14.8 Constructing $\epsilon$-NFA’s From Regular Expressions

14.8.1 A Recursive Construction

Now we come to our reason for introducing $\epsilon$-NFA’s in the first place: Given any regular expression, we can easily produce a $\epsilon$-NFA whose language is denoted by that expression\(^1\).

Obviously the idea, as in all the earlier examples where we proved something about “all regular expressions”, will be an inductive proof. If we know that the base expressions $\emptyset$ and $a$ can be simulated by $\epsilon$-NFA’s, and can take $\epsilon$-NFA’s for two expressions and form $\epsilon$-NFA’s for their union, concatenation product, and Kleene star, we are done. Each of these steps is pretty easy, so one can pretty easily get a proof that the conversion is possible. If you actually want to do the conversion, of course, and then convert the resulting $\epsilon$-NFA into an NFA and a DFA, you want to be a little more careful.

Our particular version of the construction will try to keep the $\epsilon$-moves to a minimum. The way we’ll do this is by adding on to our inductive hypothesis. Not only will we construct an $\epsilon$-NFA for each regular expression, but all our $\epsilon$-NFA’s will have:

- Exactly one final state,
- No transitions into the start state, and
- No transitions out of the final state.

We now proceed with the construction:

- For $\emptyset$, we’ll build an $\epsilon$-NFA with two states $\{\iota, f\}$, start state $\iota$, only final state $f$, and no transitions (We never said there had to be any.) This has no non-trivial paths, and the start state isn’t final, so it can’t accept anything (even $\epsilon$) (Figure 14-30).

- For the language $\{a\}$ of the base expression $a$, we make a two-state $\epsilon$-NFA with states $\{\iota, f\}$, start state $\iota$, only final state $f$, and one transition $(\iota, a, f)$. This accepts the string $a$ and nothing else (Figure 14-31).

- Given regular expressions $R_1$ and $R_2$, by the inductive hypothesis we can assume that we have $\epsilon$-NFA’s $M_1$ and $M_2$, with start states $\iota_1$ and $\iota_2$, single final states $f_1$ and $f_2$, and languages $L(M_1) = L(R_1)$ and $L(M_2) = L(R_2)$ (Figure 14-32).

\(^1\)Again, there are a variety of ways to define this conversion, some of which don’t work — we’ll consider some variations in the Exercises and Problems.
We want a single $\epsilon$-NFA $N$ with $L(N) = L(M_1) \cup L(M_2)$. One way to do this would be to put the states of $M_1$ and $M_2$ together, adding a new start state $\imath$ and a new final state $f$ (so that $\imath_1$ and $\imath_2$ are no longer start states and $f_1$ and $f_2$ are no longer final), and adding $\epsilon$-moves $(\imath, \epsilon, \imath_1)$, $(\imath, \epsilon, \imath_2)$, $(f_1, \epsilon, f)$, and $(f_2, \epsilon, f)$. This works because any path from $\imath$ to $f$ in $N$ must be a copy of a valid path in either $M_1$ or $M_2$, so that $w \in L(N)$ if and only if $w \in L(M_1)$ or $w \in L(M_2)$.

Because of our assumptions about $M_1$ and $M_2$, though, we can make a simpler $\epsilon$-NFA that still has the same paths. Instead of adding two states, we’ll subtract two by merging $\imath_1$ with $\imath_2$ and $f_1$ with $f_2$. The merged state $\imath$ is the start state of $N$, and the merged state $f$ is the only final state (Figure 14-33). Clearly any path in $M_1$ or $M_2$ will correspond to a path in $N$. But showing the converse requires our assumption, because without it a path could start into $M_1$, return to $\imath$, go into $M_2$, and eventually wind up at $f$ without being either a path in $M_1$ or a path in $M_2$. With the assumption, though, once you leave $\imath$ into $M_1$ or $M_2$ there is no way out of the machine you’re in until you reach $f$ (and you can’t leave $f$ once you’re there).

We still have to check that $N$ satisfies our assumptions, which it does because we have introduced no transition into $\imath$ or out of $f$.

- To get $L(N) = L(M_1)L(M_2)$, we have to create $N$ so that a path through $N$ represents a path through $M_1$ followed by a path through $M_2$ (Figure 14-34). Without the assumption, we could do this by making $\imath_1$ the start state, $f_2$ the final state, and adding one new transition
(f₁, ε, ν₂). With our additional assumption, we can merge f₁ and ν₂ into a single state instead, because any path from ν₁ to f₁ can only continue into M₂. So \( w ∈ L(N) \) if and only if \( ∃u : ∃v : (uv = w) ∧ (u ∈ L(M₁) ∧ v ∈ L(M₂)) \), which is the definition of \( w ∈ L(M₁)L(M₂) \). Again we must check that we’ve preserved the assumption, in that our new start state has no transition into it and our new final state has no transition out of it.

- Now we have just \( M₁ \) and have to construct \( N \) so that \( L(N) = L(M₁)^* \). Here about the only way to proceed is to add two new states \( τ \) and \( f \), make \( τ \) the new start state and \( f \) the only final state, and add in transitions \( ⟨τ, ε, ν₁⟩, ⟨ν₁, ε, f⟩, ⟨f, ε, f⟩ \) (Figure 14-35). Any path through \( N \) must start at \( τ \), jump to \( ν₁ \), run between \( ν₁ \) and \( f₁ \) either through \( M₁ \) or on the new ε-moves, and finally jump from \( f₁ \) to \( f \). Again, our extra assumption is necessary in that we know that once the path goes into \( M₁ \), it can only come out at \( f₁ \) and must then make an ε-move to either \( ν₁ \) or \( f \). We know that each passage from \( ν₁ \) to \( f₁ \) reads either a string in \( L(M₁) \) or ε, so the whole path reads a concatenation of such pieces, i.e., a string in \( (L(M₁) + ε)^* \). But \( (L(M₁) + ε)^* = L(M₁)^* \) (Why?).

In Problem 14.8.3 you’ll consider an alternate construction that just adds the ε-moves from \( ν₁ \) to \( f₁ \) and vice versa, without adding the two new states. Note that this construction would not do for our purposes, because it leaves an ε-NFA with transitions both into the start state and out of the final state.

Note also that only the last of these five steps introduces any ε-moves, telling us that for any regular expression without the star operator, we can construct an ordinary NFA. If we could extend this property to the star step of the construction, we could get rid of ε-NFA’s entirely, but this doesn’t appear to be possible.
14.8.2 An Example

Let’s do one more example before leaving the topic, by making an \( \epsilon \)-NFA from the regular expression \((ab + ba)^* + bb\) (Figure 14-36). To begin with, we make \( \epsilon \)-NFA’s for the two-letter subexpressions. The one for \( ab \) has three states \( p, q, r \), start state \( p \), final state \( r \), and two transitions, \( \langle p, a, q \rangle \) and \( \langle q, b, r \rangle \). The machines for the expressions \( ba \) and \( bb \) look the same, with the labels changed to match the expressions.

For \( ab + ba \), we combine the two expressions for \( ab \) and \( ba \). Our construction does this by merging the two initial states and the two final states, giving state set \( \{ p, q, r, s \} \), start state \( p \), only final state \( s \), and transitions \( \langle p, a, q \rangle \), \( \langle p, b, r \rangle \), \( \langle q, b, s \rangle \), and \( \langle r, a, s \rangle \).

Then we go to \((ab + ba)^*\), which requires adding two new states and four new \( \epsilon \)-moves. We have a state set of size six\(^2 \{ \iota, p, q, r, s, f \} \), with start state \( \iota \), final state \( f \), and the transitions above plus \( \langle \iota, \epsilon, p \rangle \), \( \langle p, \epsilon, s \rangle \), \( \langle s, \epsilon, p \rangle \), and \( \langle s, \epsilon, f \rangle \). We can now finish by following the procedure for union on the machine for \((ab + ba)^*\) and the machine for \( bb \). Our method in effect adds one new state \( t \) and new transitions \( \langle \iota, b, t \rangle \) and \( \langle t, b, f \rangle \).

It’s a useful exercise to turn this \( \epsilon \)-NFA into an ordinary NFA and then into a DFA. You should get a seven-state minimal DFA out of it, with states \( \{ \iota \} \) (start and final), \( \{ q \} \), \( \{ p, s, f \} \) (final), \( \{ r \} \), \( \{ r, t \} \), \( \{ f \} \) (final), and \( \emptyset \) (Figure 14-37,14-38).

We now need only one step to complete the proof of Kleene’s Theorem, a construction converting \( \epsilon \)-NFA’s to regular expressions. We’ll see this in Section 14.10.

14.8.3 Exercises

E14.8.1 Consider the following alternate constructions for each operation, that preserve the property of a single final state but not our other conditions: For concatenation add a single \( \epsilon \)-move from the final state of one \( \epsilon \)-NFA to the start state of the other, for union add a new start state, a new final state and four \( \epsilon \)-moves, and for Kleene star add just two \( \epsilon \) moves, from the start to final state and vice versa. Carry out this alternate construction to get an \( \epsilon \)-NFA with language \((ab + ba)^* + bb\). You should get fifteen states as opposed to the seven in our construction above.

E14.8.2 Construct an \( \epsilon \)-NFA whose language is denoted by the regular expression \((aa)^*(ab + bb^*a)(b + ab^*a)^*\).

E14.8.3 Prove that for any language \( L \), \( (L + \epsilon)^* = L^* \).

E14.8.4 Give a simple two-state \( \epsilon \)-NFA for \( \{ \epsilon \} \) that meets all of our requirements. Of course, officially the regular expression “\( \epsilon \)” is just an abbreviation for “\( \emptyset^* \)”. Find the \( \epsilon \)-NFA obtained by custom-designing a \( \epsilon \)-NFA for this regular expression alone rather than using a general method, we could do it in three states: state set \( \{ x, y, z \} \), \( x \) is both start and final, transitions \( \langle x, a, y \rangle \), \( \langle y, b, x \rangle \), \( \langle x, b, z \rangle \), and \( \langle z, a, x \rangle \). But as usual (e.g., with a compiler), a general method does not produce output as efficient as that of customizing. This three-state NFA could not be used later in our procedure as it has edges into the start state and out of the final state.

\(^2\)If we were custom-designing a \( \epsilon \)-NFA for this regular expression alone rather than using a general method, we could do it in three states: state set \( \{ x, y, z \} \), \( x \) is both start and final, transitions \( \langle x, a, y \rangle \), \( \langle y, b, x \rangle \), \( \langle x, b, z \rangle \), and \( \langle z, a, x \rangle \). But as usual (e.g., with a compiler), a general method does not produce output as efficient as that of customizing. This three-state NFA could not be used later in our procedure as it has edges into the start state and out of the final state.
Figure 14-7: Construction of an $\epsilon$-NFA for the Language $(ab + ba)^* + bb$
Figure 14-8: An Ordinary NFA For the Language $(ab + ba)^* + bb$

Figure 14-9: A DFA For the Language $(ab + ba)^* + bb$
slavishly following our construction for $\emptyset^*$, and explain informally why it is equivalent to your
two-state $\epsilon$-NFA.

E14.8.5 Design (or recall) a two-state DFA whose language is $(b + ab*a)^*$. Apply the construction to
this regular expression to get an $\epsilon$-NFA with the same language, and compare the number of
states in the two automata.

14.8.4 Problems

P14.8.1 Formally prove the correctness of the union construction as follows. Let $M_1$ and $M_2$ be the
two $\epsilon$-NFA’s constructed for $R_1$ and $R_2$ (so they obey the assumptions) and let $N$ be the
$\epsilon$-NFA constructed so that $L(N) = R_1 + R_2$. Let $w$ be a string such that $\Delta^*_N(\iota, w, f)$. Prove
that either $\Delta^*_{M_1}(\iota_1, w, f_1)$ or $\Delta^*_{M_2}(\iota_2, w, f_2)$. Use induction on all strings $w$.

P14.8.2 Formally prove the correctness of our construction for the concatenation operator.

P14.8.3 Suppose $M$ is an $\epsilon$-NFA with exactly one final state $f$, but not necessarily satisfying any of
our assumptions. Make a new $\epsilon$-NFA $M'$ by adding two new transitions $(f, \epsilon, \iota)$ and $(\iota, \epsilon, f)$,
where $\iota$ is the start state. Is $L(M')$ always equal to $L(M)^*$? Either prove that it is or give
an example where it is not. Does adding our other assumptions change anything?

P14.8.4 Define the **length** of a regular expression by the rules $\ell(\emptyset) = 1$, $\ell(a) = 1$, $\ell(RS) = \ell(R) + \ell(S) + 2$, $\ell(R + S) = \ell(R) + \ell(S) + 3$, and $\ell(R^*) = \ell(R) + 3$. Let $s(R)$, for any regular
expression $R$, be the number of states in the $\epsilon$-NFA made from $R$ by our construction. Prove
by induction on all regular expressions $R$ that $s(R) \leq 2\ell(R)$.

P14.8.5 (Uses Java) Given the definitions of the class `RegExp` in Section 5.5 and of the class `DFA` in
Section 14.2, write a pseudo-Java method to implement the construction of this section.