CS/Math 240: Introduction to Discrete Mathematics 4/26/2011

Lecture 24 : Finite Automata vs Regular Expressions

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we started discussing the equivalence of the expressive powers of regular expressions and finite state automata. We proved that any language that is described by a regular expression is accepted by some nondeterministic finite automaton, and then quoted a theorem stating that if a language is accepted by a nondeterministic finite automaton, it is also accepted by some finite state automaton. We proved that theorem today, and then show that if a language is accepted by some finite state automaton, it can also be described by a regular expression.

24.1 Equivalence of Nondeterministic and Deterministic Automata

Recall we are working towards proving the following theorem.

Theorem 24.1. *A language L can be written as $L(R)$ for some regular expression R if and only if L is accepted by some finite automaton M, i.e., L can be written as $L(M)$.*

Last time we proved the following lemma by structural induction.

Lemma 24.2. *For every regular expression R, there is a finite automaton M such that $L(R) = L(M)$.*

For the base cases, we designed automata for the languages $L(\emptyset)$, $L(\epsilon)$ and $L(a)$ for $a \in \Sigma$. For the induction step, we assumed the existence of automata M_1 and M_2 for $L(R_1)$ and $L(R_2)$, and designed automata N_1, N_2 and N_3 such that $L(N_1) = L(R_1 \cup R_2)$, $L(N_2) = L(R_1R_2)$ and $L(N_3) = L(R_1^*)$. We only constructed nondeterministic finite automata for the last two languages, so what we really showed was the following.

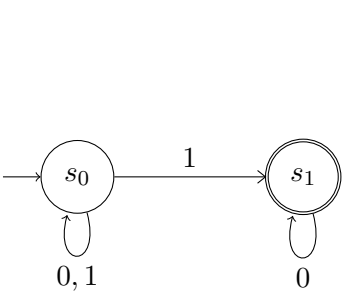
Lemma 24.3. *For every regular expression R, there is a nondeterministic finite automaton M such that $L(R) = L(M)$.*

To complete the proof of Lemma 24.2, we need the following theorem.

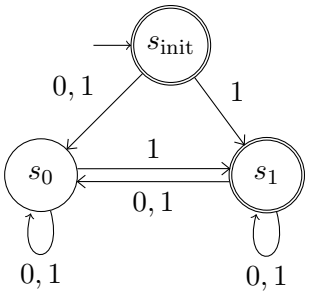
Theorem 24.4. *Let N be a nondeterministic finite automaton. Then there exists a finite state automaton M such that $L(N) = L(M)$.*

Example 24.1: Last time we constructed a nondeterministic finite automaton for the Kleene closure of a language. We show the original automaton N in Figure 24.1a, and the automaton N' for the Kleene closure of $L(N)$ in Figure 24.1b. We construct a finite state automaton M that accepts the same language as N'.

The main part of the construction of any automaton is to decide what states it should have. We need to keep track of information about N' that allows us to decide whether it accepts or not. The automaton N' accepts a string x if there is a path labeled by x that starts in the start state of N' and ends in one of its accept states. If we find all possible states N' can reach after processing x, we can decide whether N' accepts or not simply by checking whether one of the reachable states is accepting. We show this for input $x = 011010$ in Figure 24.1c. We see that one of the states reachable by N' after processing x is s_1 and this state is accepting, so N' accepts x. ⊠



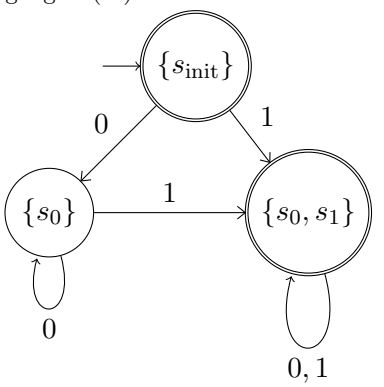
(a) The nondeterministic finite automaton N for Example 24.1.



(b) The nondeterministic finite automaton N' for the language $L(N)^*$.

Symbol	States
	s_{init}
0	s_0
1	s_0, s_1
1	s_0, s_1
0	s_0, s_1
1	s_0, s_1
0	s_0, s_1

(c) States reachable by N' after processing each symbol of 011010.



(d) A finite automaton M' such that $L(M') = L(N')$.

Figure 24.1: Converting a nondeterministic finite automaton to a deterministic finite automaton.

Proof of Theorem 24.4. Let $N = (S, \Sigma, \nu, s_0, A)$. We describe a deterministic finite state automaton $M = (S', \Sigma, \nu', s_0', A')$ such that $L(N) = L(M)$.

We keep track of all possible states N can be in after having processed the input, so define $S' = \mathcal{P}(S)$.

Suppose that T is the set of states N can be in after processing some initial portion of the input, and let a be the next symbol from the input that N reads. For any state $t \in T$, N can go to any state $s \in S$ such that $((t, a), s) \in \nu$. Now T is a state of M, and the state M goes to from there is $\{s \in S \mid (\exists t \in T)((t, a), s) \in \nu\}$, i.e., the state represented by the set of all states of N reachable from some state in T on input a. This gives us the following formal description of ν' for a state $s' \in S'$ and $a \in \Sigma$: $\nu'(s', a) = \{t \in S \mid (\exists s \in s')((s, a), t) \in \nu\}$.

When N starts its computation, it's in state s_0 and cannot be in any other state, so the start state of M is $s_0' = \{s_0\}$.

Since N accepts as long as it reaches some accepting state, any subset of S that contains an accepting state of N should correspond to an accepting state of M, so we have $A' = \{s' \in S' \mid s' \cap A \neq \emptyset\}$. This completes the construction of M.

We now argue that $L(M) = L(N)$. In particular, we prove the following invariant: After processing n symbols of the input x, the state of s' of M is labeled by the set of all possible states N can reach after reading the first n symbols of x.

For the base case, consider the situation before reading the first input symbol. Then N is in its start state s_0, and M is in the state $s_0' = \{s_0\}$. Thus, the invariant holds at the start of the

computation.

Now suppose the invariant holds after processing the first n symbols of the input, and consider processing the $(n+1)$st symbol, a, of the input. The machine M is in state s', which is the set of states N can be in after reading the first n symbols of x by the induction hypothesis. After reading a, N can reach any state t such that $((s,a),t) \in \nu$ where $s \in s'$ is a state N can currently be in. But this set is exactly $\nu'(s',a)$, so the invariant holds after processing a. This completes the proof of the invariant.

After processing the entire string x, the state of M represents the set of all possible states N can reach after processing x by the invariant we just proved. Say M is in state s' after processing x. If N can reach an accepting state s, $s \in s'$, so s' is accepting by construction. On the other hand, if N cannot reach an accepting state on input x, it rejects. Also, s' doesn't contain any accepting state of N, and is, therefore, rejecting as well. It follows that M accepts x if and only if N accepts x, so $L(M) = L(N)$. □

This completes the proof of Lemma 24.2.

We remark that the number of states in the construction of M in the proof of Theorem 24.4 is exponential in terms of the number of states of N since $|S'| = |\mathcal{P}(S)| = 2^{|S|}$. One may ask whether this amount of states is excessive. In general, it an exponential number of states may be necessary, and you will show this on your tenth homework. In some cases, it is possible to get away with much fewer states. We illustrate that on an example.

Example 24.2: Consider the automaton N' from Figure 24.1b. We can construct a finite state automaton M' with three states such that $L(M') = L(N')$. The automaton M' has the same number of states as N'.

First create the state $\{s_{\text{init}}$ that represents the only state N' can be in when it starts computing. Now on input 0 in s_{init}, N' can only go to state s_0, so we add he state $\{s_0\}$ to M, and a transition to that state from s_{init} on input 0. When in s_{init}, N' can go to states s_0 and s_1, so we add the state $\{s_0, s_1\}$ to M, and a transition to it from s_{init} on input 1.

Now let's figure out the transitions from the states $\{s_0\}$ and $\{s_0, s_1\}$. We see from Figure 24.1b that the only state N' can go to from state s_0 on input 0 is s_0, so M stays in state $\{s_0\}$ on input 0. If N receives input 1 in state s_0, it can either stay in s_0 or go to s_1. It cannot go to any other state, so M goes from $\{s_0\}$ to $\{s_0, s_1\}$ on input 1. Since N can go to states s_0 and s_1 on input 1 from s_0, and it cannot get to state s_{init} from anywhere, M stays in $\{s_0, s_1\}$ on input 1. Finally, if N is in state s_0 or s_1, it can choose to stay in its current state on input 0, and cannot go to s_{init}, so M stays in the state $\{s_0, s_1\}$ on input 0. Note that this exhausts all possible transitions M can make, and it didn't require us to add any more states to M.

To complete the construction, we make states $\{s_{\text{init}}\}$ and $\{s_0, s_1\}$ accepting. The complete automaton is in Figure 24.1d. ⊠

Example 24.3: Consider using the generic construction from the proof of Theorem 24.4 applied to the nondeterministic finite automaton N from Figure 24.1a. One of the states created by this construction is the state $\{s_1\}$. Note that there is no transition from s_1 on input 1 in N, so the equivalent deterministic finite automaton M goes to the state labeled by the empty set when it receives the input 1 in state $\{s_1\}$. We leave out all the remaining details and only show the final automaton obtained using the generic construction in Figure 24.2. ⊠

We remark that although nondeterministic finite automata have the power of guessing, they accept the same set of languages as deterministic finite automata. The only difference is an exponential blowup in the number of states, but raising 2 to a constant is still only a constant, so we don't necessarily mind the exponential blowup.

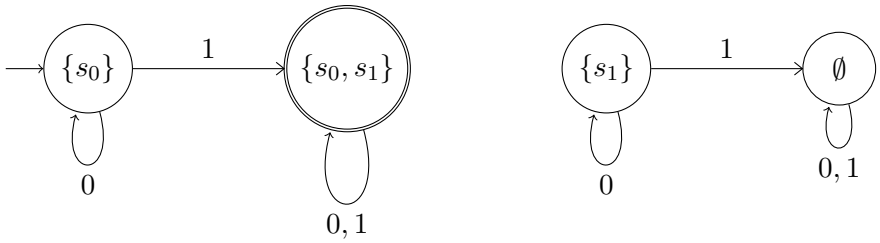


Figure 24.2: The deterministic finite automaton M that accepts the same language as the nondeterministic finite automaton N from Figure 24.1a.

Nondeterminism is connected to satisfiability and the P versus NP problem. There is an efficient nondeterministic algorithm for satisfiability: Just guess an assignment, and check if the formula is satisfied by it. This algorithm runs in nondeterministic polynomial time (this explains the name NP: it's the class of problems that can be solved in nondeterministic polynomial time). The conjecture is that no (deterministic) polynomial time algorithm (P is the class of problems that can be solved in deterministic polynomial time) for satisfiability, but no proof of this is known.

24.2 Regular Expressions from Finite Automata

We now prove the other part of Theorem 24.1.

Lemma 24.5. *For every language L decidable by some finite state automaton, there is a regular expression R such that $L = L(R)$.*

Before we prove Lemma 24.5, let's start with some examples.

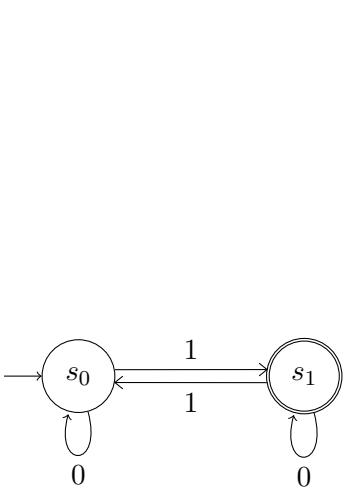
Example 24.4: Consider the automaton M_1 in Figure 24.3a that accepts binary strings with an odd number of ones. A regular expression that characterizes $L(M_1)$ must capture all strings that take M_1 from the start state s_0 to the accept state s_1.

One set of strings that brings M_1 from state s_0 to state s_1 is the set of strings that start with an arbitrary number (including none) zeros and end with a single 1. The regular expression that characterizes such strings is 0^*1. In fact, every string accepted by M_1 must start with a string that matches this regular expression.

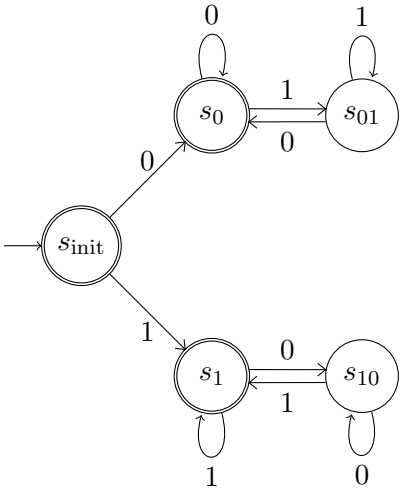
Now M_1 is in state s_1. It could leave this state and come back to it again later. The machine stays in state s_1 as long as it's reading zeros. After that, it reads a one and goes back to state s_0. A sequence of an arbitrary number of zeros followed by a 1 brings it back to s_1, and this is the only way it can happen. Thus, the strings that take M_1 from s_1 to s_1 via s_0 are given by the regular expression 0^*10^*1. Note that any number of such strings concatenated together form a string that labels a path from s_1 to s_1. Such a string matches the regular expression $(0^*10^*1)^*$.

Finally, at the very end after reading the last 1 of the input that takes M_1 to state s_1, there could be an arbitrary number of zeros in the input string. After reading those, M_1 is still in state s_1. Thus, the regular expression that characterizes $L(M_1)$ is $0^*1(0^*10^*1)^*0^*$. ⊠

Example 24.5: The machine M_2 that accepts string which start and end with the same symbol has multiple accepting states. For each accepting state, we need to find the set of strings that label some path from the starting state to that accepting state, and describe the set by a regular expression. Since M_2 can reach any accepting state to accept, we take the union of these regular expressions and obtain a regular expression that characterizes $L(M_2)$.



(a) Machine M_1 that accepts strings with an odd number of ones. We derive a regular expression for $L(M_1)$ in Example 24.4.



(b) Machine M_2 that accepts strings that start and end with the same symbol. We derive a regular expression for $L(M_2)$ in Example 24.5.

Figure 24.3: Some machines for which we construct regular expressions that characterize the languages they accept.

The start state is one accepting state. The machine M_2 cannot get back to it after reading an input symbol, so only the empty string labels a path from the start state to the start state. The corresponding regular expression is ϵ.

Another accepting state is the state s_0. To get to it, M_2 must read a zero as the first symbol in the input. It can read an arbitrary number of additional zeros and still stay in s_0. If it reads a 1, it moves to state s_{01} where it stays as long as it's reading additional 1s. It returns back to s_0 as soon as reads a zero in state s_{01}. The process of reading some number of zeros, then some number of ones, and then another zero can repeat an arbitrary number of times. Finally, after reading the last 1 in the input and returning to s_0, M_2 can read an arbitrary number of additional zeros to stay in s_0 and accept. Thus, the regular expression that characterizes the strings that label a path from the start state to s_0 is $0(0^*11^*0)^*0^*$.

By the same reasoning as in the previous paragraph, the regular expression $1(1^*00^*1)^*1^*$ characterizes all strings that label a path from the start state to the accepting state s_1.

Finally, we combine the three regular expressions we obtained and get the regular expression $\epsilon \cup 0(0^*11^*0)^*0^* \cup 1(1^*00^*1)^*1^*$ that characterizes $L(M_2)$. ⊠

The constructions in the last two examples were somewhat arbitrary. We now show a generic way of finding a regular expression R that describes the language of some finite state automaton M. The main idea is the same. For each accepting state, we find a regular expression that characterizes all strings that take M from the start state to that accepting state, and then take their union. The way we obtain those regular expressions is different, however. In the examples we presented, we reasoned about all possible paths that lead to an accepting state and constructed the regular expressions directly. But the automata we consider in the general case of Lemma 24.2 could be arbitrary complicated, so we build the regular expression systematically in steps. In particular, we initially restrict the states the automaton can use, and gradually relax the restriction by allowing the machine to use one additional state in each step of the construction.

Proof of Lemma 24.2. Let L be the language accepted by some finite state automaton M. Without loss of generality, label M's states with integers $1, 2, \ldots, |S|$. For each $i, j \in \{1, \ldots, |S|\}$ and $k \in \{0, 1, \ldots, |S|\}$, we construct a regular expression $R_{i,j,k}$ such that $L(R_{i,j,k})$ is the set of strings that bring M from state i to state j along a path on which the intermediate states have labels at most k.

In step k for $k \in \{0, \ldots, |S|\}$, we construct the regular expressions $R_{i,j,k}$ for all pairs of values of i and j.

In stage 0, a path from i to j can only go through states with label 0 or less. That is, no intermediate states are allowed. The set of strings that allows M to go from state i to state j with $i \neq j$ is then $X_{i,j} = \{a \mid \nu(i, a) = j\}$, so $R_{i,j,0} = \bigcup_{a \in X_{i,j}} a$. If $i = j$, an additional string that takes M from state i to j is the empty string (because in that case M just stays put). Thus, $R_{i,i,0} = \epsilon \cup \left(\bigcup_{a \in X_{i,j}} a\right)$.

Now consider stage $k + 1$. We have the regular expressions $R_{i,j,k}$ for all pairs of values of i and j, and we use them to build the regular expressions $Ri, j, k + 1$ for all pairs of values of i and j. So consider some i and j. A path from i to j that uses states labeled by $k + 1$ or less either uses state $k + 1$ or it doesn't. If it doesn't, it only uses states labeled by k or less, and the strings that label such a path are characterized by $R_{i,j,k}$. If the path uses the state $k + 1$, it first goes to $k + 1$ via states labeled k or less. This portion of the path is characterized by the regular expression $Ri, k + 1, k$. After that, M the path can visit $k + 1$ multiple times, and the path between each subsequent visit is a path from $k+1$ to $k+1$ using states labeled by k or less. The regular expression $R_{k+1,k+1,k}$ characterizes such a path. Finally, after its last visit to $k+1$, M follows a path to j using intermediate states labeled by k or less, and the regular expression that characterizes all strings that label such paths is $R_{k+1,j,k}$. Thus, the regular expression for the kind of path from i to j we just described is $R_{i,k+1,k} R^*_{k+1,k+1,k} R_{k+1,j,k}$. Since the two possibilities we described are the only two possibilities how M can get from state i to state j using states labeled by $k+1$ or less, we have $R_{i,j,k+1} = R_{i,j,k} \cup R_{i,k+1,k} R^*_{k+1,k+1,k} R_{k+1,j,k}$.

The construction stops after stage $|S|$ because at that point we have regular expressions characterizing all possible paths through the graph that represents the automaton M. From among those, we pick the regular expressions that characterize an accepting path from the start state of M to some accepting state of M. Thus, the final regular expression R is $R = \bigcup_{s \in A} R_{s_0,s,|S|}$. □

Example 24.6: We construct the regular expression for the language of the automaton in Figure 24.4, which is just the automaton M' from Figure 24.1d with its states renamed.

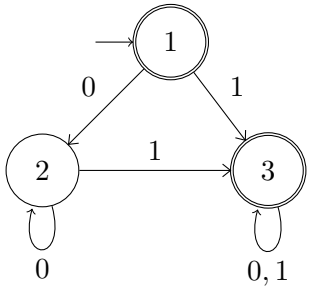


Figure 24.4

In stage 0, we construct the regular expressions $R_{i,j,0}$ for all pairs i and j. The only way to get to state 1 from state 1 using no intermediate states is on the empty path, so $R_{1,1,0} = \epsilon$. The only way to get to state 2 from state 1 without using any intermediate states is by reading a zero,

so $R_{1,2,0} = 0$. By similar reasoning, $R_{1,3,0} = 1$. There is no way to get from state 2 to state 1, so $R_{2,1,0} = \emptyset$. We summarize the remaining regular expressions from this step in Table 24.1a.

Let's also describe some of the constructions in step 1.

First, observe $R_{1,1,0}R^*_{1,1,0}R_{1,1,0} = \epsilon\epsilon^*\epsilon$. No matter how many times we concatenate the empty string with itself, we get the empty string, so $R_{1,1,0}R^*_{1,1,0}R_{1,1,0} = \epsilon$, and we have $R_{1,1,1} = R_{1,1,0} \cup R_{1,1,0}R^*_{1,1,0}R_{1,1,0} = \epsilon \cup \epsilon = \epsilon$.

Second, note $R_{2,1,0}R^*_{1,1,0}R_{1,2,0} = \emptyset\epsilon^*0$. Since the regular expression $\emptyset$ doesn't characterize any strings whatsoever, concatenating some regular expression with the empty regular expression also doesn't characterize any strings. It follows what $R_{2,1,0}R^*_{1,1,0}R_{1,2,0} = \emptyset$, so $R_{2,2,1} = R_{2,2,0} \cup R_{2,1,0}R^*_{1,1,0}R_{1,2,0} = 0 \cup \emptyset = 0$.

We leave the reasoning for all the remaining regular expressions to the reader. We made some simplifications in Table 24.1c describing step 2 of the construction to make the derivation of the final regular expressions in step 3 (shown in Table 24.1d) more readable. Note that even with the simplifications, some of the regular expressions, in particular $R_{1,3,3}$, become tedious already.

Finally, since the start state is 1 and the accepting states are 1 and 3, the regular expression R such that $L(R) = L(M')$ is $R = \epsilon \cup 1 \cup 00^*1 \cup (1 \cup 00^*1)(\epsilon \cup 0 \cup 1)^*(\epsilon \cup 0 \cup 1)$.

$\boxtimes$

i	j	$R_{i,j,0}$
1	1	ϵ
1	2	0
1	3	1
2	1	$\emptyset$
2	2	$\epsilon \cup 0$
2	3	1
3	1	$\emptyset$
3	2	$\emptyset$
3	3	$\epsilon \cup 0 \cup 1$

(a) Constructing $R_{i,j,0}$.

i	j	$R_{i,j,1}$
1	1	$\epsilon \cup \epsilon\epsilon^*\epsilon = \epsilon$
1	2	$0 \cup \epsilon\epsilon^*0 = 0$
1	3	$1 \cup \epsilon\epsilon^*1 = 1$
2	1	$\emptyset \cup \emptyset\epsilon^*\epsilon = \emptyset$
2	2	$(\epsilon \cup 0) \cup \emptyset\epsilon^*0 = \epsilon \cup 0$
2	3	$1 \cup \emptyset\epsilon^*1 = 1$
3	1	$\emptyset \cup \emptyset\epsilon^*\epsilon = \emptyset$
3	2	$\emptyset \cup \emptyset\epsilon^*0 = \emptyset$
3	3	$(\epsilon \cup 0 \cup 1) \cup \emptyset\epsilon^*1 = \epsilon \cup 0 \cup 1$

(b) Constructing $R_{i,j,1}$.

i	j	$R_{i,j,2}$
1	1	$\epsilon \cup \epsilon\epsilon^*\epsilon = \epsilon$
1	2	$0 \cup 0(\epsilon \cup 0)^*(\epsilon \cup 0) = 00^*$
1	3	$1 \cup 0(\epsilon \cup 0)^*1 = 1 \cup 00^*1$
2	1	$\emptyset \cup (\epsilon \cup 0)(\epsilon \cup 0)^*\emptyset = \emptyset$
2	2	$(\epsilon \cup 0) \cup (\epsilon \cup 0)(\epsilon \cup 0)^*(\epsilon \cup 0) = 0^*$
2	3	$1 \cup (\epsilon \cup 0)(\epsilon \cup 0)^*1 = 0^*1$
3	1	$\emptyset \cup \emptyset(\epsilon \cup 0)^*\emptyset = \emptyset$
3	2	$\emptyset \cup \emptyset(\epsilon \cup 0)^*(\epsilon \cup 0) = \emptyset$
3	3	$(\epsilon \cup 0 \cup 1) \cup \emptyset(\epsilon \cup 0)^*1 = \epsilon \cup 0 \cup 1$

(c) Constructing $R_{i,j,2}$.

i	j	$R_{i,j,3}$
1	1	ϵ
1	2	00^*
1	3	$(1 \cup 00^*1) \cup (1 \cup 00^*1)(\epsilon \cup 0 \cup 1)^*(\epsilon \cup 0 \cup 1)$
2	1	$\emptyset$
2	2	$0^* \cup 1(\epsilon \cup 0 \cup 1)\emptyset = 0^*$
2	3	$1 \cup (\epsilon \cup 0)(\epsilon \cup 0)^*1 = 0^*1$
3	1	$\emptyset$
3	2	$\emptyset$
3	3	$(\epsilon \cup 0 \cup 1) \cup (\epsilon \cup 0 \cup 1)(\epsilon \cup 0 \cup 1)^*(\epsilon \cup 0 \cup 1) = \{0,1\}^*$

(d) Constructing $R_{i,j,3}$. We make no attempt to simplify some of the regular expressions and omit the intermediate steps in some other cases.

Table 24.1: Constructing a regular expression that characterizes $L(M')$ where M' is the automaton in Figure 24.4.