We make a slight change to the heart of the proof where we left off. The change saves some time in executing the NFA-to-DFA construction when $\varepsilon$-arcs are present and reduces errors. First define

$$\delta(p, c) = E(\{(q, (p, c, q) \in \delta)\})$$

for any state $p \in Q$ and char $c$. Recall $E(\cdot)$ is $\varepsilon$-closure. So what this means in simple terms is:

1. **First** take arc(s) on $c$ out of the state $p$.
   - If there are none, **stop** and put $\delta(p, c) = \emptyset$.
   - Else collect all states $q$ reached on those arc(s).

2. **Then**, for each state $q$ reached by processing $c$, add states reached on any series of $\varepsilon$-arcs out of $q$, if there are any.

Now we can give a new definition of the DFA's transition function $\Delta$: for any $P \subseteq Q$ and $c \in \Sigma$,

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

The difference is that we avoid worrying about initial $\varepsilon$-arcs that could come before processing $c$. We only have to track trailing ones in a machine diagram. The reason is that the trailing arcs at the previous step already took care of any initial ones now. Initializing the start state $S$ of the DFA $M$ to have all states reached by $\varepsilon$-arcs out of $s$ in $N$ sets this in motion. We need to prove for all $i$:

$$G(i): \Delta^*(S, x_1 \cdots x_i) = \{r: N \text{ can process } x_1 \cdots x_i \text{ from } s \text{ to } r\}.$$

Here we have extended $\Delta$, a function of a state and a char, to $\Delta^*$ which is a function of a state and a string, by the basis $\Delta^*(R, \varepsilon) = R$ for all $R \subseteq Q$ and for $i \geq 1$,

$$\Delta^*(R, x_1 \cdots x_{i-1}x_i) = \Delta(\Delta^*(R, x_1 \cdots x_{i-1}), x_i).$$

So let $R_{i-1}$ stand for $\Delta^*(S, x_1 \cdots x_{i-1})$. Then by the inductive hypothesis $G(i - 1)$, $R_{i-1}$ equals the set of states $q$ such that $N$ can process $x_1 \cdots x_{i-1}$ from $s$ to $q$. Now put $R_i = \Delta(R_{i-1}, x_i)$.

- Let $r \in R_i$. Then $r \in \delta(q, x_i)$ for some $q \in R_{i-1}$. By IH $G(i - 1)$, $N$ can process $x_1 \cdots x_{i-1}$ from $s$ to $q$. And $N$ can process $x_i$ from $q$ to $r$ by definition of $r \in \delta(q, x_i)$. So $N$ can process $x_1 \cdots x_i$ from $s$ to $r$.
- Suppose $N$ can process $x_1 \cdots x_i$ from $s$ to $r$. Then—-and this is the key point---the processing goes to some state $q$ just before the char $x_i$ is processed. By IH $G(i - 1)$, $q$ belongs to $R_{i-1}$. Moreover, $r \in \delta(q, x_i)$ because we first do the step that processed the char $x_i$ at $q$, then any trailing $\varepsilon$-arcs. Thus $r \in \Delta(R_{i-1}, x_i)$, which means $r \in R_i$. 

Thus we have established that $R_i$ equals the set of states $r$ such that $N$ can process $x_1 \cdots x_i$ from $s$ to $r$. This is the statement $G(i)$, which is what we had to prove to make the induction go through. This finally proves the NFA-to-DFA part of Kleene's Theorem. ☒

The extra things pointed out have to do with how the states of the DFA tell what the NFA can and cannot process:

- The NFA cannot process the string $bbb$ from its start state at all. However you try, you come
Now here is a simple algorithm for telling whether a given string \( x \) matches a given regexp \( \alpha \):

1. Convert \( \alpha \) into an equivalent NFA \( N_\alpha \).
2. Convert \( N_\alpha \) into an equivalent DFA \( M_\alpha \).
3. Run \( M_\alpha \) on \( x \). If it accepts, say "yes, it matches", else say "no match".

This algorithm is **correct**, but it is **not efficient**. The reason is that step 2 can blow up. An intuitive
reason for the gross inefficiency is that step 2 makes you create in advance all the "set states" that would ever be used on all possible strings \( x \), but most of them are unnecessary for the particular \( x \) that was given.

There is, however, a better way that builds just the set-states \( R_1, \ldots, R_i, \ldots, R_n \) that are actually encountered in the particular computation on the particular \( x \). We have \( R_0 = S = E(s) \) to begin with. To build each \( R_i \) from the previous \( R_{i-1} \), iterate through every \( q \in R_{i-1} \) and union together all the sets \( \delta(q, x_i) \). If \( N_a \) has \( k \) states---which roughly equals the number of operations in \( \alpha \)---then that takes order \( n \cdot k \cdot k \) steps. This is at worst cubic in the length \( \tilde{O}(n + k) \) of \( x \) and \( \alpha \) together, so this counts as a polynomial-time algorithm. It is in fact the algorithm actually used by the grep command in Linux/UNIX.

**Generalized NFAs (GNFAs) --- having only 2 states.**

A generalized NFA \( G \) can have any regular expression on any arc. A string \( x \) is "accepted" by \( G \) if it can be broken into \( m \) 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 \( f \). A regular NFA in in fact a GNFA in which every arc has a "basic" regular expression---that is, just a char \( c \) in \( \Sigma \), or \( \epsilon \).

I do not regard GNFAs as "machines" that can be "executed"---even in the sense where we could say that the grep algorithm executed the NFA \( N_a \) on \( x \). I regard them as helpful shorthand for diagramming languages. The most illuminating case IMHO of this is for two-state GNFAs:

**Diagram:**

\[
\begin{align*}
L(G) &= L_{sf} = (\alpha + \beta\gamma^*\eta)^*\beta\gamma^* \\
&= \alpha^*\beta(\gamma + \eta\alpha^*\beta)^*
\end{align*}
\]

\[
\begin{align*}
L(G) &= L_{ss} = (\alpha + \beta\gamma^*\eta)^*.
\end{align*}
\]

\( \eta \) is pronounced "ate-a" in the US, "eat-a" in the UK.