FROM
NFA RE

Big-Oh Notation

An expression like $O(n)$ is a shorthand for "at most some constant times n." Technically, we say a function $f(n)$, perhaps the running time of some step of an algorithm, is $O(g(n))$ if there are constants c and n_0, such that whenever $n \geq n_0$, it is true that $f(n) \leq cg(n)$. A useful idiom is "$O(1)$," which means "some constant." The use of this *big-oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time, yet lets us express the rate at which the running time of an algorithm grows.

the state s_0. Lines (2), (7), and (8) each take $O(1)$ time. Thus, the running time of Algorithm 3.22, properly implemented, is $O((k(n+m))$. That is, the time taken is proportional to the length of the input times the size (nodes plus edges) of the transition graph.

3.7.4 Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that defines the same language. The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression. For each subexpression the algorithm constructs an NFA with a single accepting state.

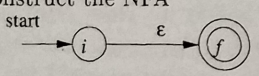
Algorithm 3.23: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

INPUT: A regular expression r over alphabet Σ.

OUTPUT: An NFA N accepting $L(r)$.

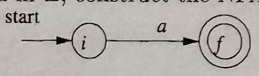
METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

BASIS: For expression ϵ construct the NFA



Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

For any subexpression a in Σ, construct the NFA



where again i and f are new states, the start and accepting states, respectively. Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of ϵ or some a as a subexpression of r.

INDUCTION: Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t, respectively.

a) Suppose $r = s|t$. Then $N(r)$, the NFA for r, is constructed as in Fig. 3.40. Here, i and f are new states, the start and accepting states of $N(r)$, respectively. There are ϵ-transitions from i to the start states of $N(s)$ and $N(t)$, and each of their accepting states have ϵ-transitions to the accepting state f. Note that the accepting states of $N(s)$ and $N(t)$ are not accepting in $N(r)$. Since any path from i to f must pass through either $N(s)$ or $N(t)$ exclusively, and since the label of that path is not changed by the ϵ's leaving i or entering f, we conclude that $N(r)$ accepts $L(s) \cup L(t)$, which is the same as $L(r)$. That is, Fig. 3.40 is a correct construction for the union operator.

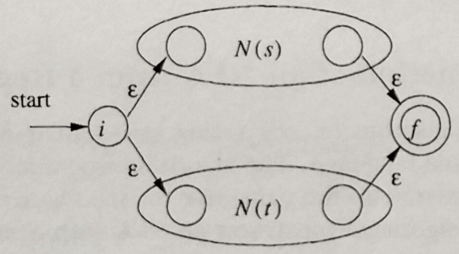


Figure 3.40: NFA for the union of two regular expressions

b) Suppose $r = st$. Then construct $N(r)$ as in Fig. 3.41. The start state of $N(s)$ becomes the start state of $N(r)$, and the accepting state of $N(t)$ is the only accepting state of $N(r)$. The accepting state of $N(s)$ and the start state of $N(t)$ are merged into a single state, with all the transitions in or out of either state. A path from i to f in Fig. 3.41 must go first through $N(s)$, and therefore its label will begin with some string in $L(s)$. The path then continues through $N(t)$, so the path's label finishes with a string in $L(t)$. As we shall soon argue, accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter $N(s)$ after leaving it. Thus, $N(r)$ accepts exactly $L(s)L(t)$, and is a correct NFA for $r = st$.

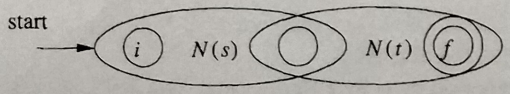


Figure 3.41: NFA for the concatenation of two regular expressions

c) Suppose $r = s^*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.42. Here, i and f are new states, the start state and lone accepting state of $N(r)$. To get from i to f, we can either follow the introduced path labeled ϵ, which takes care of the one string in $L(s)^0$, or we can go to the start state of $N(s)$, through that NFA, then from its accepting state back to its start state zero or more times. These options allow $N(r)$ to accept all the strings in $L(s)^1$, $L(s)^2$, and so on, so the entire set of strings accepted by $N(r)$ is $L(s^*)$.

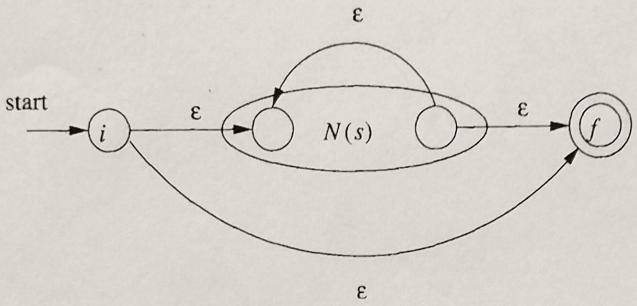


Figure 3.42: NFA for the closure of a regular expression

d) Finally, suppose $r = (s)$. Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.

□

The method description in Algorithm 3.23 contains hints as to why the inductive construction works as it should. We shall not give a formal correctness proof, but we shall list several properties of the constructed NFA's, in addition to the all-important fact that $N(r)$ accepts language $L(r)$. These properties are interesting in their own right, and helpful in making a formal proof.

1. $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.

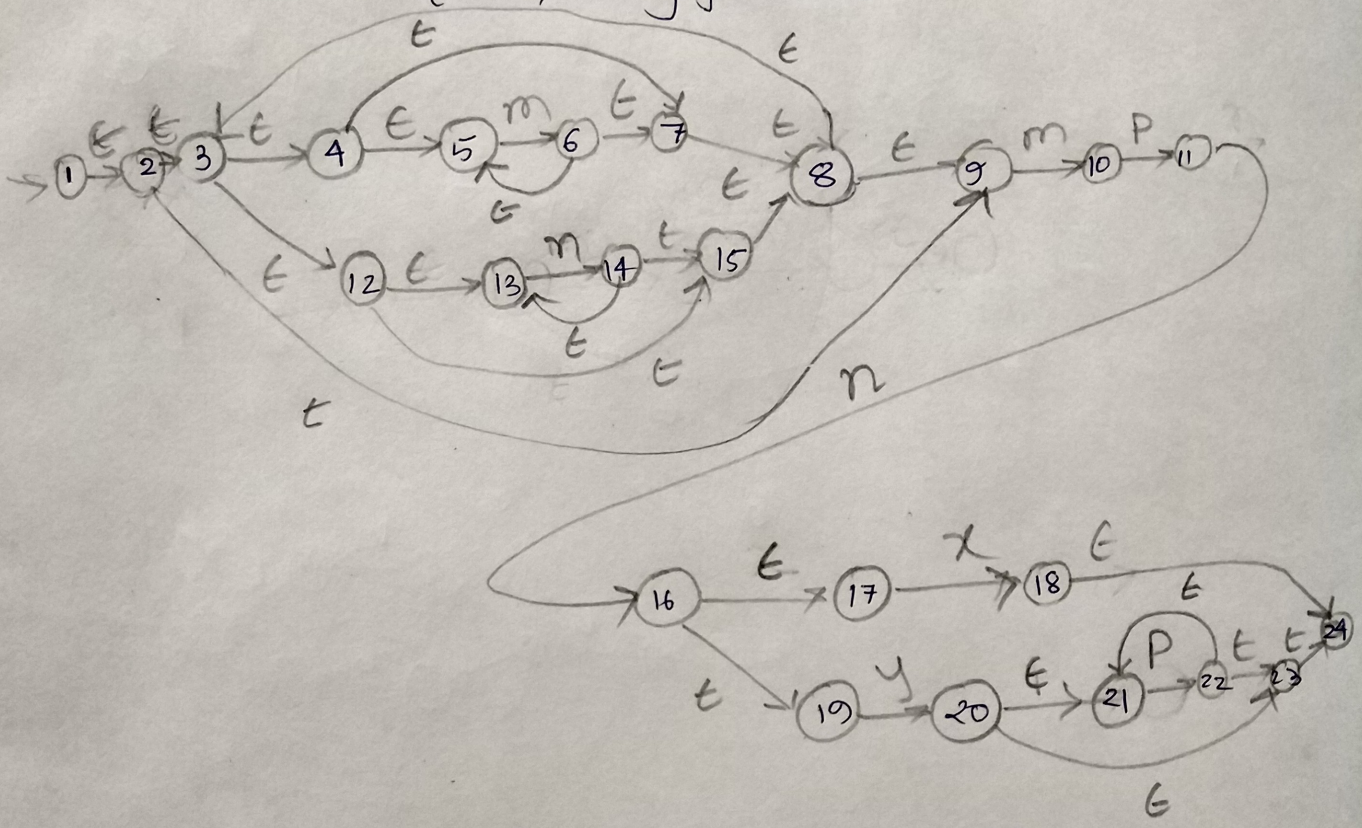
2. $N(r)$ has one start state and one accepting state. The accepting state has no outgoing transitions, and the start state has no incoming transitions.

3. Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in Σ or two outgoing transitions, both on ϵ.

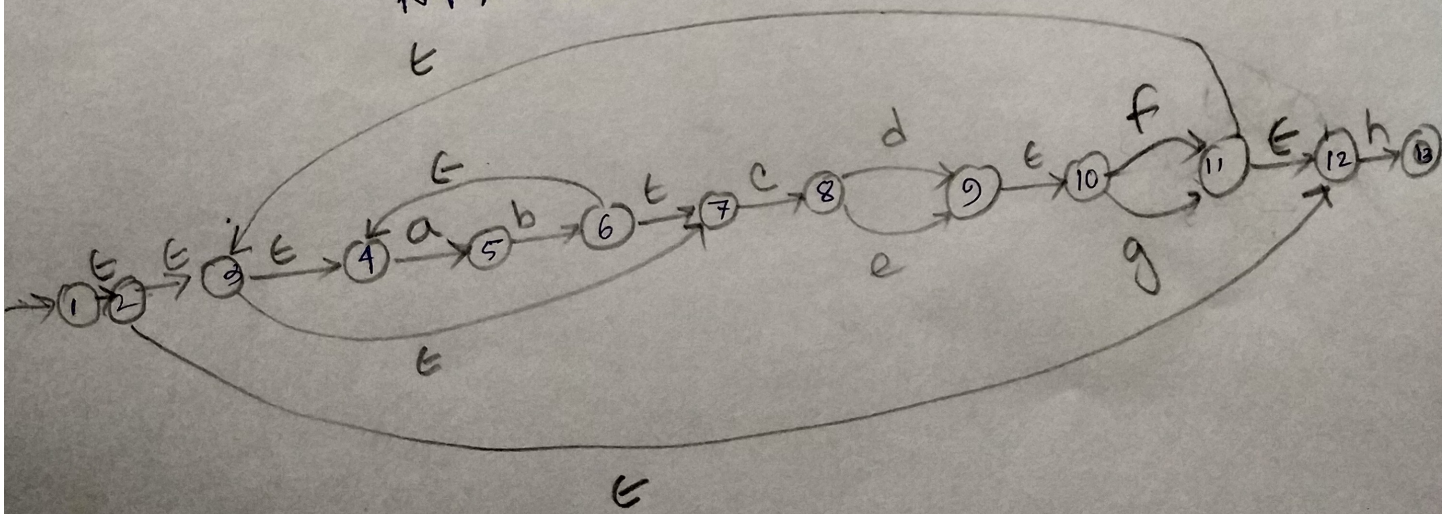
Example 3.24: Let us use Algorithm 3.23 to construct an NFA for $r = (\mathbf{a}|\mathbf{b})^*\mathbf{abb}$. Figure 3.43 shows a parse tree for r that is analogous to the parse trees constructed for arithmetic expressions in Section 2.2.3. For subexpression r_1, the first **a**, we construct the NFA:

RE to NFA:

Q1: Convert $(m^*+n^*)^* mpr(x+yp^*)$ over $\Sigma = \{m, n, p, r, x, y\}$ to NFA



Q2: Convert $(ab)^*c(d|e)(f|g)^*h$ to NFA



Q3: Given $S = k(m^*/n^*)^* bb(k|b)^*$, give the ϵNFA