CS/Math 240: Introduction to Discrete Mathematics 4/21/2011

Lecture 23 : Nondeterministic Finite Automata

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we designed finite state automata for a few more languages. After that, we started discussing regular expressions and languages that they describe. Today we show that every language that can be described by a regular expression can be recognized by some finite state automaton.

23.1 Connection between Regular Expressions and Finite Automata

Regular expressions and finite automata define the same class of languages. We now formalize this equivalence of the expressive powers of regular expressions and finite automata.

Definition 23.1. *A language L is* regular *if and only if it can be defined by a regular expression, i.e., it can be written as $L(R)$ for some regular expression R.*

Theorem 23.2. *A language L is regular if and only if it is accepted by some finite automaton, i.e., there exists a finite automaton M such that $L = L(M)$.*

The proof of Theorem 23.2 consists of proving two implications, which we state as lemmas. We only prove the first implication today.

Lemma 23.3. *Every regular language can be decided by some finite automaton. That is, for every regular expression R, there is a finite automaton M such that $L(R) = L(M)$.*

Lemma 23.4. *For every language L decidable by some finite state automaton, there is a regular expression R such that $L = L(R)$.*

23.1.1 A Note on Accepting

Let's conclude this section with a different way of describing what it means for a finite state automaton to accept. This will be useful later in this lecture. Consider a graph representation of a finite state automaton M. We say a path $e_1, e_2, \ldots, e_n$ is labeled by a string $x = x_1 x_2 \ldots x_n$ if edge e_i is labeled by x_i for $i \in \{1, \ldots, n\}$. The machine M accepts x if there is a path from the start state to an accepting state that is labeled by x, and rejects otherwise.

Example 23.1: Consider the machine M_1 from Figure 23.2a. The only path labeled with the string 010 starts at α and the next vertices on this path are α, β, α. This path does not lead to an accepting state, so M_1 rejects x.

On the other hand, M_1 accepts $x = 001$ because the path labeled by this string starts in state α and the next three states it visits are α, α, β. The last state on the path is accepting. ⊠

23.2 Finite Automata from Regular Expressions

We start with the proof of Lemma 23.3. Regular expressions are defined inductively, and we exploit this definition in a proof by structural induction. Given a regular expression R, we construct a finite automaton M such that $L(R) = L(M)$. All regular expressions we discuss will be over an alphabet Σ.

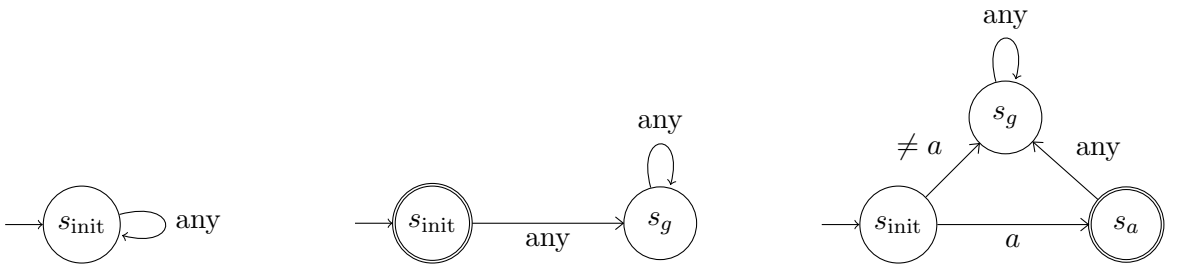
23.2.1 Base Cases

Recall that there are three elementary regular expressions, namely $\emptyset$, ϵ, and a for $a \in \Sigma$. Their corresponding languages are $\emptyset$, $\{\epsilon\}$, and $\{a\}$ for $a \in \Sigma$, respectively.

The empty regular expression corresponds to the empty language. One automaton that decides this language is the automaton that starts in some non-accepting state and stays in that state for the entirety of its computation. We show it in Figure 23.1a.

The regular expression ϵ corresponds to the language consisting only of the empty string. This is accepted by the automaton that starts in an accepting state, and moves to a rejecting state on any input. It stays in that rejecting state until it processes the entire input. We show this automaton in Figure 23.1b.

Finally, see the automaton for the language of the regular expression a in Figure 23.1c. It starts in a rejecting state since ϵ is not in the language. From there, it goes to an accepting state if the input is a, and goes to a rejecting state otherwise. For any additional input, the automaton goes to a rejecting state and stays there. Thus, the only way for the automaton to get to an accepting state is if the first and only symbol in the input is a.



(a) Automaton for $L(\emptyset)$. (b) Automaton for $L(\epsilon)$. (c) Automaton for $L(a)$ with $a \in \Sigma$.

Figure 23.1: Automata for the three simple regular expressions.

For the induction step, we need to show that if we have automata M_1 and M_2 for languages $L(R_1)$ and $L(R_2)$, respectively, we can construct finite state automata N_1, N_2 and N_3 such that $L(R_1 \cup R_2) = L(N_1)$, $L(R_1 R_2) = L(N_2)$, and $L(R_1^*) = L(N_3)$. We do so in the rest of this lecture.

We remark that you designed automata for some languages obtained by concatenation on the last homework. Constructing automata for those languages required some insights. By getting a deeper understanding of the structure of a language, we can often construct very small automata that recognize it. For the inductive step of the proof of Lemma 23.3, we present general constructions one can use to construct automata for any union, concatenation, and Kleene closure of languages.

23.2.2 Union

Let R_1 and R_2 be regular expressions over the alphabet Σ. It is possible to handle the case where the two regular expressions are over different alphabets, but for now let's agree that both are over the same alphabet.

Suppose $M_1 = (S_1, \Sigma, \nu_1, s_1, A_1)$ and $M_2 = (S_2, \Sigma, \nu_2, s_2, A_2)$ are such that $L(R_1) = L(M_1)$ and $L(R_2) = L(M_2)$. We combine M_1 and M_2 into a finite state automaton $N_1 = (S, \Sigma, \nu, s_0, A)$ that recognizes $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$. The finite automaton N_1 should accept a string x if and only if at least one of M_1 and M_2 accepts x.

Let's start with a construction that does not work. To see if $x \in L(R_1 \cup R_2)$, we could run the machine M_1 on input x, and then run M_2 on input x. We accept if at least one of the machines accepts. Unfortunately, we cannot model this procedure using a finite state automaton as it requires us to read x twice. Finite automata cannot "rewind" the input halfway through and start reading it again from the beginning.

Instead of running M_1 and M_2 in a sequence, we can run them in parallel. For that to work, we need to keep track of the state of both machines at the same time, and update the state of both machines after reading a symbol from the alphabet. To keep track of the states of M_1 and M_2 at the same time, we create a state for each pair $(s_1, s_2) \in S_1 \times S_2$. Since S_1 and S_2 are finite, $S = S_1 \times S_2$ is also finite (more specifically, $|S| = |S_1| \cdot |S_2|$).

Now let's design the transition function. Say N_1 is in state (t_1, t_2) and reads the input symbol a. The first component of the new state should correspond to the state M_1 goes to on input a from state t_1, and the second component of the new state should correspond to the state M_2 goes to on input a from state t_2. Thus, we have We have $\nu((t_1, t_2), a) = (\nu_1(t_1, a), \nu_2(t_2, a))$.

We start running M_1 in state s_1, and M_2 in state s_2, so the start state is $s_0 = (s_1, s_2)$.

Finally, N_1 should accept if at least one of M_1 or M_2 accepts. If M_1 is in an accepting state t_1, M_2 can be in any state. This means that if $t_1 \in A_1$, $(t_1, t_2) \in A$ for any $t_2 \in S_2$. Similarly, if the state of M_2 is $t_2 \in A_2$, it doesn't matter what the state of M_1 is, so any state of the form (t_1, t_2) with $t_1 \in S_1$ and $t_2 \in A_2$ should be accepting. In other words, $A = A_1 \times S_2 \cup S_1 \times A_2$. The first part of the union takes care of the states where M_1 accepts, and the second part takes care of the states where M_2 accepts.

Example 23.2: Let M_1 and M_2 be the machines from Lecture 21 that accept all binary strings with an odd number of ones and all binary strings that start and end with the same symbol, respectively. Let's use the procedure we described to construct the automaton for $L(M_1) \cup L(M_2)$. The result is in Figure 23.2.

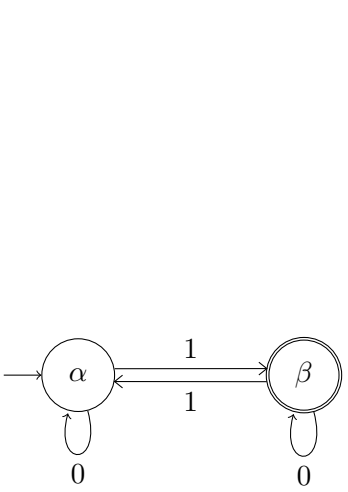
Note that $|S_1 \times S_2| = 10$, but we only have 9 states in Figure 23.2c. That is because the state (β, F) is not reachable from the start state (α, F). Machine M_1 can only get to state β after reading at least one input symbol, whereas M_2 leaves state F right after reading the first symbol and never returns to that state. ⊠

23.2.3 Nondeterministic Finite Automata

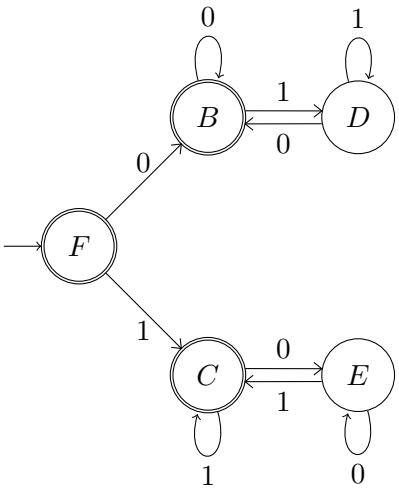
Assuming the same notation as in the previous section, we now describe how to construct a finite state automaton N_2 that accepts the concatenation $L(R_1)L(R_2)$ assuming that we have automata M_1 and M_2 that accept $L(R_1)$ and $L(R_2)$, respectively.

We outline an initial attempt that runs the two machines in a sequence. Combine two machines M_1 and M_2 into N_2 with the set of states $S = S_1 \cup S_2$. Start in the start state of M_1, and eventually move to some state of M_2. If the empty string does not belong to $L(M_2)$, is undesirable for N_2 to accept while it's still in one of M_1's states because at that point it has not verified the input ends with a string in $L(R_2)$. Accepting in such a state may not be the right thing to do in this situation. Hence, if M_2 doesn't contain the empty string, only the states in A_2 are accepting. Otherwise the set of accepting states is $A_1 \cup A_2$.

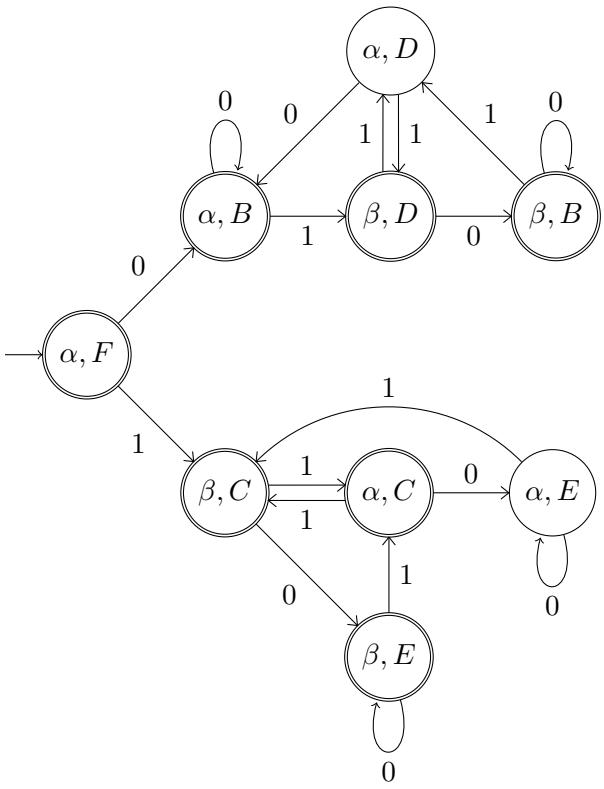
Note that M_1 and M_2 now run on separate parts of the input, so it is indeed possible to run them in a series. On the other hand, a different problem arises: When do we stop the computation of M_1 and start running M_2? We can start M_2's computation after M_1 reaches an accepting state. The transition on input a from that accepting state is to the state M_2 goes to on input a from its start state s_2. When we make this transition, we are indicating that we think the part of the



(a) Machine M_1.



(b) Machine M_2.



(c) Machine N_1.

Figure 23.2: Combining M_1 and M_2 into a machine N_1 for $L(M_1) \cup L(M_2)$.

input that belongs to $L(M_1)$ has ended, and that a is the first symbol of the part of the input that belongs to $L(M_2)$.

As we will now see, going to M_2 right after reaching an accepting state of M_1 for the first time may be a mistake.

Example 23.3: Let's design a finite automaton for the language $L(M_2)L(M_2)$ using the strategy we outlined in the previous paragraph. Recall from Lecture 22 that $L(M_2)L(M_2) = \Sigma^*$ so the

automaton we create should accept every string.

We make two copies of M_2. Let's call the second copy M_2'. Consider running the machine on input 00011. The machine M_2 starts in an accepting state. Thus, if we decide to move to a state of M_2' right after reaching an accepting state of M_2, we go to the state B' of M_2' right when we read the first symbol of the input. But now M_2' thinks the string starts with a zero, and it will end in the rejecting state D' when it's done processing the input.

We see that transitioning to M_2' as soon as possible is not the right move. We should transition to a state of M_2' when we read the first 1 in this input, in which case we end in the state C'. We stay in that state until the rest of the computation, and accept. Unfortunately, moving to M_2' after seeing the third zero also doesn't always work. For example, consider the input 000011. ⊠

The previous example should convince you that transitioning to M_2's states is not a trivial problem. In fact, for any strategy we choose in the previous example, there is a string that causes incorrect behavior. To remedy this, we allow multiple transitions on the same input. On input a in an accepting state of M_1, we allow both a transition to a state in M_1 on input a according to ν_1, and we also allow a transition to the state $\nu_2(s_2, a)$. Unfortunately, now the transitions are not defined by a function, so we don't have a finite automaton anymore. The transitions are now defined by a relation, which gives rise to a computational model known as the *nondeterministic finite automaton*. We show the nondeterministic finite automaton for the language $L(M_2)L(M_2)$ from Example 23.3 in Figure 23.3. Note that since $\epsilon \in L(M_2)$, the states corresponding to the first copy of M_2 remain accepting.

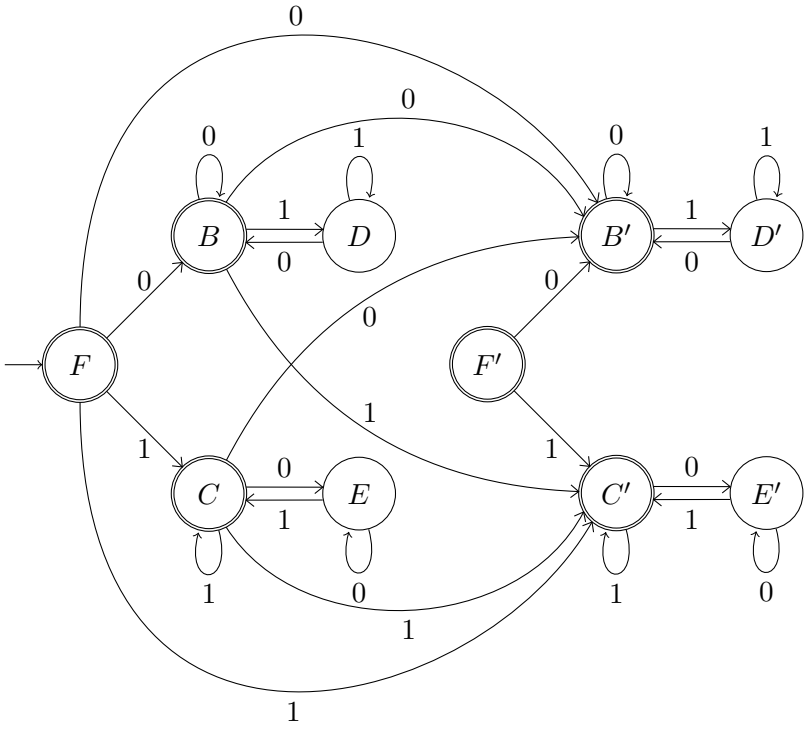


Figure 23.3: A nondeterministic finite automaton for the language $N_2 = L(M_2)L(M_2)$.

Definition 23.5. *A* nondeterministic finite automaton *is a* 5*-tuple* $N = (S, \Sigma, \nu, s_0, A)$ *where*

- S *is a finite set of* states.

- Σ *is a finite set of symbols called the* alphabet.
- *The* transition relation ν *is a relation from* $(S \times \Sigma)$ *to* S*, and* $((s, a), t) \in \nu$ *means that* N *can go from state* s *to state* t *if it reads symbol* a*. If there is no tuple* $((s, a), t) \in \nu$ *for some* s *and* a*,* N *rejects immediately without reading the rest of the input.*
- *The automaton starts in the* start state $s_0 \in S$*.*
- *The states in* $A \subseteq S$ *are the* accepting states*. The machine* N accepts x *if there is a path from* s_0 *to some state* $t \in A$ *that is labeled by* x*. The machine* N rejects x *otherwise.*

With the exception of the transition relation, Definition 23.5 is exactly the same as the definition of a finite state automaton. Also note that nothing really changes with the graph representation, except now multiple edges leaving a vertex can have the same label.

This justifies why we defined acceptance using paths in a graph in Section 23.1.1. There could now be multiple paths from the start state that are labeled with a string x. If any one of them leads to an accept state, the machine accepts. Otherwise, the machine rejects. It is also possible that there is no path labeled with x, in which case the nondeterministic finite automaton rejects.

Example 23.4: Consider the nondeterministic finite automaton in Figure 23.4 that operates on the alphabet $\{0, 1\}$. Observe that there is no transition from some state s_1 on input 1, so if it gets to state s_1 and reads a 1, it rejects. On the other hand, on any input x with a 1 in it, there is a path to s_1 that is labeled by x: Stay in s_0, and transition to s_1 on the last occurrence of 1 in x. Thus, the automaton accepts all strings that contain a 1, and rejects all strings without ones.

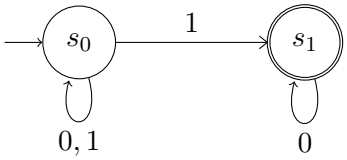


Figure 23.4: A nondeterministic finite automaton for Example 23.4.

⊠

A nondeterministic automaton is no longer a realistic model of computation because it has the capability of arbitrarily choosing which path to take, which is something a computer cannot do. One way of thinking about running a nondeterministic finite automaton is that it's a process which spawns off a copy of itself for each valid transition, and each copy of the process makes a different transition and continues computing on its own. It then suffices if one copy of this process accepts. Another way to think about it is in terms of graphs. The existence of a path from a start state labeled x and ending in an accept state implies that x is in the language. Yet another way to think about it is that the machine somehow magically knows which path to follow in each case. If there is an accepting path, the machine picks a transition that follows that path whenever it has multiple transitions to choose from.

23.2.4 Concatenation

Let's now go back to designing a finite state automaton for the language $L(R_1R_2)$ out of M_1 and M_2. The strategy we described in the previous section gives us a generic way of constructing an automaton for such a language. In fact, the strategy we described works even if we have two

nondeterministic finite automata N_a and N_b and we want to construct a nondeterministic finite automaton for the language $L(N_a)L(N_b)$. We present this more general construction.

Let $M_1 = (S_1, \Sigma, \nu_1, s_1, A_1)$ and $M_2 = (S_2, \Sigma, \nu_2, s_2, A_2)$ be nondeterministic finite automata. The nondeterministic finite automaton $N_2 = (S, \Sigma, \nu, s_0, A)$ for the language $L(M_1)L(M_2)$ is defined as follows.

- $S = S_1 \cup S_2$
- $\nu = \nu_1 \cup \nu_2 \cup \{((s, a), t) \mid s \in A_1, ((s_2, a), t) \in \nu_2\}$
- $s_0 = s_1$
- $A = \begin{cases} A_2 & \epsilon \notin L(M_2) \\ A_1 \cup A_2 & \text{otherwise} \end{cases}$

We are not done yet because we want a deterministic finite automaton for $L(R_1R_2)$. The following theorem which we will prove next time completes the construction.

Theorem 23.6. *Let N be a nondeterministic finite automaton. Then there exists a finite state automaton M such that $L(N) = L(M)$.*

23.2.5 Kleene Closure

Finally, let R_1 be a regular expression and $M_1 = (S_1, \Sigma, \nu_1, s_1, A_1)$ a finite state automaton such that $L(M_1) = L(R_1)$. We describe how to use M_1 in the construction of an automaton N_3 that accepts the language of the regular expression R_1^*. We only construct a nondeterministic automaton and then appeal to Theorem 23.6.

We use some ideas from the construction of a nondeterministic finite automaton that recognizes the concatenation of two languages. Unfortunately, we cannot just repeat the construction multiple times because this would require an infinite amount of states. Instead, we show that one copy of the automaton M_1 that recognizes $L(R_1)$ is sufficient.

We construct N_3 as a copy of M_1 with some additional transitions. When M_1 is in an accepting state and receives input a, we allow it to transition to any state t that satisfies $((s_0, a), t) \in \nu_1$ (with this notation, it also follows that M_1 can be nondeterministic for this construction to work). This allows N_3 to decide that the last symbol it read from an input z was the first symbol of the next string in $L(R)$ that is used as part of z.

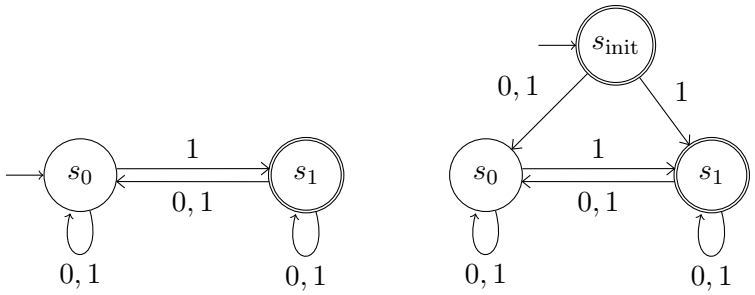
Notice that the empty string belongs to $L(R^*)$ no matter was R is, so if the start state of M_1 is rejecting, our construction so far fails to accept the empty string. We can remedy that by adding a start state s_{init} that is accepting and to which N_3 is never going to return. Then on input a, we add a transition from s_{init} to state t if $((s_1, a), t) \in \nu_1$. This completes the construction of N_3.

Example 23.5: Since the construction we described applies to nondeterministic finite automata, let's illustrate it on the automaton N from Figure 23.4.

First, we disregard the empty string and only add transitions from the accepting state s_1 of machine N. In particular, we add the transitions $((s_1, 0), s_0)$, $((s_1, 1), s_0)$ and $((s_1, 1), s_1)$ to the transition relation. This is shown in Figure 23.5a.

We complete the construction by adding a state s_{init} which accepts the empty string in addition to all other strings in $L(N)^*$. We add the transitions $((s_{\text{init}}, 0), s_0)$, $((s_{\text{init}}, 1), s_0)$ and $((s_{\text{init}}, 1), s_1)$ to the transition relation. The result is in Figure 23.5b. ⊠

Here is a formal description of the automaton $N_3 = (S, \Sigma, \nu, s_0, A)$ for $L(R_1^*)$ where the nondeterministic finite automaton $M_1 = (S_1, \Sigma, \nu_1, s_1, A_1)$ satisfies $L(M_1) = L(R_1)$.



(a) Adding transitions allowing to start processing the next string in the concatenation.

(b) Adding a new start state to ensure that our automaton accepts the empty string.

Figure 23.5: Turning the automaton N from Figure 23.4 into an automaton that recognizes the language $L(N)^*$.

- $S = S_1 \cup \{s_{\text{init}}\}$
- $\nu = \nu_1 \cup \{((s_1, a), t) \mid ((s_1, a), t) \in \nu_1 \wedge s_1 \in A_1\} \cup \{((s_{\text{init}}, a), t) \mid ((s_1, a), t) \in \nu_1\}$
- $s_1 = s_{\text{init}}$
- $A = A_1 \cup \{s_{\text{init}}\}$

Let's argue that $L(R_1^*) \subseteq L(N_3)$. Since $L(R_1^*) = L(R_1)^* = \bigcup_{k=0}^{\infty} L(R_1)^k$, it suffices to show that $L(R_1)^k \subseteq L(N_3)$ for all $k \in \mathbb{N}$. We argue by induction.

For the base case $L(R_1)^0 = \{\epsilon\}$, note that N_3 starts in an accepting state, so it accepts the empty string. Therefore, $L(R_1)^0 \subseteq L(N_3)$.

Now assume that $L(R_1)^k \subseteq L(N_3)$, and consider a string $x \in L(R_1)^{k+1}$. We can write x as the concatenation $x = x_1 x_2 \dots x_k x_{k+1}$ where $x_i \in L(R_1)$ for $i \in \{1, \dots, k+1\}$. Let $x' = x_1 x_2 \dots x_k$. Then we have $x = x' x_{n+1}$ where $x' \in L(R_1)^k$ and $x_{n+1} \in L(R_1)$. Since $L(R_1)^k \subseteq L(N_3)$ by the induction hypothesis, there is a path labeled by x' that starts in the start state of N_3 and ends in an accepting state t of N_3. The automaton N_3 is in this state when it starts processing x_{n+1}. Since the state is accepting, N_3 correctly accepts if $x_{n+1} = \epsilon$ and $\epsilon \in L(R_1)$. When $x_{n+1} \neq \epsilon$, there is a path labeled by x_{n+1} that starts in M_1's start state s_1 and ends in some accepting state t'. We constructed N_3 so that for any input symbol a, it can get from any of its accepting states to any state of M_1 that is reachable from s_1 if the next input symbol is a. Furthermore, since N_3 contains a copy of M_1 in it, it can follow the same path as M_1 after processing the first symbol a, and thus can reach an accepting state if M_1 can. This means that $x \in L(N_3)$. Now the proof that $L(R_1^*) \subseteq L(N_3)$ is complete.

Now we argue by induction on the length of a string in $L(N_3)$ that $L(N_3) \subseteq L(R_1)^*$.

For the base case, note that $\epsilon \in L(N_3)$ because N_3 starts in an accepting state. We also have $\epsilon \in L(R)^0$, so $\epsilon \in L(R_1)^*$, and the base case is proved.

Now assume that every string y of length at most n that belongs to $L(N_3)$ belongs to $L(R_1)^*$. In other words, for every y of length at most n, there is an integer $k \in \mathbb{N}$ such that $y \in L(R_1)^k$. Consider $x \in L(N_3)$ such that $|x| = n+1$, and look at a path labeled by x that leads from the state s_{init} to an accepting state of N_3. Consider the last time on the path when N_3 goes from an accepting state on input symbol a to a state that is reachable from M_1's start state s_1 on input symbol a. If the last time is after reading the initial symbol, this means that N_3 only uses transitions that are present in M_1 after reading the initial symbol, and M_1 can get to the same state as N_3 on the

initial input symbol by construction. Thus, M_1 accepts the input, so $x \in L(M_1)$, which means $x \in L(R_1)$, and, therefore, $x \in L(R_1)^*$. Otherwise the last time happens after reading some symbol x_i for $i > 1$, and N_3 moves to state t after reading x_i. By construction, N_3 can go to state t from its start state s_{init} on input x_i. But then there is a path labeled by the string $x_i x_{i+1} \dots x_{n+1}$ of length at most n from s_{init} to an accepting state of N_3, which means $x_i x_{i+1} \dots x_{n+1} \in L(N_3)$, and this string also belongs to $L(R_1)^k$ for some k by the induction hypothesis. Furthermore, the string $x_1 \dots x_{i-1}$ of length at most n is also a string that labels a path from s_{init} to an accepting state of N_3, which means that $x_1 \dots x_{i-1} \in L(R_1)^\ell$. But then $x \in L(R_1)^{\ell+k}$, and we are done.