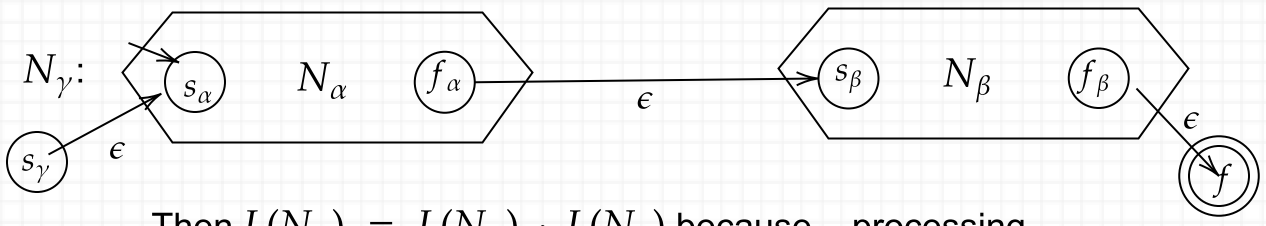
CSE396 Spring 2026 Week 3 Thu.: Regexps to NFAs to DFAs.

Continuing both the inductive definition of regular expressions and the proof of their having equivalent NFAs (with ϵ-transitions):

(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....

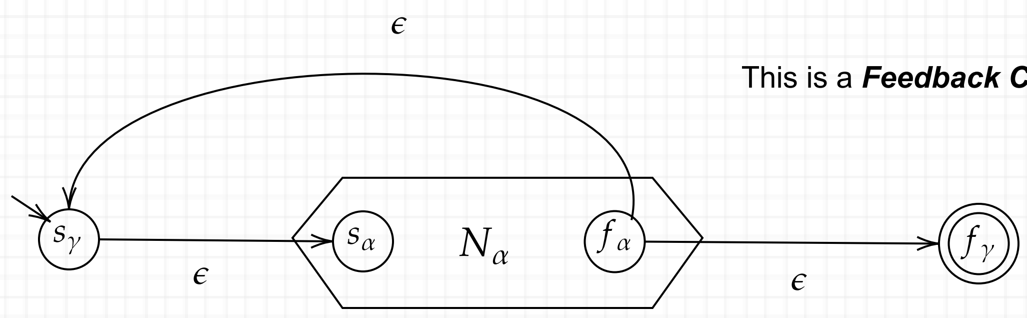
To write the reasoning out: N_γ can process a string z from its start state $s_\gamma = s_\alpha$ to its (unique) final state $f_\gamma = f_\beta$ if and only if z has a first part x that gets processed from s_α to f_α and a second part y that gets processed from s_β to f_β (with the ϵ from f_α to s_β 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.

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

$N_\gamma =$

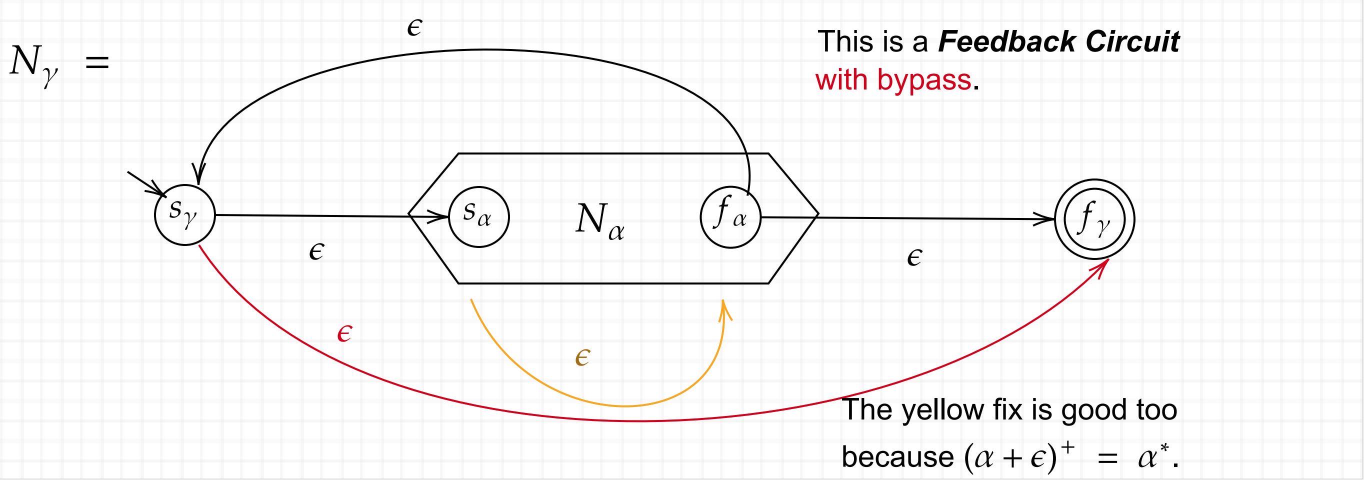
This is a ***Feedback Circuit***



Is this good? We want to make $L(N_\gamma) = L(N_\alpha)^*$. Then the IH $L(N_\alpha) = L(\alpha)$ will give $L(N_\gamma) = L(\alpha)^* = L\left(\alpha^*\right) = L(\gamma)$ 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:

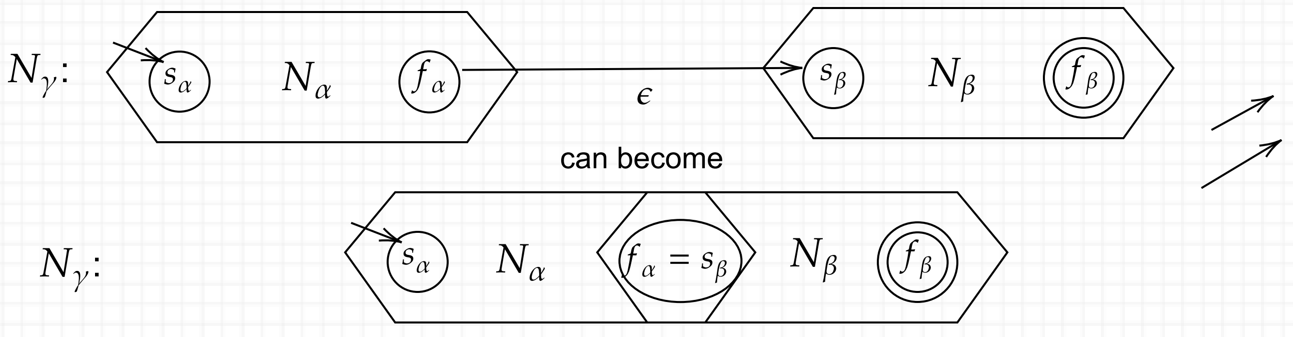


$N_\gamma =$
ϵ
This is a ***Feedback Circuit*** with bypass.
s_γ
s_α
N_α
f_α
f_γ
ϵ
ϵ
ϵ
ϵ
The yellow fix is good too because $(\alpha + \epsilon)^+ = \alpha^*$.

This completes both the formal inductiuve definition of regular expressions γ and also the inductive proof that there is always an NFA N_γ such that $L(N_\gamma) = L(\gamma)$. Moreover, the proof gives an algorithm for building an equivalent NFA. The algorithm works by recursion on operators in the regular expression.

In practice, you don't have to follow the above proof 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; $L(\gamma) = A \cdot B = \{xy : x \in A \wedge y \in B\}$.

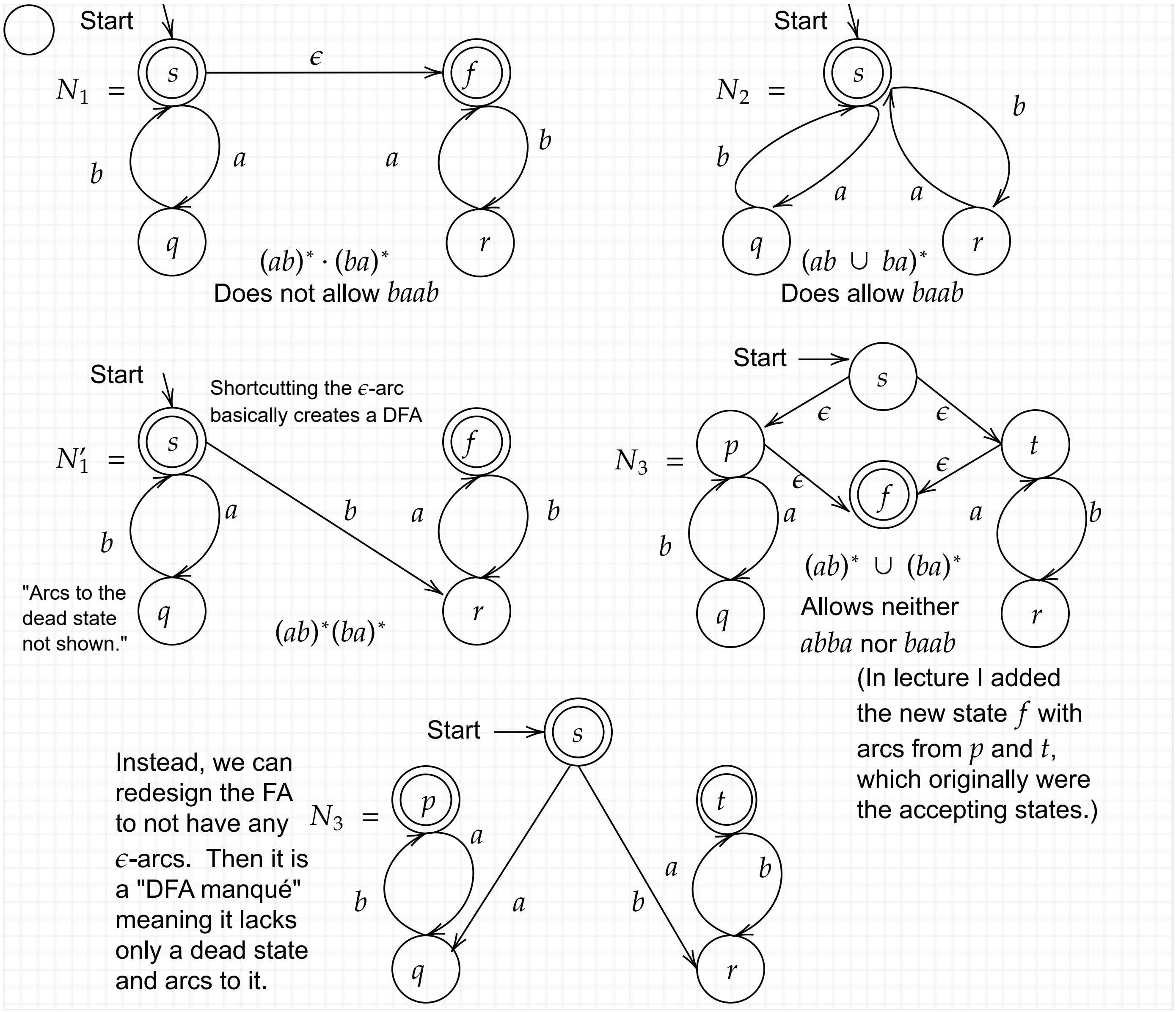


unless there is both an arc into s_β in N_β **and** an arc out of f_α in N_α.

What's "electric" about this? Think of the N_α and/or N_β as like resistors (or capacitors) in electrical circuit diagrams. The s and f points are like entry and terminal nodes; the extra step of having just one accepting state f_γ in the final machine is like the standard advice to ground the final circuit at a single terminus. Then say, what fundamental circuit construction primitives do the three induction cases embody? I will try to argue later this month that these language operations actually do reflect "wiring" in our brains.

Some Examples

In the earlier lecture example lecture of $(ab)^*(ba)^*$, we do need the ϵ-arc in the middle:



Shortcutting the ϵ-arc basically creates a DFA

"Arcs to the dead state not shown."

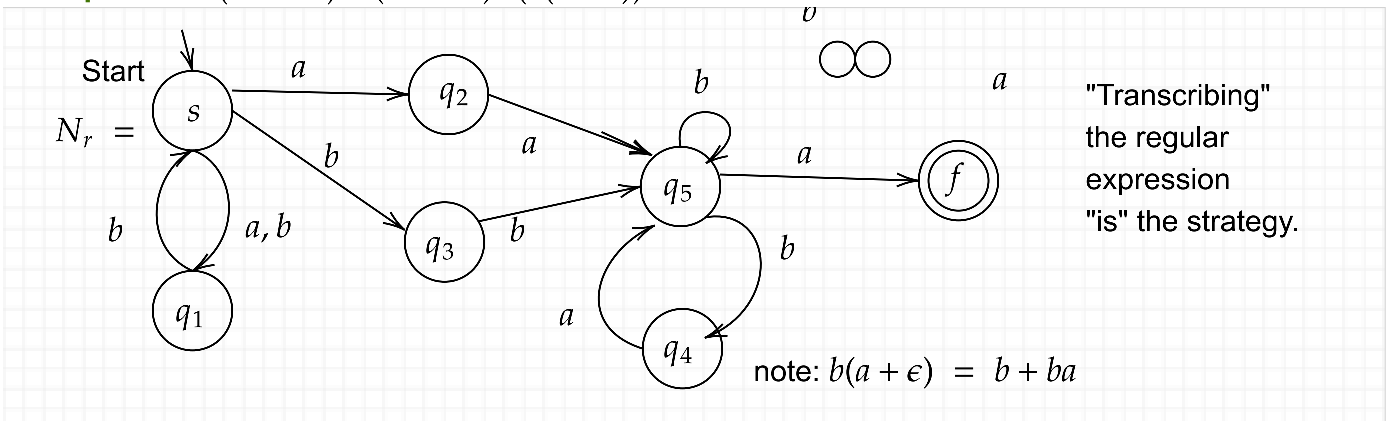
Allows neither $abba$ nor $baab$

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


Start
$N_r =$
s
q_1
q_2
q_3
q_4
q_5
f
a
b
a, b
"Transcribing" the regular expression "is" the strategy.
note: $b(a + \epsilon) = b + ba$

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 conversion in the next lecture is to keep track of all the possibilities in-parallel:

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

The Equivalence Theorem, Part II: NFA-to-DFA

Theorem: Given any NFA $N = (Q, \Sigma, \delta, s, F)$ we can build a DFA $M = (\mathcal{Q}, \Sigma, \Delta, S, \mathcal{F})$ such that $L(M) = L(N)$.

Notice that s got capitalized to S, which hints that S is a *set* rather than a single element. And δ got capitalized to Δ. Q and F were already sets, but they got...curlier. What does that mean? Well, that they are "of an even higher order"---sets of sets, for instance. An important set of sets is:

$\mathcal{P}(Q)$, also written 2^Q, called the *power set* of Q and defined as $\{R : R \subseteq Q\}$.

Unlike what textbooks tend to say, we will not necessarily make $\mathcal{Q}$ be all of $\mathcal{P}(Q)$, just those subsets R that are *reachable* from S. What this means is that the states of the DFA will be sets of states of the NFA---the states that are *possible* upon *processing* a given part of the input string x.

This suggests the question, which states (of N) are possible **before** we process any chars in x? Obviously the start state s of N is possible, but are there any others? Yes, if there are ϵ-transitions out of s. Define $E(s)$ to be the set of states of N that are reachable this way. If N has no ϵ-arcs (out of s or overall), then $E(s)$ is just $\{s\}$. Thus we begin building M by taking

$$S = E(s).$$

We could have said "S" in place of "$E(s)$" to begin with, but the E notation in the textbook is useful because we can use it to define the following for any set $R \subseteq Q$ of states of N:

$$E(R) = \{r : \textit{for some } q \in R,\ N \textit{ can process } \epsilon \textit{ from } q \textit{ to } r\}$$

This is called the *epsilon-closure* of R. If $E(R) = R$ then R is already *epsilon-closed*. It sounds "weeny" technical, but we will only need to use subsets that are ϵ-closed. The insights are

- $E(E(R)) = E(R)$: applying a closure operation once is always enough.
- *The states of the DFA need only be the* ***possible*** *subsets of states of the NFA*.
- *A subset R is good if it contains at least one accepting state, i..e, if $R \cap F \neq \emptyset$, because that will mean it is possible for the NFA to accept the string.*

We are thus ready to specify this much of the DFA:

- $\mathbb{Q} = \{\textit{possible } R \subseteq Q\}$;
- Σ is the same;
- $S = E(s)$;
- $\mathcal{F} = \{R \in \mathbb{Q} : R \cap F \neq \emptyset\}$.

The only component of M left to define is Δ. For any $P \in \mathbb{Q}$ (i.e., $P \subseteq Q$ and P is possible) and $c \in \Sigma$ define

$$\Delta(P, c) = \{r : \textit{for some } p \in P,\ N \textit{ can process } c \textit{ from } p \textit{ to } r\}.$$

This means that (p, c, r) can be a virtual instruction, but it might not be a literal instruction in δ because we might have to process ϵ's before and after the character c. We can save half the trouble by realizing that any "ϵ's before" are taken care of by the possible set-states P already being ϵ-closed. The text doesn't say this, but when solving these problems, it is IMHO a help to use the following definition first to build a table from the given NFA:

$$\underline{\delta}(p, c) = \{r : \text{you can get from } p \text{ to } r \text{ by first processing } c \text{ at } p\text{, then doing any } \epsilon\text{-arcs}\}.$$

More formally, $\underline{\delta}(p, c) = \{r : (\exists q)[(p, c, q) \in \delta \wedge r \in E(q)\}$. Then for **possible** $P \in \mathbb{Q}$ and $c \in \Sigma$, we get the equivalent definition

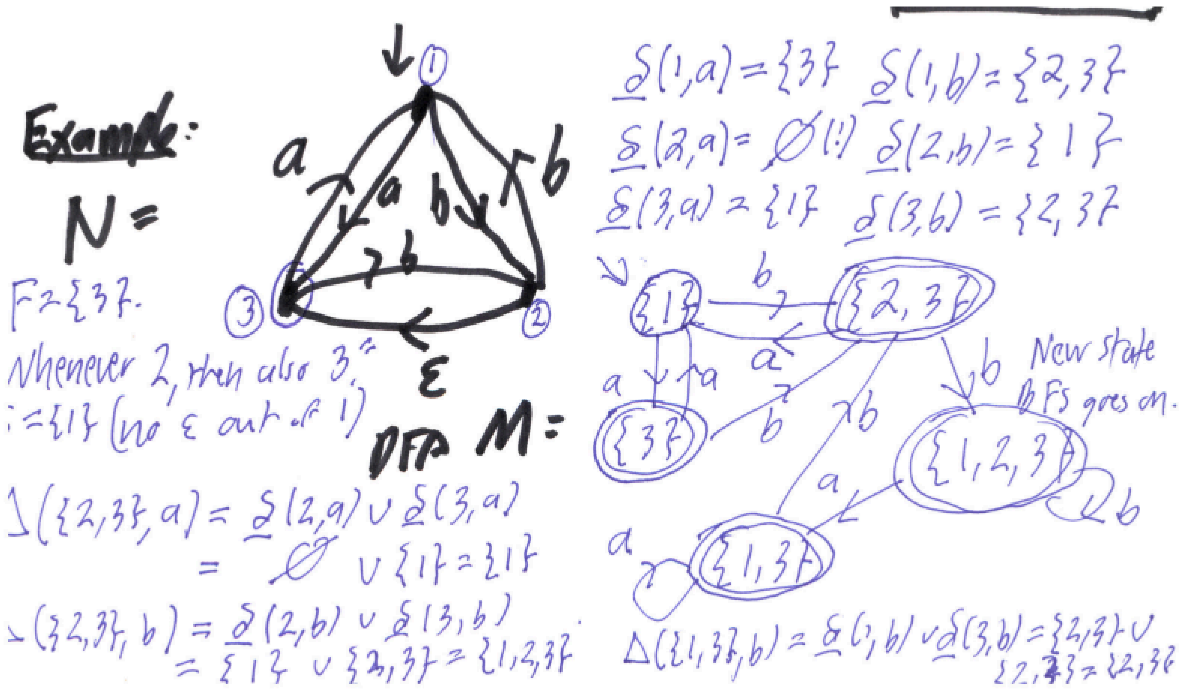
$$\Delta(P, c) = \bigcup_{p \in P} \underline{\delta}(p, c).$$

Even if there are no ϵ's, the idea of limiting to "possible" P often helps in a second way: we avoid having to define instructions for set-states that are never actually encountered. At the beginning, we *encounter* S. Then "expanding" S means computing $\Delta(S, c)$ for each char c. Thus, if $\Sigma = \{0, 1\}$ then the "first generation" are the states $P_0 = \Delta(S, 0)$ and $P_1 = \Delta(S, 1)$. One (or both) of these might equal S again, in which case we have nothing more to do with it. But whichever one(s) are new need to be expanded again to fill out the "second generation." We keep on expanding new set-states---"new" meaning we have not encountered that exact set before---until a generation turns up no new state. Then we say "the DFA has closed" and we're done.

[FYI: It is really a **breadth-first search** that has closed. If you've seen breadth-first search executed on graphs, this one is scaled up in a big way. It is not done on the graph G_N of N but rather on the potentially exponentially bigger graph $\mathcal{G}$ whose nodes are the sets of states. The graph $\mathcal{G}$ is given **implicitly** via the table $\underline{\delta}$ and the rule for Δ. Just don't think you necessarily have to write out all $2^4 = 16$ set-states when given a 4-state NFA like some sources show in diagrams.

If there are no ϵ's, then $\underline{\delta}$ is just the same as the text's set-valued δ function. But when there are ϵ's, writing out the $\underline{\delta}$ table (which cuts out the ϵ's) is a much better use of your time, IMPO, than just copying out the text's δ table with the ϵ column. Why just recopy information that is already explicitly present in the diagram? Whereas, IMHO, the step from N to $\underline{\delta}$ is not typo-prone when done on-the-fly and most usefully breaks your work in half.]

Example


Example:
N =
F = {3}.
Whenever 2, then also 3.
S = {1} (no ε out of 1)
DFA M:
$\underline{\delta}(1,a) = \{3\}$ $\underline{\delta}(1,b) = \{2,3\}$
$\underline{\delta}(2,a) = \emptyset$ (!) $\underline{\delta}(2,b) = \{1\}$
$\underline{\delta}(3,a) = \{1\}$ $\underline{\delta}(3,b) = \{2,3\}$
$\Delta(\{2,3\},a) = \underline{\delta}(2,a) \cup \underline{\delta}(3,a) = \emptyset \cup \{1\} = \{1\}$
$\Delta(\{2,3\},b) = \underline{\delta}(2,b) \cup \underline{\delta}(3,b) = \{1\} \cup \{2,3\} = \{1,2,3\}$
New state BFS goes on.
$\Delta(\{1,3\},b) = \underline{\delta}(1,b) \cup \underline{\delta}(3,b) = \{2,3\} \cup \{2,3\} = \{2,3\}$

[Lecture ended here. See next Tuesday's notes for the pickup.]