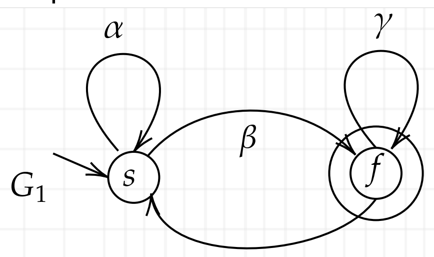
CSE396 Lecture Thu. 2/25: FA-to-Regexp Proof and Algorithm; Regular Languages

Three possibilities for the base-case two-state GNFAs:

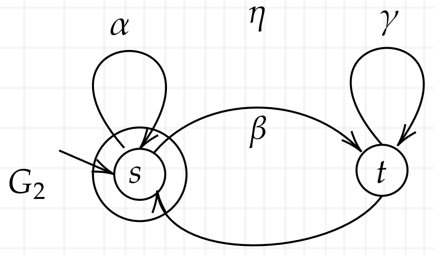


$$L_{sf} = L_{ss} \cdot \text{"}s \text{ to } f \text{ once"} = \text{"}s \text{ to } f \text{ once"} \cdot L_{ff}$$

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

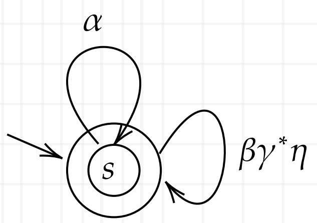
$$\textit{Equivalently, it} = \alpha^*\beta\gamma^*(\gamma + \eta\alpha^*\beta)^*$$

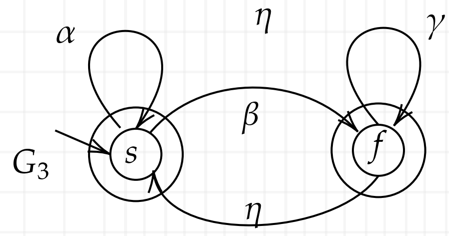
η is pronounced "ate-a" in the US, "eat-a" in the UK.



$$L_{ss} = (\alpha \cup \beta\gamma^*\eta)^*$$

$$L_{ff} = (\gamma \cup \eta\alpha^*\beta)^*$$

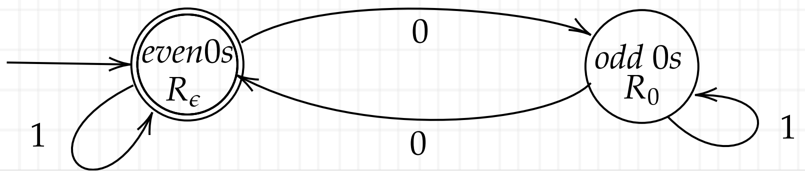




$$L(G_2) = L_{ss} = (\alpha + \beta\gamma^*\eta)^*$$

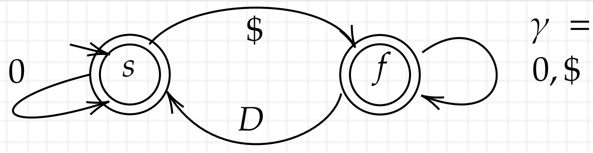
$$L(G_3) = L_{ss} \cup L_{sf}$$

Example HW has something like this (but with accept and reject states and 0 and 1 both switched):



Let's instead do:

N:



$\gamma = (0 \cup \$)$ $\quad x = \$DD$

$x \in L_{s,dead}$ but $dead$ is not in F so x is not in the language.

$$L_{ss} = (0 \cup \$(0 \cup \$)^* D)^*$$

$L_{sf} = L_{ss} \cdot (\textit{ways to go from } s \textit{ to } f \textit{ without going back to } s) = L_{ss} \cdot \$(0 \cup \$)^*$

$$L(N) = L_{ss} \cup L_{sf} = L_{ss} + L_{ss} \cdot \$(0+\$)^* = L_{ss} \cdot (\epsilon + \$(0+\$)^*)$$

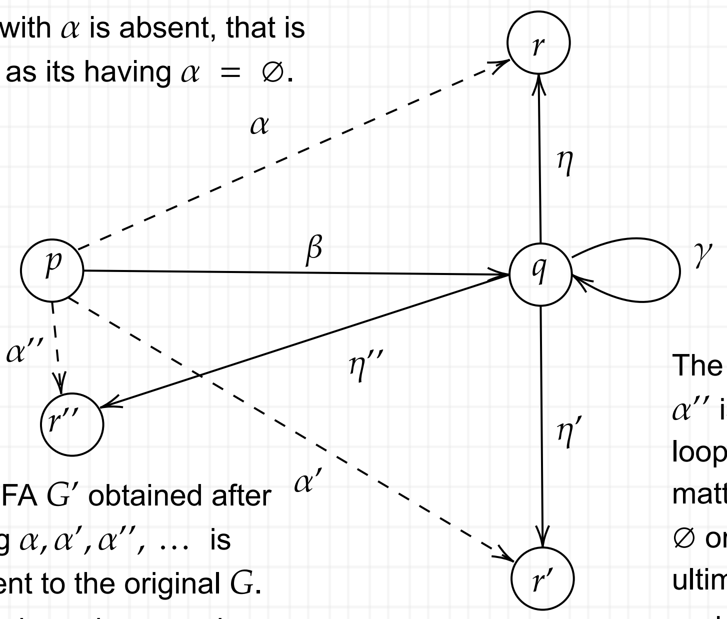
$$L(N) = L_{s.s} \cup L_{s,f} = (0 + \$(0+\$)^*D)^*(\epsilon + \$(0+\$)^*)$$

Without the dead state and arc to it, the NFA N on input $x = \$DD$ would "crash" in state s. Even though s is an accepting state (and even though this would count as legal termination by a Turing machine), not all of x would be processed, so it does not count in the FA's language. With the dead state present, x gets processed to $dead$, but $dead \notin F$ so $x \notin L(N)$ still.

Note that in G_2, we called the second state t rather than f because it is not accepting. No accepting computation can begin or end at a non-final state q that is different from the start state. Hence, if the 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:

General GNFA State Elimination Case:

If the arc with α is absent, that is the same as its having $\alpha = \emptyset$.



$$\alpha_{new} = \alpha_{old} + \beta\gamma^*\eta$$

$$\alpha'_{new} = \alpha'_{old} + \beta\gamma^*\eta'$$

$$\alpha''_{new} = \alpha''_{old} + \beta\gamma^*\eta''$$

The last works if $p = r''$ when α'' is a self-loop at p. If the self-loop is absent, it turns out not to matter whether you take it to give $\emptyset$ or ϵ. The reason is that it will ultimately be inside a Kleene star, and $(\emptyset + \zeta)^* = (\epsilon + \zeta)^* = \zeta^*$ for any regular expression ζ (zeta).

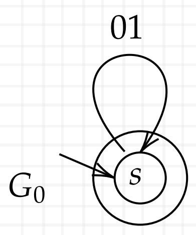
The GNFA G' obtained after updating $\alpha, \alpha', \alpha'', \ldots$ is equivalent to the original G. Once we have *bypassed* every edge into q, we can *delete* q.

If we are 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`. We can number the non-accepting states different from the start state by $m, \ldots, n$ for whatever m applies. (If start is the only accepting state then we could take m as low as 2, but it saves "mess" to take $m = 3$ in this case too so that execution will end with G_2 above, at which point the answer can be shortcutted by saying what $\alpha, \beta, \gamma, \eta$ are and citing the abstract formula. Most sources say to add a new start state and make all original final states go to a new one, but while doing this makes the proof look neater, it is more work that is highly typo-prone.) Then let one loop variable k run over the nodes q to be eliminated, let i run over all states up to $k-1$ which are treated as possible entry states p, and let j run over potential exist states r. Then the main code is simply:

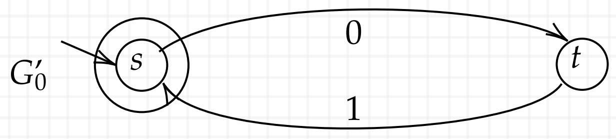
```
for k = n downto m:
    for i = 1 to k-1:
        for j = 1 to k-1:
            T(i,j) = T(i,j) + T(i,k) · T(k,k)* · T(k,j).
```

(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 $T(k,k) = \emptyset$ (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 $T(i,k) = \emptyset$, 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 $T(k,j) = \emptyset$.

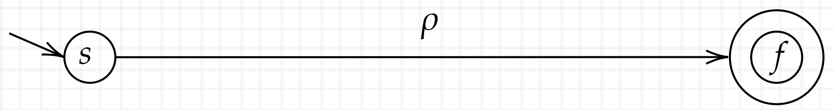
The result of executing the code is a GNFA G' with all states accepting except possibly the start state. If the start state, too, is accepting, it is tempting to think $L(G') = \Sigma$, i.e., that G' accepts all strings, but that is not true because GNFA arcs can have "holes" that prevent matching and hence processing all strings. For example, consider the simple one-state GNFA



Then $L(G_0) = (01)^*$ but this is not all strings. The reason is that G_0 was really abbreviating the NFA shown below, which can "crash" on 1 at its start state and on 0 at state t:



So if you get a G' with two or more accepting states different from the start state, then you do have to add a new final state f with arcs from all the old final states, declare f to be the only final state, and eliminate all of the previous accepting states apart from s. If you also make s a new, non-accepting state, then you do get the final answer $\rho = L(G)$ "on a silver platter":



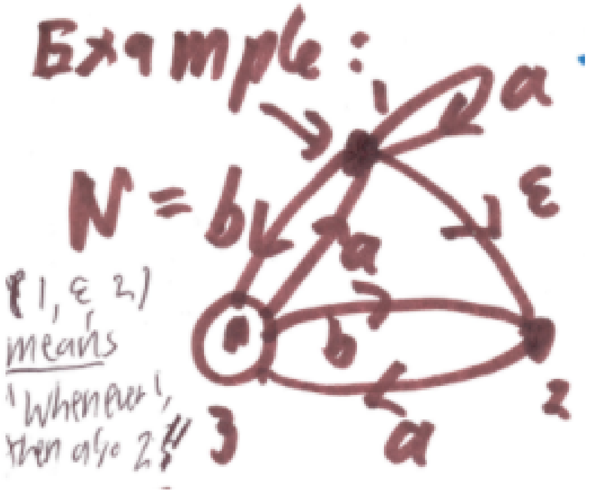
But the final expression ρ you get is often quite long, and the steps for the last one or two states you eliminated often amount to hand-copying long subexpressions corresponding to $\alpha, \beta, \gamma, \eta$ in the above formulas for the 2-state GNFAs anyway. The ground rules are hence that once you get down to two states, you can just cite the abstract formula to say what the final regular expression will be. And if the originally given GNFA has at most one accpeting state besides the start state, then the above code body will give your final answer without needing to add a new final state. Why add one or two iterations to the outside of a triply-nested loop if you can avoid 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 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 $T(3,3)$.

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

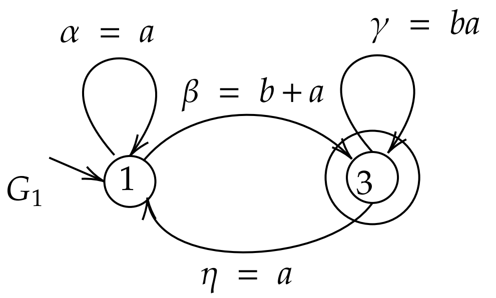
Could also say

$= \epsilon + b\cdot\epsilon\cdot a = \epsilon + ba$ ---which is different! Will we get this wrong?

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

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

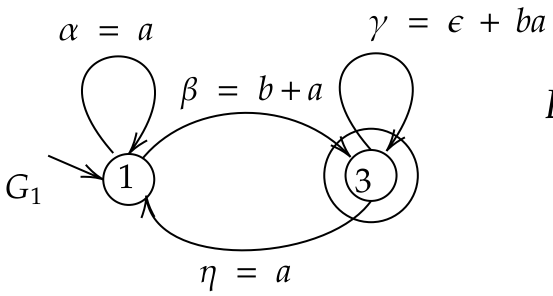
The new GNFA is



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

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

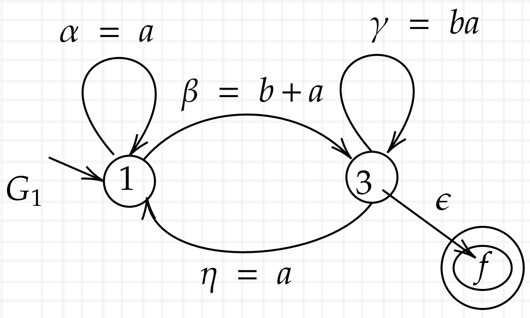
$$Fact: \gamma^* = (\gamma + \epsilon)^* = (\gamma + \epsilon)^+ \textit{ for any } \gamma$$



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

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

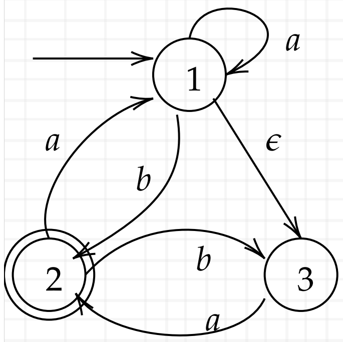
If you add a new final state like the text says to do, you get this:



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

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

Here is the same example "in code" showing the T-matrix but with the states renumbered.



Initial T-matrix

$$\left[\begin{array}{c|ccc} T & 1 & 2 & 3 \\ \hline 1 & a & b & \epsilon \\ 2 & a & \varnothing & b \\ 3 & \varnothing & a & \varnothing \end{array}\right]$$

The green a on the main diagonal can be $a + \epsilon$ and the green $\varnothing$ entries can be changed to ϵ too without throwing off the final answer. But the red $\varnothing$ off the main diagonal must stay as $\varnothing$.

To eliminate state 3:
In from 1 and 2, out to 2 only.
Hence update $T(1,2)$ & $T(2,2)$.
$T(1,2)$ += $T(1,3)T(3,3)^*T(3,2)$, so
new $T(1,2) = b + \epsilon\varnothing^*a = b + a$
$T(2,2)$ += $T(2,3)T(3,3)^*T(3,2)$, so
new $T(2,2) = \varnothing + b\varnothing^*a = ba$

New T-matrix = $\begin{bmatrix} a & b+a \\ a & ba \end{bmatrix}$

Now we are in the same 2-state case, except with the final state being numbered 2 rather than 3. It really comes out the same. The reason having $T(1,1) = a + \epsilon$ would not be wrong is that the diagonal entries $T(1,1)$ etc. all get starred, and $(a + \textit{whatever})^* = (a + \epsilon + \textit{whatever})^*$.