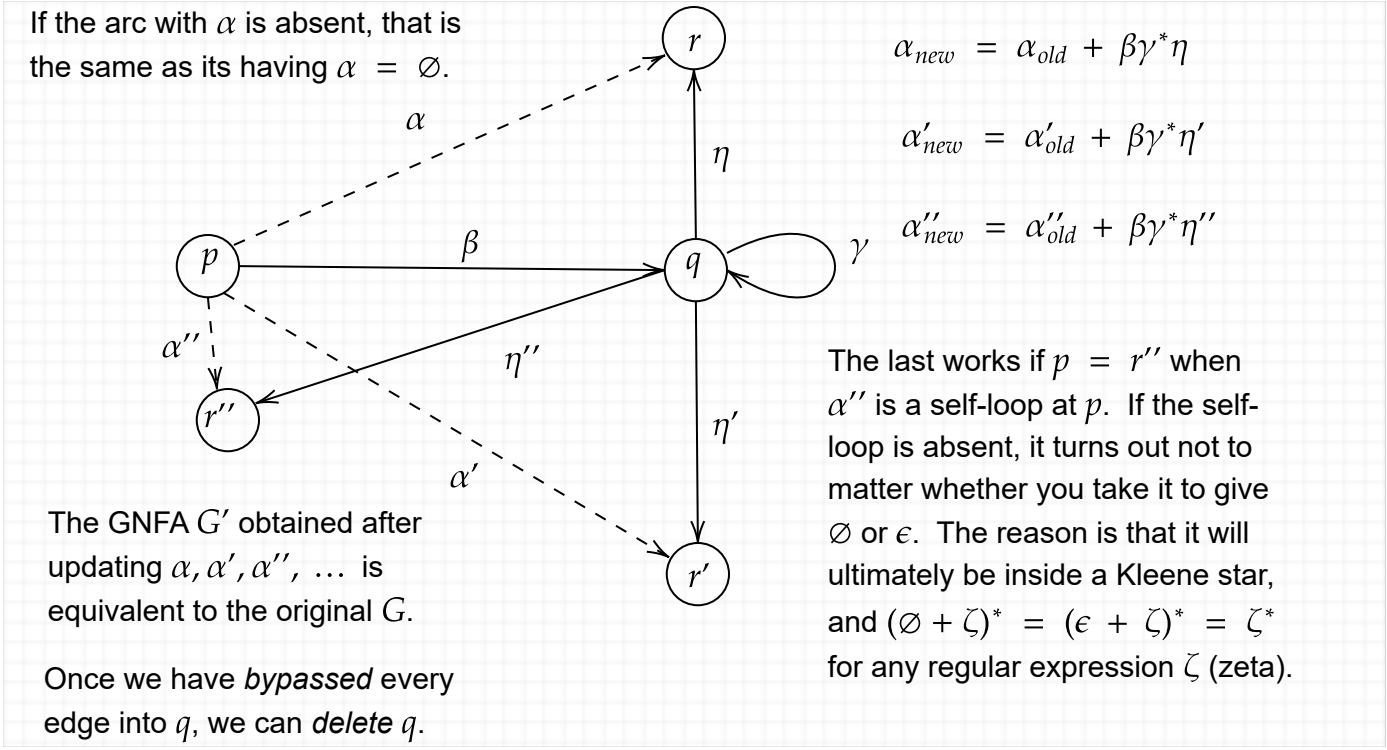
General GNFA Case

Here is the algorithm for converting an n-state automaton (of any kind) into an equivalent regular expression:

1. Subcases:
   (a) If there are two or more accepting states besides the start state s, make a new accepting state f with ϵ-arcs from the old ones. Then you'll be in case (b) but with "n" incremented by 1 (which makes more work).
   (b) If there is exactly one accepting state f other than s, number $s = 1$, $f = 2$, and the other states $3, \ldots, n$. It does not matter whether s is accepting too.
   (c) If s is the only accepting state, then number $s = 1$ and choose an arbitrary other state to get numbered 2.
2. All of the states $q = 3, \ldots, n$ are nonaccepting states. Looping from n down to 3, we will **eliminate** them one-by-one, until one of the above two-state base cases is left.
3. If we were given an NFA or DFA, we reinterpret it as a GNFA with basic regular expressions on the edges. Non-edges implicitly have $\emptyset$ as their label.
4. As we eliminate states, we update the regular expressions on edges between states that still exist. Those expressions will become non-basic (and may get bushy).
5. Eventually only states 1 and 2 are left, and we read off the answer as above.

The idea of eliminating a state q that is not the start state and is non-final is that *no accepting computation can begin or end at q.* Hence, if an accepting computation enters q from some state p, then it must exit at some state r (which can be the same as p).

Considering multiple such states r, r', r'' gives us the following diagram:



That's the entire algorithm: we delete such states q one-by-one until only two states are left.

Symbolic Implementation

If we imagine ourselves programming this with a `RegExp` package, then we can represent a given n-state finite automaton (DFA, NFA, or GNFA, all the same to start with) by an $n \times n$ matrix T of `RegExp`. If we have two or more accepting states other than the start state, then we do have to do the preamble with an extra final state---but in any event, we get to the situation where states 3 to n are all nonaccepting states. Then the main code is simply:

```
for k = n downto 3:
    for i = 1 to k-1:
        for j = 1 to k-1:
            T(i,j) += T(i,k) · T(k,k)* · T(k,j).
```

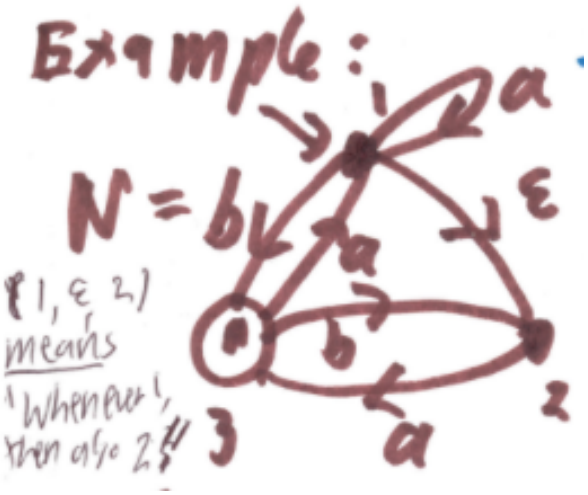
(The convenience of writing "+=" here is one reason I like using $+$ rather than $\cup$ for union.) Note that even if there is no self-loop at q, so that $T(k,k) = \varnothing$ (or ϵ; it doesn't matter), the update is not killed because $T(k,k)^* = \epsilon$. But if there is no arc from i into k, that is, if $T(i,k) = \varnothing$, then the right-hand side does get nulled and the update is simply a no-op. Likewise if no arc from k out to j, whereupon $T(k,j) = \varnothing$.

That's it. Anyway, what we have proved is:

Theorem. Given any DFA, NFA, or GNFA G, we can calculate a regular expression ρ (Greek rho) such that $L(\rho) = L(G)$.

This also completes the proof of the final part of Kleene's Equivalence Theorem.

Example---revisiting a previous NFA:



We want to eliminate state 2. If we were using the code approach, we could re-number it as state 3. But we can also do it "graphically": list the "In"coming and "Out"going arcs and update all combinations of them. Here we have:

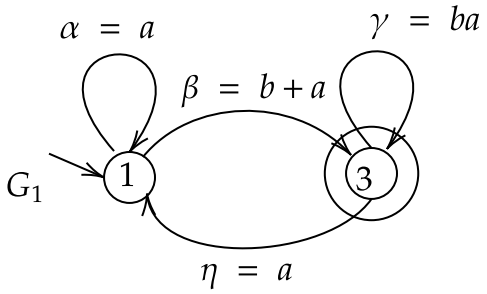
In: 1 (on ϵ) and 3 (on b).
Out: only to 3 (on a).
Update: $T(1,3)$ and $T(3,3)$.

$$\begin{aligned} T(1,3)_{new} &= T(1,3)_{old} + T(1,2)T(2,2)^*T(2,3) \\ &= b + \epsilon \cdot \epsilon \cdot a = b + a. \\ T(3,3)_{new} &= T(3,3)_{old} + T(3,2)T(2,2)^*T(2,3) \\ &= \varnothing + b \cdot \epsilon \cdot a = ba. \end{aligned}$$

[Suppose we try to update $T(3,1)$. The rule would be

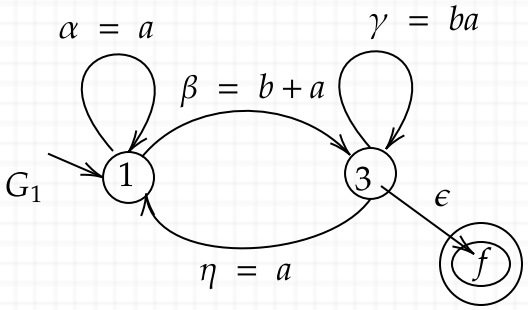
$$\begin{aligned} T(3,1)_{new} &= T(3,1)_{old} + T(3,2)T(2,2)^*T(2,1) \\ &= a + b \cdot \epsilon \cdot \varnothing \quad \text{because there is no arc from 2 to 1.} \\ &= a + \varnothing = a, \text{ which is no change from } T(3,1)_{old}.] \end{aligned}$$

The new GNFA is



$$L(G_1) = L_{sf} = \left(\alpha + \beta\gamma^*\eta\right)^*\beta\gamma^*$$

$$= (a + \left((b+a)(ba)^*a\right)^*(b+a)(ba)^*.$$



$$L(G_1) = L_{sf} = \left(\alpha + \beta\gamma^*\eta\right)^*\beta\gamma^*$$

$$= (a + \left((b+a)(ba)^*a\right)^*(b+a)(ba)^*.$$