Lecture Overview

- **Deterministic Finite Automata (DFA)**
  - **accepting a string**
  - **defining a language**
- **Nondeterministic Finite Automata (NFA)**
  - **converting to DFA (subset construction)**
  - **constructed from a regular expression**
- **Regular expression for the language of a DFA**

Chapter 2 of text.

Deterministic Finite Automaton (DFA)

$(Q, \Sigma, \delta, q_0, F)$

Q: Finite set of states (text uses S)

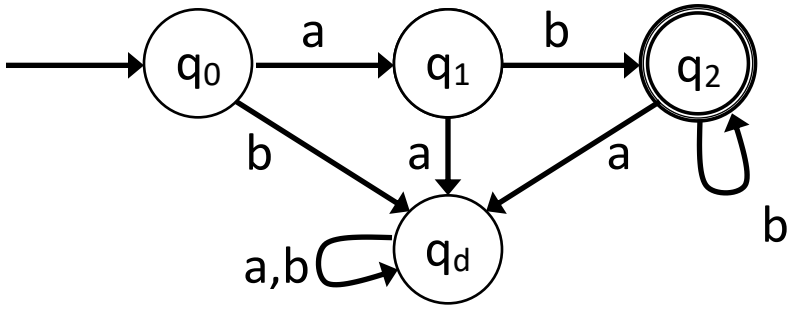
Σ: Finite alphabet

δ: Transition function $Q \times \Sigma \rightarrow Q$

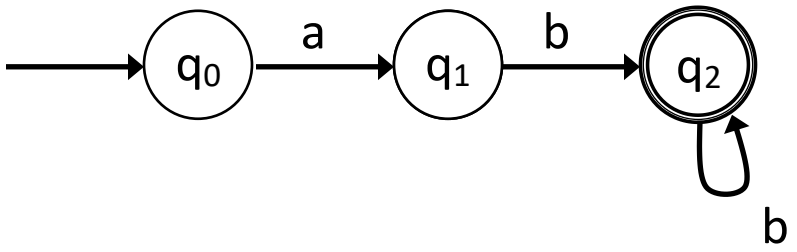
q_0: Start state $q_0 \in Q$

F: Final states $F \subseteq Q$

Typically, DFAs are specified by a **transition diagrams.** In a transition diagram, each state is drawn as a circle. Each transition $\delta(q, a) = q'$ is drawn as an arrow from the circle for q to the circle for q', labelled with a. The start state has an arrow "from nowhere" pointing to it. Final states are indicated by a double circle. The alphabet is all transition arrow labels.



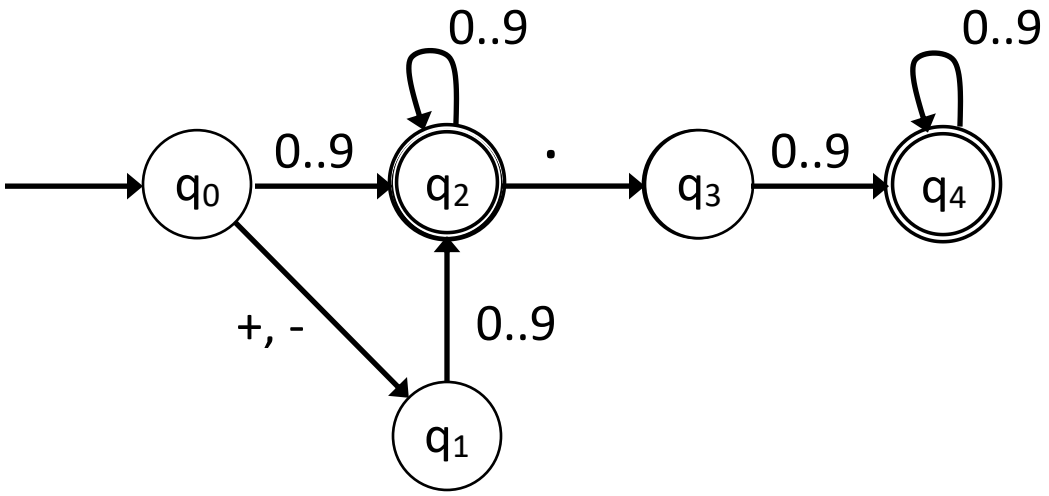
There is a convention that DFAs contain a "dead state" q_d that has $\delta(q_d, a) = q_d$ for all elements a of the alphabet. This state is not normally drawn on the transition diagram. Then, any state q that does not have a transition arrow on symbol $a \in \Sigma$ is assumed to have $\delta(q, a) = q_d$. Thus the above transition diagram is more usually and more simply drawn:



Accepting a String

Deterministic finite automata are given a string as input, and either **accept** or **reject** it.

- The machine starts in state q_0.
- In step k, the machine reads the k^{th} input character c and makes a transition from its current state q to the state $\delta(q, c)$. [On the diagram, it follows the arrow labelled with c.]
- It stops when there is no more input.
- If it stops in a final state, then it **accepts** the string, otherwise it **rejects** it.

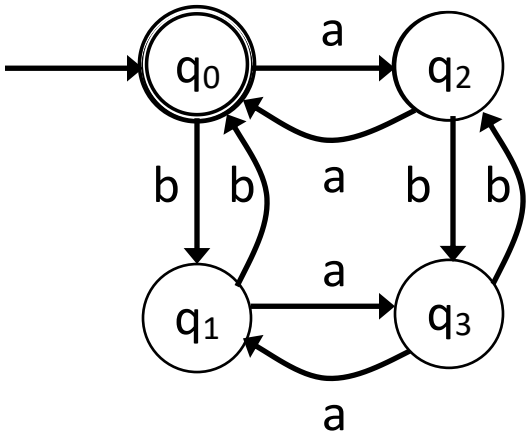


The **language accepted by a DFA** is simply the set of all strings accepted by the DFA.

The language accepted by our first DFA can be characterized by the regular expression ab^+. The language accepted by the second DFA is characterized by the regular expression $(+|-)^?[0..9]^+(.\ [0..9]^+)$. It **recognizes** signed integers and signed floats, where floats must have a digit before and after the decimal point.

Exercises:

- design a DFA to recognize the floats from the pika-1 specification.
- what does the following DFA recognize?



Nondeterministic Finite Automaton (NFA)

$(Q, \Sigma, \delta, q_0, F)$

Q: Finite set of states (text uses S)

Σ: Finite alphabet

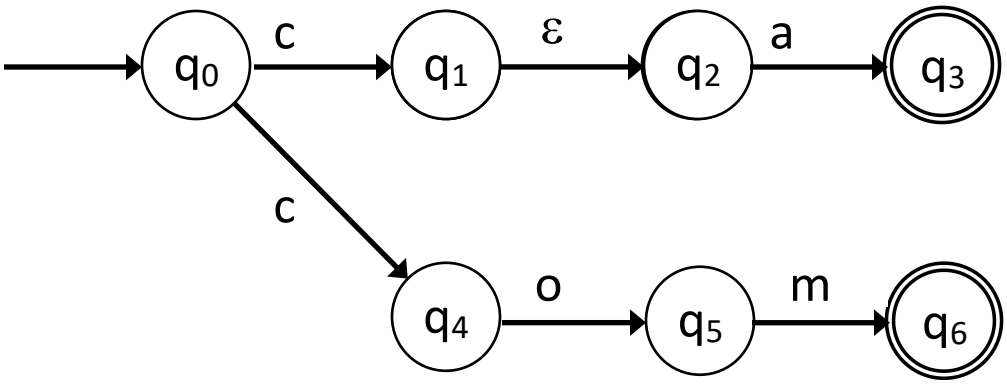
δ: Transition function $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$

q_0: Start state $q_0 \in Q$

F: Final states $F \subseteq Q$

NFAs are like DFAs, but…

- You can have more than one outgoing arrow from a state that is labelled with the same character.
- There are transitions on (arrows with label) ε, which can be taken at any time during the operation of the machine.
- The machine **accepts** if **there exists** a path labelled with the input (and epsilons) from the start state to a final state.



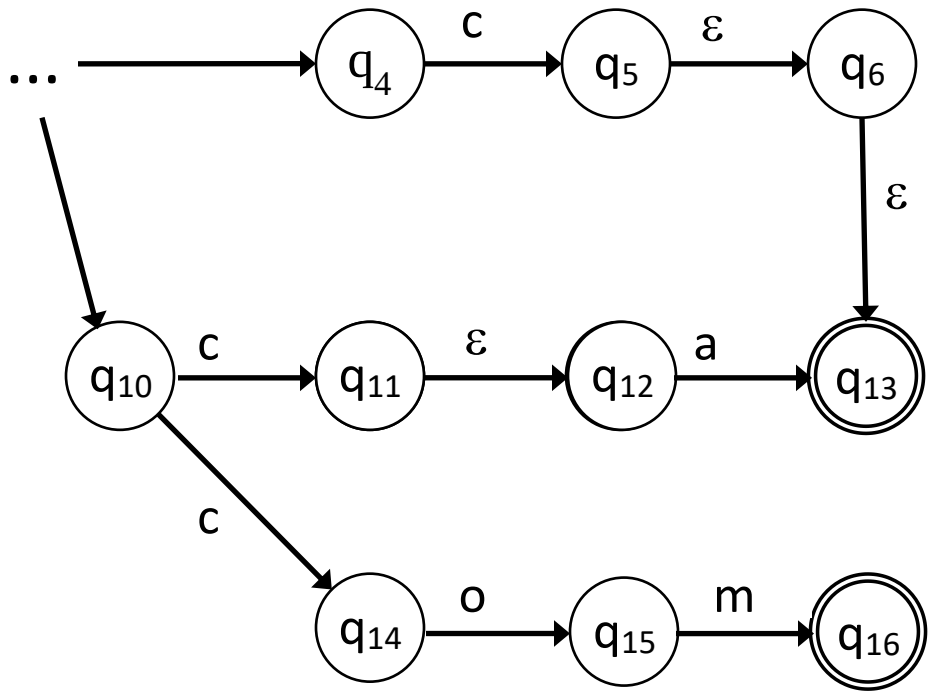
Converting an NFA to a DFA (Subset Construction)

Given NFA ($N, \Sigma, \delta_N, n_0, F_N$), we construct DFA ($D, \Sigma, \delta_D, d_0, F_D$).

- The alphabet for the machines is the same.
- The DFA states D correspond to **subsets** of the NFA states N.
- The DFA start state d_0 corresponds to ("is") the subset of N consisting of just n_0, and what you can reach from n_0 on ε-transitions.
- The final states F_D are those subsets of N that contain at least one final state of F_N.
- The transition function is the tricky part.

To find $\delta_D(d, a)$:

d corresponds to a subset N' of N. Construct $\delta_N(N', a)$ as the union, over all $n \in N'$, of $\delta_N(n, a)$. This gives a new subset N* of N. Take the **ε-closure** of N*, which is all states reachable from a state of N* by a series of 0 or more **ε-transitions**. $\delta_D(d, a)$ is this ε-closure.



In this example, if d = {q_4, q_{10}} and we have character **c**, then N* = $\delta_N(d, c)$ = {q_5, q_{11}, q_{14}}. ε-closure(N*) = { q_5, q_6, q_{13}, q_{11}, q_{12}, q_{14}}.

We could construct a state for every subset of N, and compute the transitions for every character from every state. But this is $O(|N| \, 2^{|N|} \, |\Sigma|)$ information, which is huge.

In practice, relatively few of the subsets correspond to states that the DFA can reach from the start state, so we take the approach of starting with the DFA start state and only **expanding** states corresponding to subsets that we reach. Expanding is computing the transition function for that state on each input symbol.

Here's the subset construction, from the text, but I've changed some variable names to protect the innocent:

d_0 = ε-closure(n_0)

D = {d_0} // set of all states of DFA

worklist = {d_0} // states discovered but not
// expanded

while (!worklist.isEmpty()) {

remove a state d from the worklist

// expand state d:

for each character c $\in \Sigma$ do

d' = ε-closure(δ_N(d,c))

δ_D[d, c] = d'

if d' $\notin$ D then

D = D $\cup$ {d'}

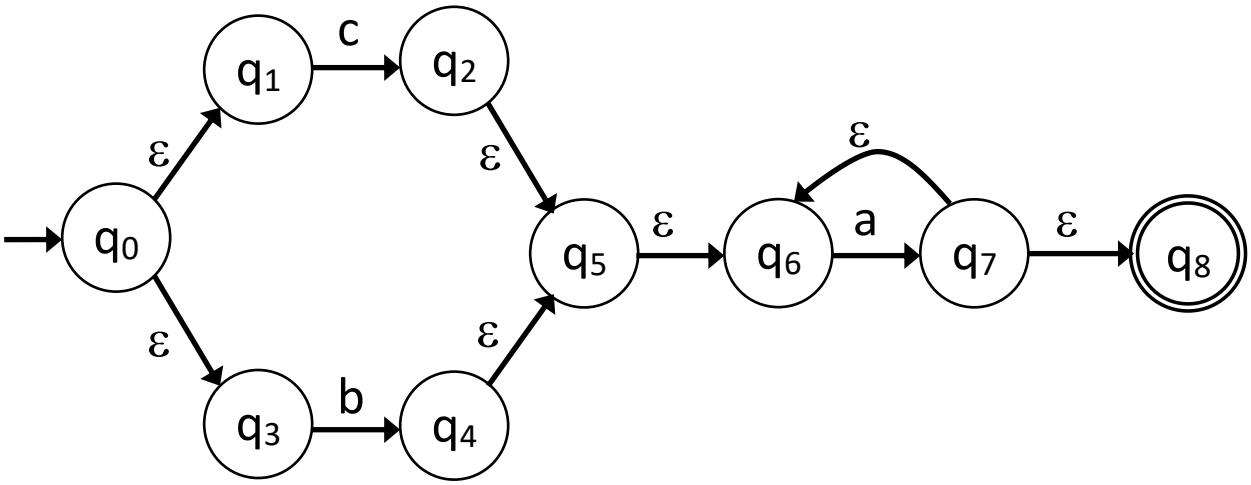
add d' to worklist

end if

end for

end while

Example conversion of NFA to DFA



First, d_0 = ε-closure(q_0) = {q_0, q_1, q_3}. The worklist starts with d_0 on it, and it is immediately removed to be expanded.

Note: we ignore the dead state of the NFA. This halves the labour and does not affect the result.

- $\delta_N(d_0, a)$ is empty so ε-closure($\delta_N(d_0, a)$) is also empty. An empty transition is interpreted as a transition to the DFA's dead state.
- $\delta_N(d_0, b)$ = {q_4}. Its ε-closure is {q_4, q_5, q_6}. This subset is not in the set D, so we name it d_1, put it in D, and add it to the worklist.

- $\delta_N(d_0, c) = \{q_2\}$. Its ε-closure is $\{q_2, q_5, q_6\}$. This subset is not in the set D, so we name it d_2, put it in D, and add it to the worklist.

We're done expanding d_0, and we now have d_1 and d_2 on the worklist. We choose d_1 to expand next.

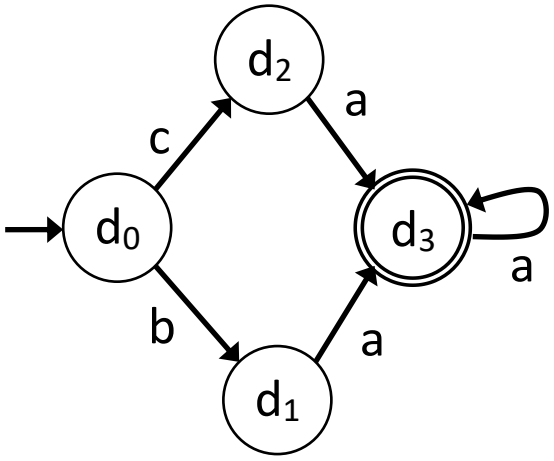
- $\delta_N(d_1, a) = \{q_7\}$. Its ε-closure is $\{q_6, q_7, q_8\}$. This subset is not in the set D, so we name it d_3, put it in D, and add it to the worklist.
- $\delta_N(d_1, b)$ and $\delta_N(d_1, c)$ are empty and give us transitions from d_1 to the DFA's dead state.

We're done expanding d_1, and we now have d_2 and d_3 on the worklist. We choose d_2 to expand next.

- $\delta_N(d_2, a) = \{q_7\}$. Its ε-closure is $\{q_6, q_7, q_8\}$. This subset is already in D (and called d_3), so we do nothing further with it.
- $\delta_N(d_2, b)$ and $\delta_N(d_2, c)$ are empty and give us transitions from d_2 to the DFA's dead state.

We're done expanding d_2, and we now have only d_3 on the worklist. We expand d_3 next.

- $\delta_N(d_3, a) = \{q_7\}$. Its ε-closure is $\{q_6, q_7, q_8\}$. This subset is already in D (and called d_3), so we do nothing further with it.
- $\delta_N(d_3, b)$ and $\delta_N(d_3, c)$ are empty and give us transitions from d_3 to the DFA's dead state.
- We're done expanding d_3, and we now have an empty worklist. We are done. Here is the DFA that we constructed:



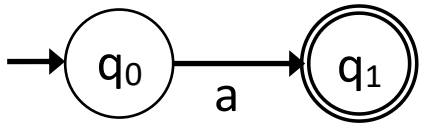
This machine is not the simplest (the simplest has 3 nondead states) but it's not 2^8 states, which is what we would get if we used every subset of the NFA's states.

There is an algorithm that takes a DFA and constructs another DFA for the same language, but with a minimum number of states. You can find this algorithm in your text.

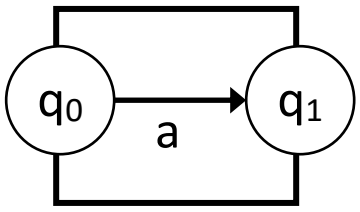
Constructing an NFA from a regular expression

Also known as **Thompson's Construction**.

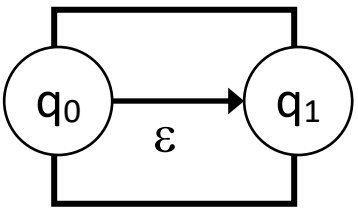
We construct an NFA recursively as we build the regular expression recursively. For any symbol $a \in \Sigma$, the NFA for a is:



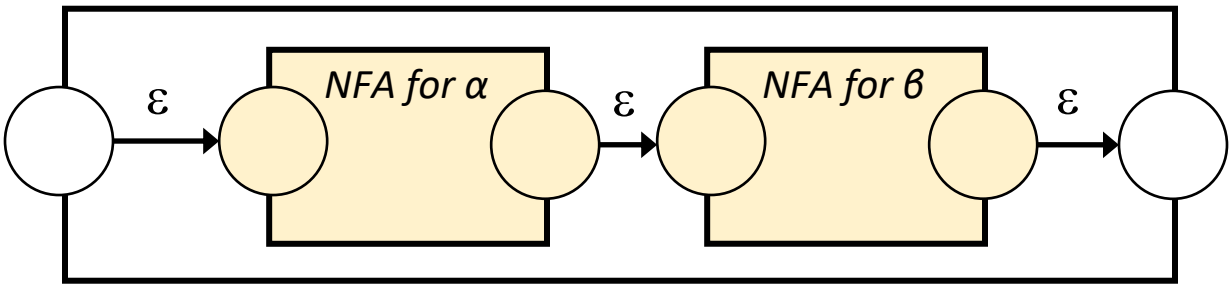
But we will be drawing them a bit differently. The start state of an NFA will be on the left of a box, and the final state (there will only be one) will be on the right of the same box. So our NFA above becomes:



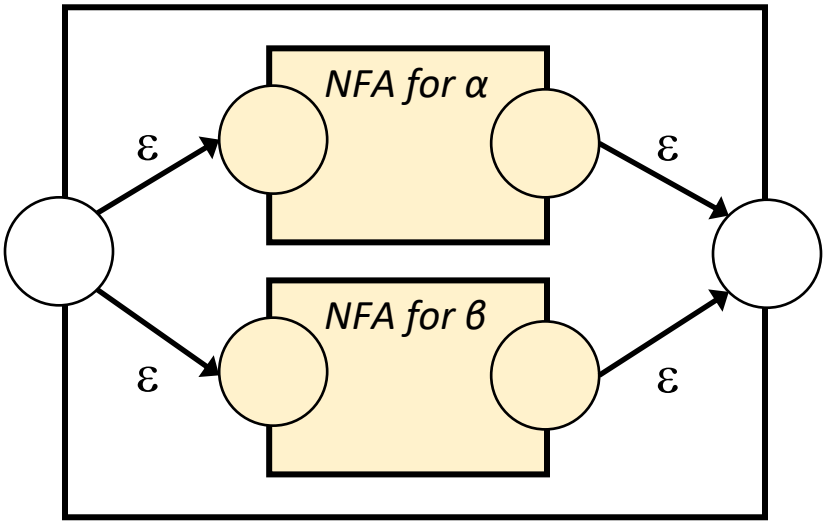
We have a similar NFA for ε:



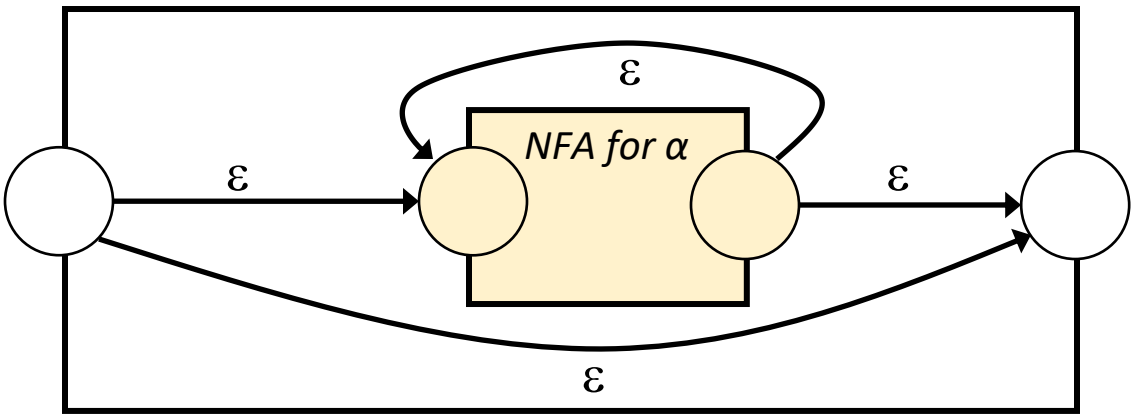
We must construct NFAs for three operations: concatenation, alternation, and Kleene closure. Here is the machine for the concatenation αβ of regular expressions for α and β:



And here is the machine for the alternation α|β:

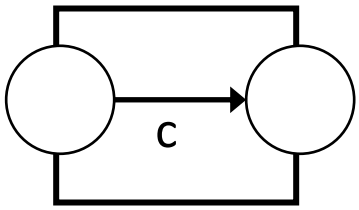


Finally, here is the machine for α*:

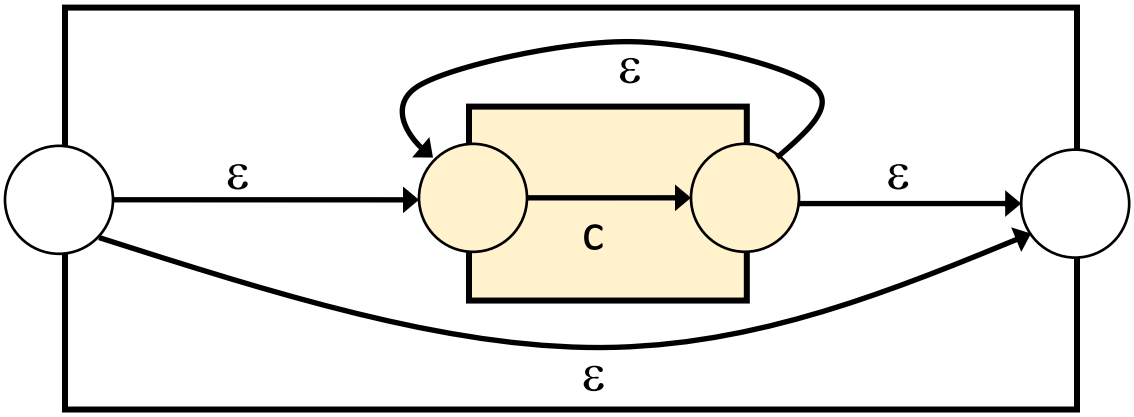


Now, we'll do a simple example of Thompson's Construction: c*(a|b)

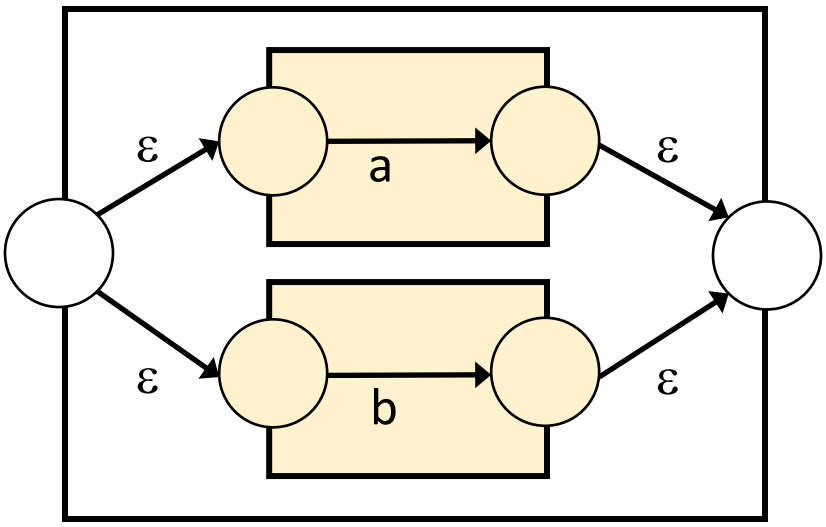
First we take the machine for c:



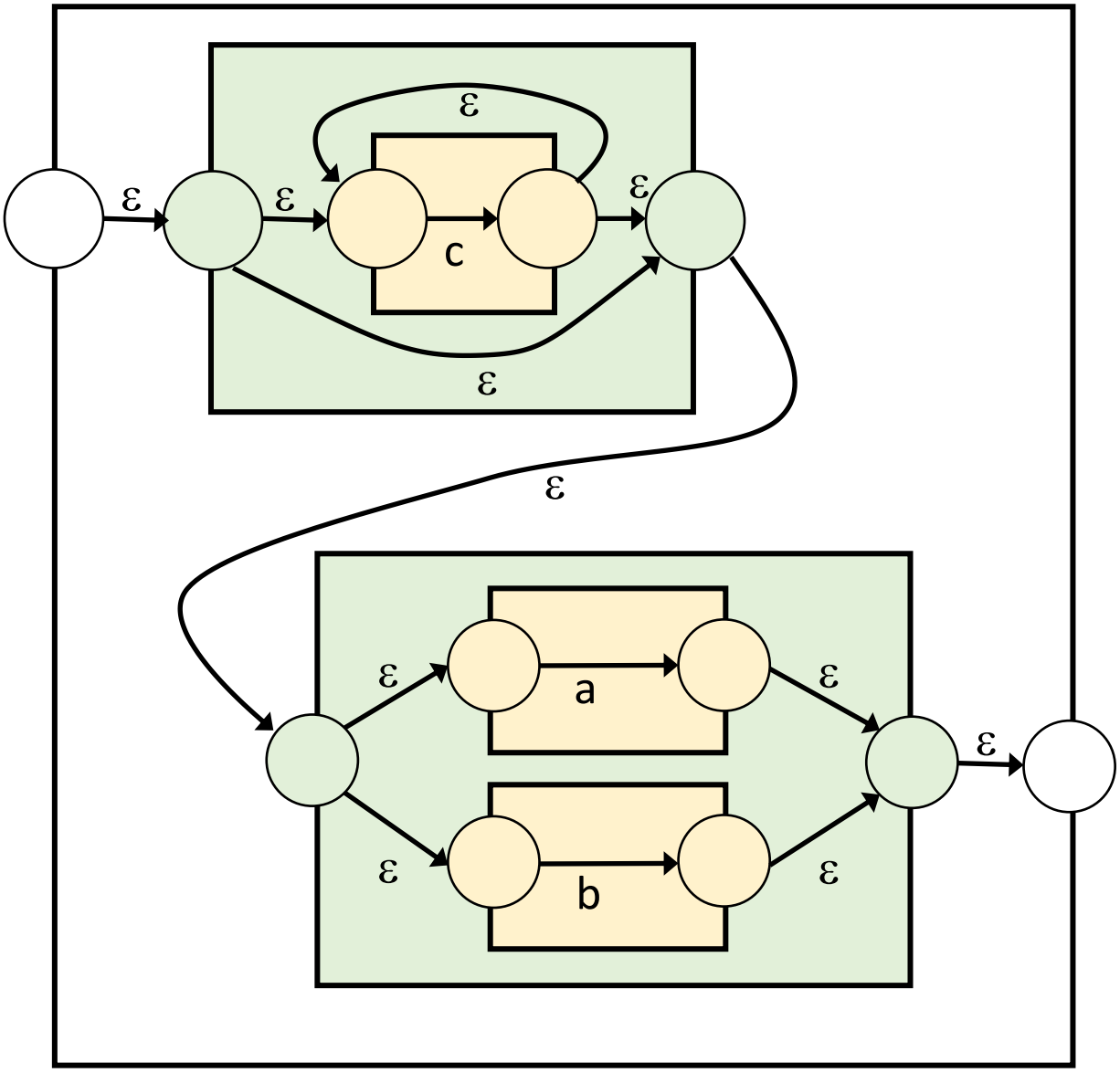
and apply the construction for α* to it:



So that's the machine for c*. Now we construct the machine for (a|b) by the construction for (α | β) applied to the machines for a and b:

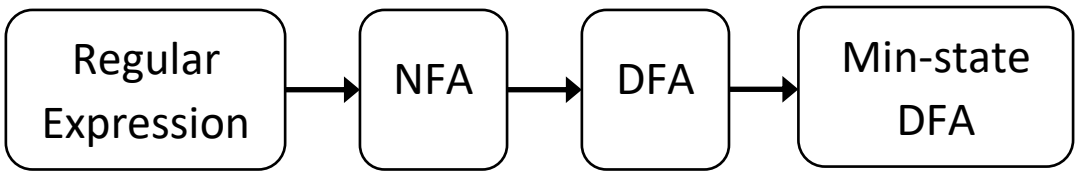


Now we apply the concatenation construction to the machines for a* and (b|c):



And that completes the construction of the NFA for c*(a|b).

So far, we've seen that given a regular expression, you can construct an equivalent NFA for it. By the subset construction, you can then get an equivalent DFA, and then an equivalent minimum-state DFA.



DFAs are easy to implement: the transition function is just a table (2D array) listing a next state for a given state and input symbol. Given a regular expression, one can construct this transition table for an equivalent minimum-state DFA; this is sometimes called **compiling** the regular expression. This operation is used in most libraries and applications that deal with regular expressions. It is **the** way to speedily handle searching for instances of the regular expression.

It is also the basis for automatic lexical analyzer generators; the programmer supplies the regular expressions for different tokens, and the generator

combines them into a single big NFA and converts that to a minimum-state DFA. Then scanning can be done with one table-lookup per input character.

DFAs can recognize any language that NFAs can. The converse is also true: DFAs are simply NFAs that don't take advantage of the extra power NFAs have.

NFAs can recognize any language a regular expression can. Is the converse true?

The answer is **yes**. NFAs, DFAs, and regular expressions are equivalent in expressive power.

We will prove that there is a regular expression equivalent to an arbitrary DFA.

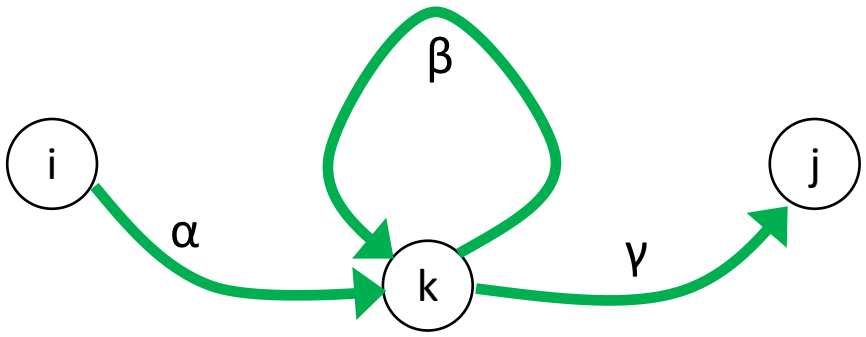
Kleene's Construction

Given the DFA, we construct a three-dimensional table R_{ij}^{k}. The entry in R_{ij}^{k} will be a regular expression for all of the paths from q_i to q_j in the DFA, using only intermediate states numbered k or less…i.e. using only intermediate nodes q_0, q_1,… q_k.

We construct starting with k=-1 (no intermediate nodes), then continuing with k=0, k=1, etc. up to k=n-1. For k=-1,

R_{ij}^{-1} = the "or" of all symbols on edges from q_i to q_j, with ε also "or"ed in if i=j.

To compute R_{ij}^{k} for other k, consider a path from q_i to q_j that uses intermediate states numbered k or less. If the path does not use state k, then it is accounted for in R_{ij}^{k-1}. If it does, then it must go on a path from q_i to the first occurance of q_k, followed by several cycles going from q_k back to q_k, followed by a last path section from q_k to q_i.



That is, all paths from i to j are described by αβ*γ.

The paths in α are described by the regular expression R_{ik}^{k-1}. The cycles of β are described by the regular expression R_{kk}^{k-1}. And the paths in γ are described by the regular expression R_{kj}^{k-1}.

So $R_{ij}^{k} = R_{ij}^{k-1} \mid \alpha\beta^*\gamma = R_{ij}^{k-1} \mid R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1}$

Assuming a single final state q_{n-1}, the regular expression equivalent to the DFA will be $R_{0,n-1}^{n-1}$.

Multiple final states are handled by “or”ing together the regular expressions for the individual final states.