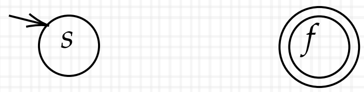
CSE396 Lecture Tue. 2/16: From Regular Expressions To NFAs

HW1 due today, 11:59pm. I will have office hour **10pm--11pm** this evening for last-minute Qs. My remaining office hours still TBA: last call for Survey sheets (only 23 received so far).

NFAs have $N = (Q, \Sigma, \delta, s, F)$ with $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

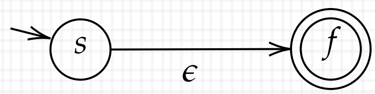
Regular Expressions and Their Corresponding NFAs (with ϵ-transitions):

(B1) $\emptyset$ is a regexp; $L(\emptyset) = \emptyset$; $N_{\emptyset} =$



$(\delta = \emptyset)$

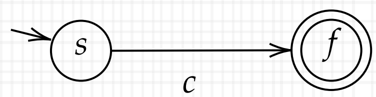
(B2) ϵ is a regexp; $L(\epsilon) = \{\epsilon\}$; $N_{\epsilon} =$



δ has (s, ϵ, f)

For all chars $c \in \Sigma$:

(B3) $\mathbf{c}$ is a regexp; $L(\mathbf{c}) = \{c\}$; $N_{\mathbf{c}} =$



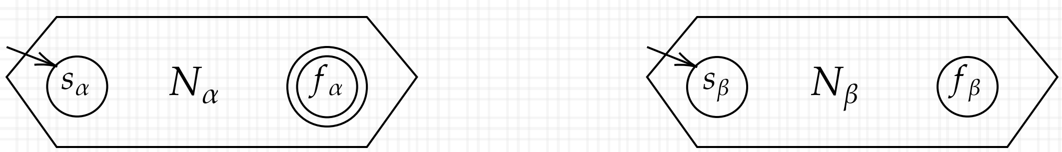
δ has (s, c, f)

This completes the *basis* of an *inductive definition* of regular expressions. Now let α and β be any two regular expressions, with languages $A = L(\alpha)$ and $B = L(\beta)$. By *inductive hypothesis* (**IH**) we have NFAs N_α and N_β such that $L(N_\alpha) = A$ and $L(N_\beta) = B$. Then:

(I1) $\gamma = \alpha \cup \beta$ is a regexp; $L(\gamma) = A \cup B$. **TopHat 5565**

gamma alpha beta

Now to complete the *induction case* (I1) we need to show how to build an NFA$_\epsilon$ N_γ such that $L(N_\gamma) = L(\gamma)$. What we have to work with is (are) N_α and N_β. We know they have start states we can call s_α and s_β. Taking a cue from the base case NFAs, and mainly for convenience, we may suppose they have unique accepting states f_α and f_β. Besides that, we make no assumptions about their internal structure, so we draw them as "blobs":



The goal is to connect them together to make N_γ with needed properties, also for the cases:

(I2) $\gamma = \alpha \cdot \beta$ is a regexp; $L(\gamma) = A \cdot B$.

(I3) $\gamma = \alpha^*$ is a regexp; $L(\gamma) = A^*$. (In I3 we have only N_α given.)

$\mathit{This\ does:}\ L(N_\gamma) \ = \ L(N_\alpha) \ \cup \ L(N_\beta)$

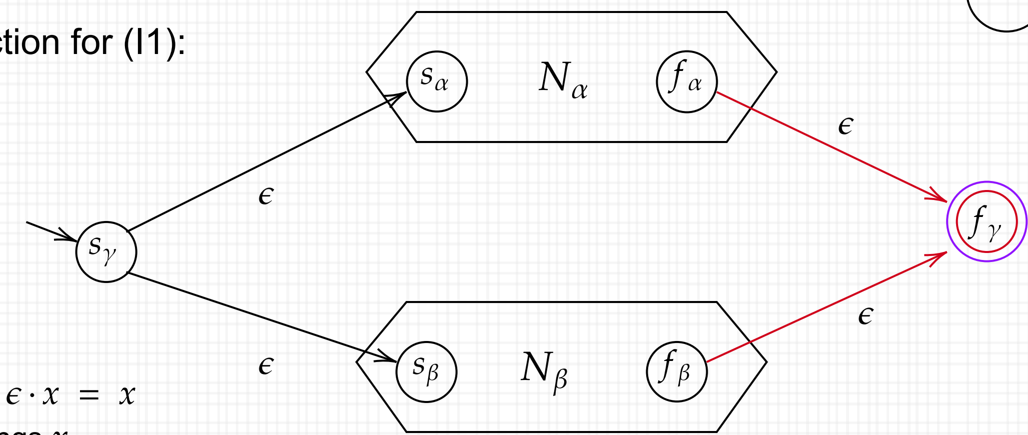
$\qquad\qquad\qquad\qquad\qquad || \qquad\quad ||$

$\qquad\qquad\qquad\qquad\qquad L(\alpha) \ \cup \ L(\beta)$

$\mathit{Target:}\ L(\gamma) \ = \ L(\alpha) \ \cup \ L(\beta)$

Construction for (I1):

$N_\gamma \ =$



note rule: $\epsilon \cdot x \ = \ x$
for all strings x.

This builds N_γ, but we still need to prove it is correct, i.e., $L(N_\gamma) \ = \ L(\gamma)$. *Note the rhythm:*

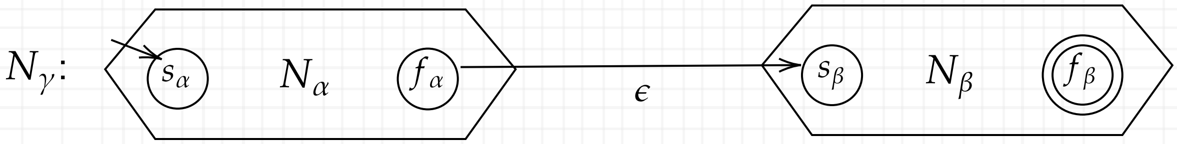
1. $L(N_\gamma) \ = \ L(N_\alpha) \ \cup \ L(N_\beta)$ by machine construction;
2. $L(N_\alpha) \ = \ L(\alpha)$ and $L(N_\beta) \ = \ L(\beta)$ by inductive hypothesis;
3. Thus $L(N_\gamma) \ = \ L(\alpha) \ \cup \ L(\beta) \ = \ L(\alpha \cup \beta) \ = \ L(\gamma)$ by definition of γ.

[I will continue as time permits by copy-and-paste and moving things around to do the other two inductive cases to complete the proof. But first, are you completely happy with N_γ as it stands?]

[Answer was *no*: adding the state f_γ and ϵ-arcs shown in red "preserves the invariant" of the NFAs all having a single accepting state.]

(I2) $\gamma = \alpha \cdot \beta$ is a regexp; $L(\gamma) = A \cdot B = \{xy : x \in A \wedge y \in B\}$.

$L(\gamma) = L(\alpha) \cdot L(\beta)$



Then $L(N_\gamma) = L(N_\alpha) \cdot L(N_\beta)$ because....processing....

To write the reasoning out: N_γ can process a string z from its start state $s_\gamma = s_\alpha$ to its (unique) final state $f_\gamma = f_\beta$ if and only if z has a first part x that gets processed from s_α to f_α and a second part y that gets processed from s_β to f_β (with the ϵ from f_α to s_β silently in-between). I.e.: $z \in L(N_\gamma) \iff z \in \{x \cdot y : x \in L(N_\alpha) \wedge y \in L(N_\beta)\} \iff z \in L(N_\alpha) \cdot L(N_\beta)$. Thus $L(N_\gamma) = L(N_\alpha) \cdot L(N_\beta)$.

By **IH**, this equals $L(\alpha) \cdot L(\beta)$, which by how the semantics of $\gamma = \alpha \cdot \beta$ is defined via $L(\gamma) = L(\alpha) \cdot L(\beta)$ finally gives us the needed conclusion $L(N_\gamma) = L(\gamma)$.

Now back to our recursive construction of regular expressions and NFAs corresponding to them. This proves one part of a theorem discovered by Stephen Kleene in the 1950s.

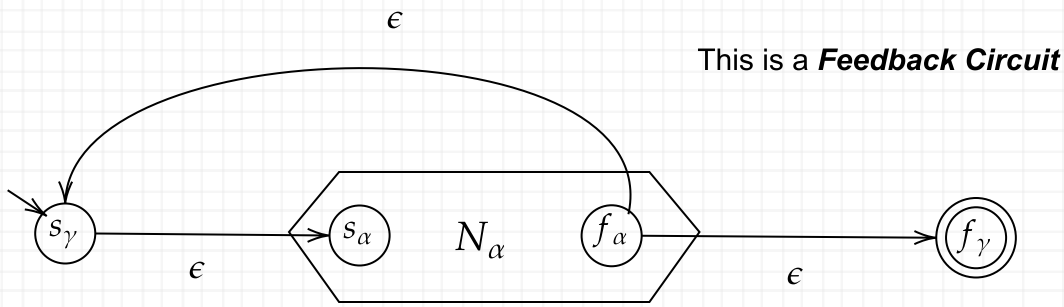
Theorem: For any language A over an alphabet Σ, the following statements are equivalent:

1. There is a regular expression α such that $A = L(\alpha)$.
2. There is an NFA N such that $A = L(N)$.
3. There is a DFA M such that $A = L(M)$.

We are in the middle of proving 1 $\implies$ 2. Next will be 2 $\implies$ 3. Then 3 $\implies$ 1 would "complete the cycle of equivalence" but in fact we will use something more general than an NFA to go to 1.

(I3) Given any regexp α, $\gamma = \alpha^*$ is a regexp; $L(\gamma) = L(\alpha)^*$; and we can build:

$N_\gamma =$

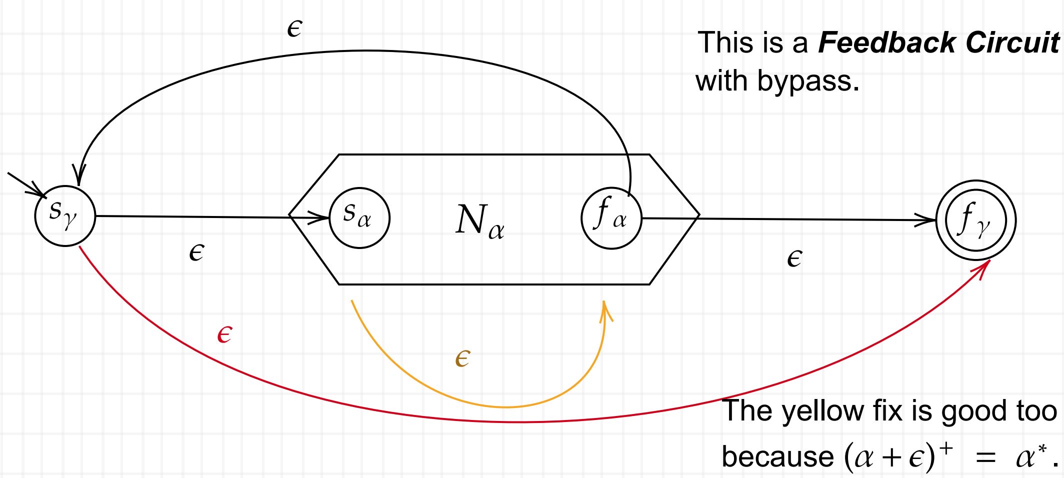


Is this good? We want to make $L(N_\gamma) = L(N_\alpha)^*$. Then the IH $L(N_\alpha) = L(\alpha)$ will give $L(N_\gamma) = L(\alpha)^* = L(\alpha^*) = L(\gamma)$ as needed---to finish the whole proof.

Whoops: The machine requires N_α to be entered at least once, so it really does $L(N_\alpha)^+$, not $L(N_\alpha)^*$. There was what we now consider a glitch in an old programming language's for-loop where it would execute at least once even if the range was null. To get * for "zero-or-more" rather than superscript $^+$ for "one-or-more" we can add an extra ϵ-arc:

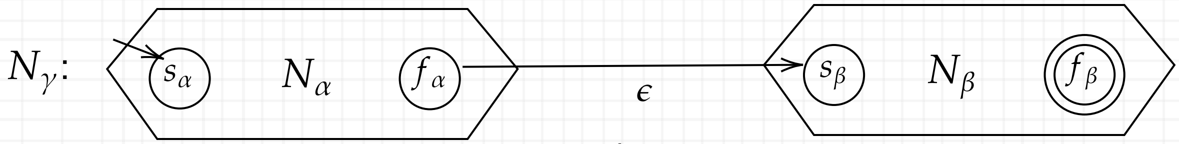
(I3) Given any regexp α, $\gamma = \alpha^*$ is a regexp; $L(\gamma) = L(\alpha)^*$; and we can build:

$N_\gamma =$

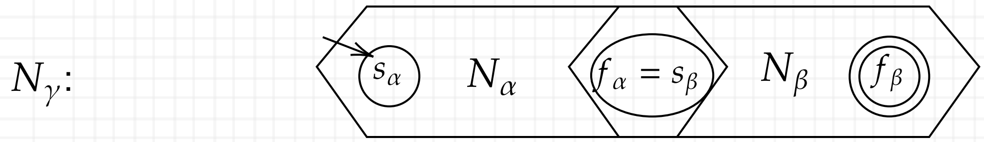


The proof yields an algorithm for converting any regular expression into an equivalent NFA. The algorithm works by recursion on operators in the regular expression. In practice, you don't have to follow it quite so literally, and you can often avoid most fo the ϵ-arcs that it introduces. The most common place to save is in the concatenation case.

(I2) $\gamma = \alpha \cdot \beta$ is a regexp; $L(\gamma) = A \cdot B = \{xy : x \in A \wedge y \in B\}$.

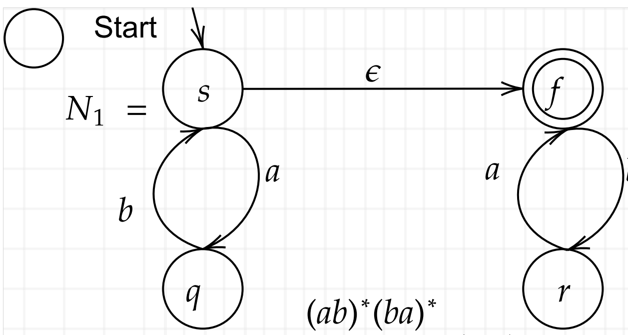


can become

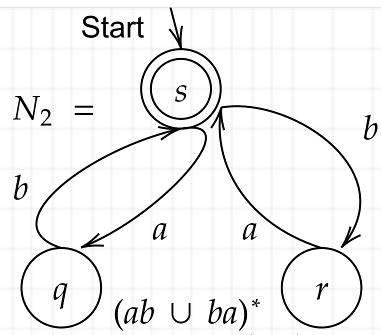


unless there is both an arc into s_β in N_β **and** an arc out of f_α in N_α.

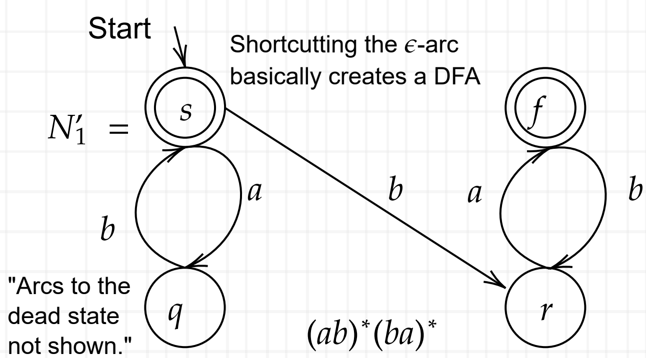
So in the example from the Thu. 2/11 lecture, we needed the ϵ-arc:

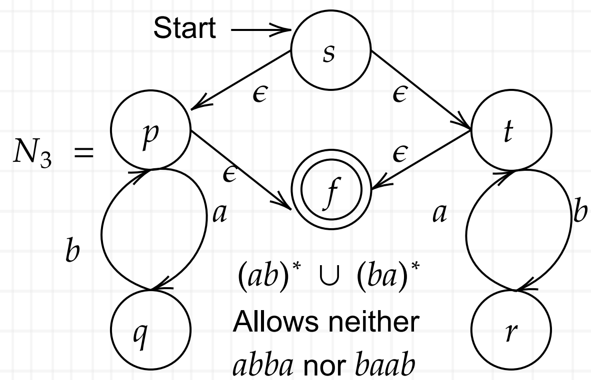


$(ab)^*(ba)^*$
Does not allow $baab$



$(ab \cup ba)^*$
Does allow $baab$

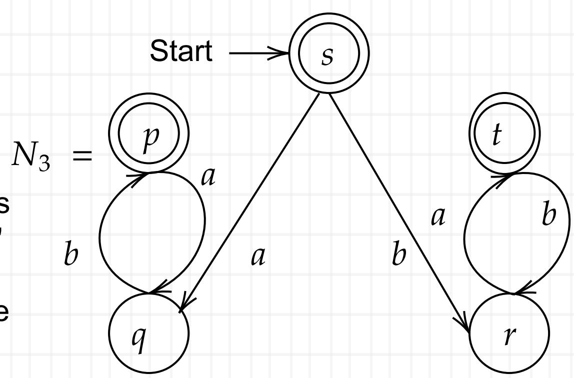




$(ab)^* \cup (ba)^*$
Allows neither
$abba$ nor $baab$

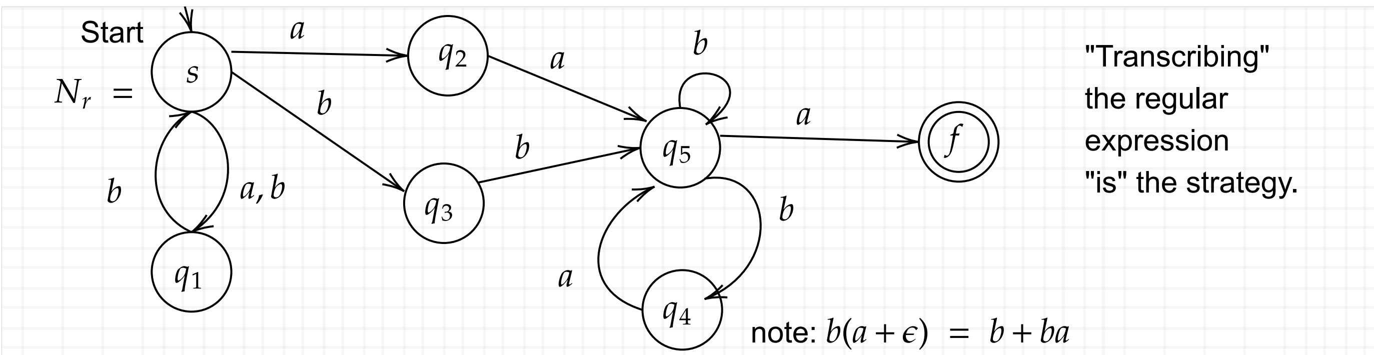
(In lecture I added the new state f with arcs from p and t, which originally were the accepting states.)

Instead, we can redesign the FA to not have any ϵ-arcs. Then it is a "DFA manqué" meaning it lacks only a dead state and arcs to it.



The example at bottom right could be "shortcutted" by making s an accepting state (which you can do anyway) and making its arcs go on a to state q and on b to state r instead. Some texts stop to prove the theorem that every NFA with ϵ-arcs can be (efficiently!) converted into an equivalent NFA without them, in order to do "NFA-to-DFA" without them. Our text by Sipser tries to have it both ways by doing the proof first without them and then with them, but (on Thursday) I will prefer to embrace the ϵ's. But for building NFAs, you can usually avoid the ϵ-arcs on the fly because many common examples involve languages where things naturally go forward.

Example: $r \;=\; (ab+bb)^*(aa+bb)(b(a+\epsilon))^*a\,.$



How can we track this machine on an input such as $x \;=\; bbaabbaa$? We can try individual computations by trial-and-error:

$(s, b, q_3, b, q_5, a, f, a$ ---? Crash!
$(s, b, q_1, b, s, a, q_2, a, q_5, b, q_4, b,$ --- Crash!
$(s, b, q_1, b, s, a, q_2, a, q_5, b, q_5, b, q_5, a, f, a$ --- Cannot process the final a, so Crash!
$(s, b, q_1, b, s, a, q_2, a, q_5, b, q_5, b, q_4, a, q_5, a, f)$: end of string, and state is f, so *accept*.

The idea of the DFA is to keep track of all the possibilities in-parallel:

$(\{s\}, b, \{q_1, q_3\}, b, \{s, q_5\}, a, \{f, q_2\}, a, \{q_5\}, b, \{q_4, q_5\}, b, \{q_4, q_5\}, a, \{f, q_5\}, a, \{f\})$.