

## Lecture Friday 16 Sept. 2022: Generalized NFAs (GNFAs):

A *generalized NFA*  $G$  can have any regular expression on any arc. The "instructions" are triples  $(p, \alpha, q)$  where  $p$  and  $q$  are states as before and  $\alpha$  belongs to  $\text{Regexp}(\Sigma)$ , by which is meant the set of regular expressions over the alphabet  $\Sigma$ . Note that character arcs and  $\epsilon$ -arcs can both be regarded as special cases of this, so an NFA "is-A" GNFA. Thus in object-oriented terms, a GNFA is a more-basic concept than an NFA. **A little philosophical note:** I believe that NFAs are "real" but GNFAs are not--they are IMHO just bookkeeping devices. Hence the arched-eyebrow quotes above and below.

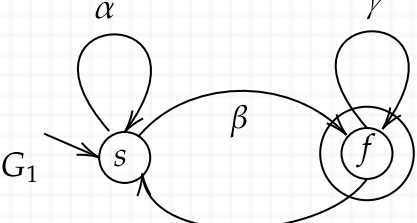
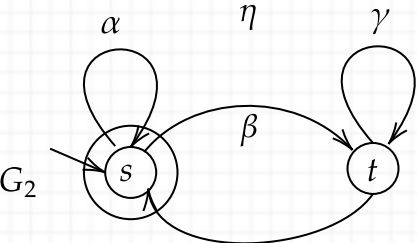
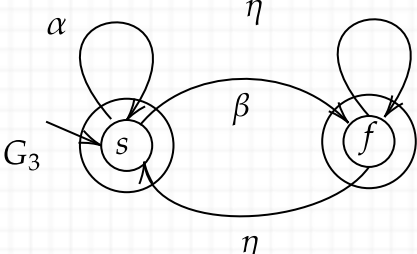
A string  $x$  is "accepted" by  $G$  if  $x$  can be broken into some number  $m$  of substrings such that each substring matches the respective regexp in a path of  $m$  arcs of  $G$  that begins at  $s$  and ends in a final state  $f$ . That is to say:

**Definition:** A "computation" by a GNFA  $G = (Q, \Sigma, \delta, s, F)$ , where  $\delta \subseteq Q \times \text{Regexp}(\Sigma) \times Q$ , that processes a string  $x$  from some state  $p$  to some state  $r$ , is a sequence

$$(q_0, u_1, q_1), (q_1, u_2, q_2), \dots, (q_{m-1}, u_m, q_m)$$

such that  $x = u_1 \cdot u_2 \cdots u_m$ ,  $q_0 = p$ ,  $q_m = r$ , and for each  $i$ ,  $1 \leq i \leq m$ , there is an instruction  $(q_{i-1}, \alpha, q_i) \in \delta$  such that  $u_i$  matches  $\alpha$ . Then  $x \in L_{p,r}$  and  $L(G) = \cup_{f \in F} L_{s,f}$  as before.

The real essence is that the regular expressions can be manipulated **as if they were characters**. In particular, we can apply the logic we used for the parity-DFA case to write down the processing languages for any two-state GNFA:

	$L(G_1) = L_{sf} = (\alpha + \beta\gamma^*\eta)^*\beta\gamma^*$ $= \alpha^*\beta(\gamma + \eta\alpha^*\beta)^*$ <p><math>\eta</math> is pronounced "ate-a" in the US, "eat-a" in the UK.</p>
	$L(G_2) = L_{ss} = (\alpha + \beta\gamma^*\eta)^*$
	$L(G_3) = L_{ss} \cup L_{sf}$ $= (\alpha + \beta\gamma^*\eta)^*(\epsilon + \beta\gamma^*)$

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 Case:

If the arc with  $\alpha$  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  $\alpha''$  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  $\epsilon$ . The reason is that it will ultimately be inside a Kleene star, and  $(\emptyset + \zeta)^* = (\epsilon + \zeta)^* = \zeta^*$  for any regular expression  $\zeta$  (zeta).

The GNFA  $G'$  obtained after updating  $\alpha, \alpha', \alpha'', \dots$  is equivalent to the original  $G$ .

Once we have *bypassed* every edge into  $q$ , we can *delete*  $q$ .

Understanding how this diagram enables one to pluck out non-accepting states  $q$  is the bulk of the proof that any finite automaton  $N$  can be converted into a regular expression for its language.

1. If  $N$  has at most one accepting state different from start, then we can just eliminate the non-accepting states and things will land in one of the above two base cases.
2. If  $N$  has two or more accepting states different from start, then we (in general) need to add a single new accepting state  $f$ , put  $\epsilon$ -arcs from the previous accepting states, and make the previous states nonaccepting. Then we are in the good case 1, but we have extra work of dealing with one more state, and  $\epsilon$ -arcs are error-prone. (You can postpone this until you've eliminated all the *non*accepting states different from start.)
3. Some sources say to make a new start state as well, but that is just so they can always get down to a single-arc base case to show off. This makes unnecessary extra work; don't do it.

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  $m, \dots, n$  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 exit states  $r$ . Then the main code is simply:

for  $k = n$  downto 3:

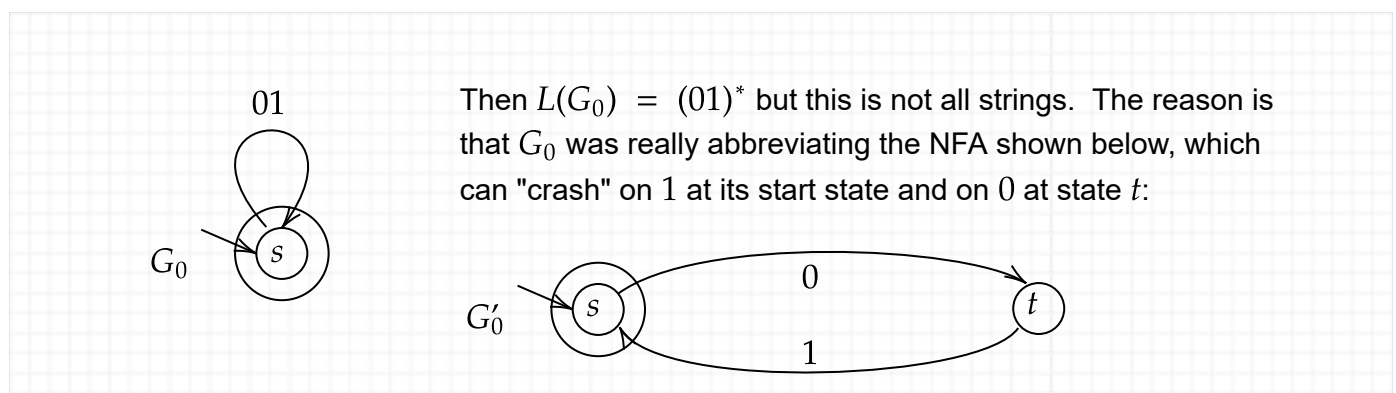
  for  $i = 1$  to  $k-1$ :

    for  $j = 1$  to  $k-1$ :

$$T(i,j)_{\text{new}} = T(i,j)_{\text{old}} + T(i,k) \cdot T(k,k)^* \cdot T(k,j).$$

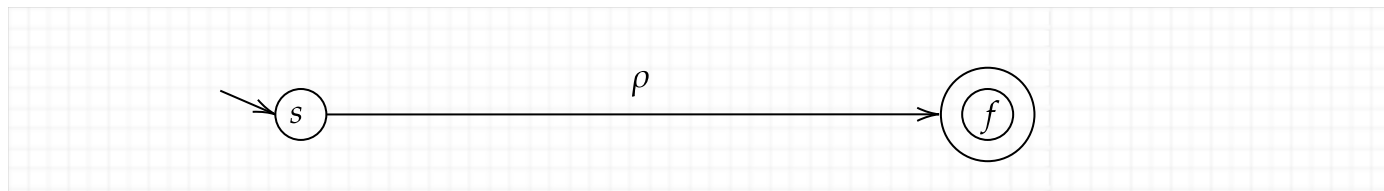
(The convenience of writing "+=" here is one reason I like using  $+$  rather than  $\cup$  for union.) Note that even if there is no self-loop at  $q$ , so that  $T(k,k) = \emptyset$  (or  $\epsilon$ ; it doesn't matter), the update is not killed because  $T(k,k)^* = \epsilon$ . But if there is no arc 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



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  $\epsilon$ -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":



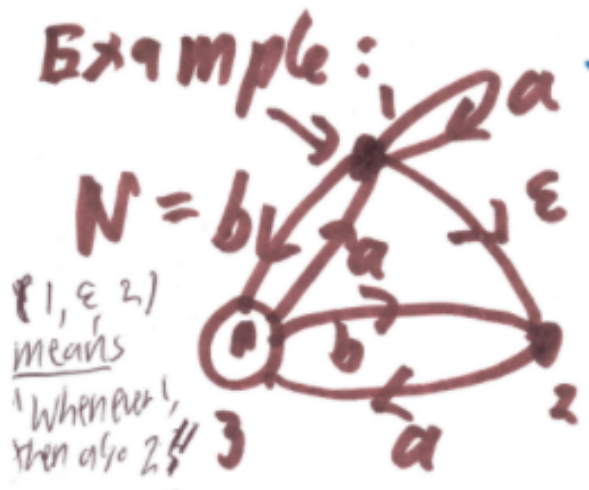
But the final expression  $\rho$  you get is often quite long, and the steps for the last one or two states you eliminated often amount to hand-copying long subexpressions corresponding to  $\alpha, \beta, \gamma, \eta$  in the above formulas for the 2-state GNFA's anyway. The ground rules are hence 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 accepting 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  $\rho$  (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  $\epsilon$ ) and 3 (on  $b$ ).

Out: only to 3 (on  $a$ ).

Update:  $T(1,3)$  and  $T(3,3)$ .

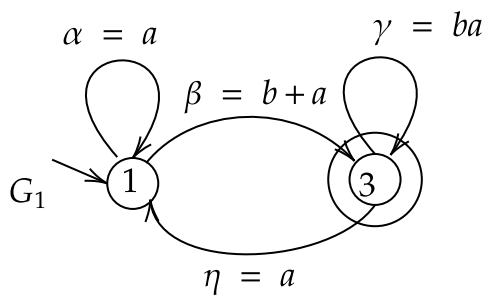
$$\begin{aligned} T(1,3)_{new} &= T(1,3)_{old} + T(1,2)T(2,2)^*T(2,3) \\ &= b + \epsilon \cdot \epsilon \cdot a = b + a. \end{aligned}$$

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

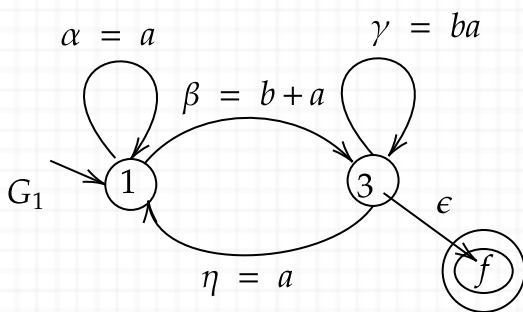
[Suppose we try to update  $T(3,1)$ . The rule would be

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

The new GNFA is



$$\begin{aligned} L(G_1) &= L_{sf} = (\alpha + \beta\gamma^*\eta)^* \beta\gamma^* \\ &= (a + (b+a)(ba)^*a)^* (b+a)(ba)^*. \end{aligned}$$



$$\begin{aligned} L(G_1) &= L_{sf} = (\alpha + \beta\gamma^*\eta)^* \beta\gamma^* \\ &= (a + ((b+a)(ba)^*a)^* (b+a)(ba)^*. \end{aligned}$$