Formal Languages and Automata Theory
- Regular Expressions and Finite Automata -

Samarjit Chakraborty
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zürich

March 17, 2003

Contents

1 Why should you read this?

If you already know what a regular expression is and what a finite state machine (or a finite automaton) is, then you probably also know that given a regular expression many times it is very difficult to directly construct the corresponding finite automaton. This is because there are two intermediate steps that you should go through before you can come up with the finite automaton. This note will explain to you what these two steps are. Most of the proofs given here are constructive in nature, i.e. they will help you to come up with an algorithm. All the proofs are to a large extent informal, and the aim has been to explain the basic underlying concepts. If you understand these, then you should be able to come up with more formal proofs on your own.

2 A word about notation

There is a slight deviation in notation from what has been presented to you during the lecture, and what you are going to read here. Be assured that both the notations mean the same thing.

First of all, we use the term *finite automaton* (FA). In many books this is also called a *finite state machine*. Both are exactly the same. In the script and in the lecture this was referred to as the *Endlicher Automat*.

We denote an FA by the 5-tuple (Q, E, q_0, δ, A), where Q is a set of states, E is an alphabet, q_0 is the starting state, δ is a transition function, and A is the set of accepting states.

In the script (and the slides presented in the lecture), this 5-tuple is denoted by (E, X, f, x_0, F). E corresponds to our E, X corresponds to our Q (set of states), f corresponds to our δ (transition function), x_0 corresponds to our q_0 (initial state), and lastly F corresponds to our A (set of accepting states).

3 Languages

To define what we mean by *language*, we first have to define what an *alphabet* is. An alphabet is a finite set of symbols which are used to form words in a language. An example of an alphabet might be a set like $\{a, b\}$. In this note we will denote an alphabet using the symbol E. A *string* over E is some number of elements of E (possibly none) placed in order. So if $E = \{a, b\}$ then strings over E can be $a, ab, bbaa, abab, aaaabbaab$ and so on and so forth. A very important string, which is always a string over E, no matter what E is, is the *null string* denoted by ε. This is the string with no symbols. If x is a string over E when we will use $|x|$ to denote the length of x. Hence $|aba| = 3$ and $|a| = 1$, and $|\varepsilon| = 0$. Note also that strings of length one over E are the same as elements of E.

For some alphabet E, we use E^* to denote the set of all possible strings over E. Applying * to some set is called the closure operation. So if $E = \{a, b\}$ then

$$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \ldots\}$$

A *language* over E will be a set of strings over E, so it will be some subset of E^*. So examples of languages might be:

$$\{\varepsilon, a, aa, bb\}$$

$$\{x \in \{a, b\}^* \mid |x| \leq 3\}$$

$$\{x \in \{a, b\}^* \mid |x| = 4\}$$

$$\{x \in \{a, b\}^* \mid x \text{ has equal number of } a\text{s and } b\text{s}\}$$

$$\{x \in \{a, b\}^* \mid x \text{ always ends with an } a\}$$

It is possible to give many more examples of languages, but hopefully the idea is clear to you by now.

Since languages are simply sets of strings, it is possible to generate new languages by applying standard operations on sets. So, for example, if L_1 and L_2 are languages over E then $L_1 \cup L_2, L_1 - L_2, L_1 \cap L_2$, etc. are also languages over E. We use L' to denote the complement of L. So $L' = E^* - L$.

In addition to the standard set operations, we can also use operations on strings like *concatenation* to generate new languages. If x and y are strings over E then the concatenation of x and y, denoted by xy is a new string formed by writing the symbols of x followed by the symbols of y. So if $x = aa$ and $y = bbb$ then $xy = aabbb$. If L_1 and L_2 are two languages then we can generate a new language L_1L_2 which is defined as follows.

$$L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

We can extend this notion of concatenation as follows. If L is a language over E then for any integer $k \geq 0$ we can define a language

$$L^k = LL \ldots L \ (k \text{ times } L)$$

So L^k is the set of strings obtained by concatenating k elements of L and $L^0 = \{\varepsilon\}$.

Hence from the above discussion you might have already started feeling that there are principally two ways of specifying languages. In the first way, we define how a language can be *constructed*. An example of this can be, if L_1 and L_2 are two languages, then we can construct a new language $L = L_1 \cup L_2$. In the second approach, we define some means by which the strings that belong to the language (which we are trying to specify) can be *recognized*. An example of this might be

$$L = \{x \in \{a,b\}^* \mid x \text{ always ends with an } a\}$$

Clearly, there is no hard line distinguishing the two approaches. As you read this note you will learn more ways of *generating* languages, which will be more sophisticated than the way the example language that we gave before as an example of *constructing* languages. You will also read more about recognizing languages, which is more or less what is note is all about.

4 Regular Expressions and the Corresponding Languages

Here we will see how new languages can be constructed from existing ones by using three simple operations–union (denoted by $+$), concatenation, and the closure operation (denoted by *). If we start with the simplest possible languages–those that consist of a single string which is either of length one, or is the null string–and then apply any combination of the three operators, then the resulting languages are called *regular languages*. Such languages can be described by explicit formulas called *regular expressions* and they consist of the three operations mentioned, i.e. union, concatenation, and the closure operation.

Example 1

	Language	*Regular Expression representing the language*
	$\{0\}$	0
	$\{0,1\}$ *i.e.* $\{0\} \cup \{1\}$	$0+1$
Example 1	$\{0,01\}$	$0+01$
	$\{0,\varepsilon\}\{001\}$	$(0+\varepsilon)001$
	$\{1\}^*\{10\}$	1^*10
	$\{10,11,1100\}^*$	$\{10+11+1100\}^*$

To explain our definition of regular languages a bit more clearly, we might say that they are languages containing only ε or a string of length one, together with those which can be obtained from such languages by a finite sequence of steps, where each step consists of applying one of the three operations to the languages that were obtained at an earlier step.

Example 2 1^*10 *is a* regular expression *representing the language consisting of all strings which consist of the substring* 10 *preceded by arbitrarily many* 1*'s.*

Applying our step by step procedure, we can obtain this by:

1. *Apply concatenation to* $\{1\}\ \{0\}$*, yielding* $\{10\}$
2. *Apply* * *to* $\{1\}$*, yielding* $\{1\}^*$
3. *Apply concatenation to* $\{1\}^*$ *and* $\{10\}$*, yielding* $\{1\}^*\{10\}$

Now we are ready to give formal definitions.

Definition 1 (Regular Expression) *A regular expression over the alphabet E is defined as follows:*

1. $\emptyset$ *is a regular expression corresponding to the empty language* $\emptyset$.
2. ε *is a regular expression corresponding to the language* $\{\varepsilon\}$.
3. *For each symbol $a \in E$, a is a regular expression corresponding to the language* $\{a\}$.
4. *For any regular expression r and s over E, corresponding to the languages L_r and L_s respectively, each of the following is a regular expression corresponding to the language indicated*
   (a) (rs) *corresponding to the language* L_rL_s
   (b) $(r+s)$ *corresponding to* $L_r \cup L_s$
   (c) r^* *corresponding to the language* ${L_r}^*$
5. *Only those "formulas" that can be produced by the application of rules 1-4 are regular expressions over E.*

Definition 2 (Regular Language) *A language over the alphabet E is a regular language if there is some regular expression over E corresponding to it.*

Example 3 *Let L be the language consisting of all strings of 0s and 1s that have even length. Note that since ε is of length zero, and zero is even, ε is in L.*

Note that L can be thought of as consisting of a number, possibly zero, of strings of length 2 concatenated. Hence the regular expression corresponding to L can be given as $(00+01+10+11)^*$.

Exercise 1 *What would be the regular expression corresponding to the language which consist of all strings of 0s and 1s that have odd length?*

Exercise 2 *Prove that every finite language is regular. (Hint: Use mathematical induction)*

5 Deterministic Finite Automata

In this section we will discuss about simple machines which will be used for *recognizing* the languages introduced in the last section. This means that given a language L, we will design a machine M_L, which on given any string s as input, will *accept* it if $s \in L$, and *reject* it otherwise.

We will see that regular languages can be characterized in terms of the "memory" required to recognize them. We will restrict ourselves to machines which will read any string presented to them in a single pass from left to right. This will help us to clarify what information needs to be "remembered" during the process of recognizing a language, and allows a classification of languages on the basis of how much needs to be remembered at each step in order to recognize the language. Regular languages are the simplest in this respect, there are other more "difficult" languages which we will not consider here.

Example 4 *Consider the language*

$$L = \{x \in \{0,1\}^* \mid x \textit{ ends in } 1 \textit{ and doesn't contain the substring } 00\}$$

Try to convince yourself that L is regular, and corresponds to the regular expression $(1+01)^*$.

Now let us try to design the machine for recognizing the language L in Example 4. This machine examines any input string one character at a time and at some stage decides to accept or reject the string. The easiest case, when the string can be disposed off, is when 00 occurs as a substring. Let us call this case N and for this we just need to remember if 00 occurs.

If 00 has not yet occurred while we are reading the string, there might be two other cases: case $L0$ when the last symbol read is 0, and case $L1$ when the last symbol read is 1. If we are in case $L1$ we might assume that the string is in L. In this case, if the next symbol read is a 0 then we are in case $L0$, and if the next symbol read is a 1 then we continue in case $L1$. In case $L0$, the symbols 0 and 1 would take us to case N and $L1$ respectively. The only other case that we have left out is the ε. This case should be treated separately because receiving 0 or 1 in this case requires different transition than what happens in other cases.

So we see that for recognizing L we don't need to remember exactly what substring we have read so far, but only remember which of the four states we are currently in.

The discussion above can be summarized in the form of the diagram shown in Figure 1.

The arrow to the circle labeled ε indicates where to start, when no symbols of the input string have been examined yet. The double circle indicates that if we are at this state when we reach the end of the string, then the answer is "Yes, the input string belongs to the language L". Ending at any other state indicates that the string is not in the language. Note that the four circles correspond to the four different cases described above.

You can think of the above diagram as a "machine" because it is possible to visualize a piece of hardware doing the job we described above. At any time the machine is in one of the four states/cases which we have labeled as ε, $L0$, $L1$ and N. Initially the machine is in state ε and as it reads one symbol at a time from the input string, it jumps from one state to the other. The double circle is an *accepting state* since it indicates that the substring read so far belongs to the language L. So if the machine is in this

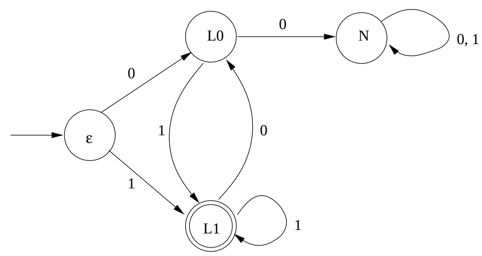

Figure 1: Recognizing the language in Example 4

state after the entire input string has been read, then this indicates that the string is in the language.

Observe that at the heart of the machine is a *set of states* and a *function* which receives a state and a symbol as input and outputs the *next state*. The crucial property of the machine is the finiteness of the set of states. So while recognizing a string, we need not remember the entire substring that has been read so far, but only which of the n ($n = 4$ in our case) states the substring belongs to. These set of states is precisely the "memory" aspect that we talked about at the beginning of this section.

Now we will give a formal definition of machines like the one we just described. We call such a machine a *(deterministic) finite automaton* or a *finite-state machine*.

Definition 3 (Deterministic Finite Automaton) *A finite automaton (FA) is a* 5*-tuple* (Q, E, q_0, δ, A) *where* Q *is a finite set of* states*,* E *is a finite set of* input symbols*,* $q_0 \in Q$ *is the* initial state*, and* δ *is a function from* $Q \times E$ *to* Q *(known as the* state transition function*), and lastly* $A \subseteq Q$ *is the set of* accepting states*.*

Note that it is possible to extend our definition where the transition function takes as an input not a single symbol, but a string of symbols. Let δ^* denote this new transition function. Then if for any string $y \in E^*$ and symbol $a \in E$, and state $q \in Q$, $\delta^*(q, ya) = \delta(\delta^*(q, y), a)$. The rest of the functionally of the FA remains the same.

Now let us state a theorem, whose proof we will work out later.

Theorem 1 *A language* $L \in E^*$ *is regular if and only if there is a FA that recognizes* L*.*

Using the above theorem, we can state another theorem describing an important property about regular languages.

Theorem 2 *If* L_1 *and* L_2 *are regular languages in* E^**, then* $L_1 \cup L_2, L_1 \cap L_2, L_1 - L_2$ *and* ${L_1}'$ *(complement of* L_1*), are all regular languages.*

We will now try to sketch the proof of the above theorem. Let L_1 and L_2 be recognized by the FAs $M_1 = (P, E, p_0, \delta_1, A_1)$ and $M_2 = (Q, E, q_0, \delta_2, A_2)$ respectively. Suppose that we wish to find an FA $M = (R, E, r_0, \delta, A)$ to recognize $L_1 \cup L_2$. If we can find M then by Theorem 1 we prove that $L_1 \cup L_2$ is regular.

Note that the FA M while processing a string needs to keep track of the status of both M_1 and M_2 simultaneously, and M accepts when either of M_1 or M_2 accepts. How can we do this? Basically, M_1 should remember the state it is in and M_2 should remember the state it is in, and M needs to remember the states of both M_1 and M_2. For this to happen, we construct the states of M to be pairs (p,q) where $p \in P$ and $q \in Q$. So $R = P \times Q$. Initially M_1 is in the state p_0 and M_2 is in the state q_0, and so M should be in the state (p_0, q_0). If M is in the state (p,q) and receives input a, then it should go to the state $(\delta_1(p,a), \delta_2(q,a))$, since $\delta_1(p,a)$ and $\delta_2(q,a)$ are the states to which M_1 and M_2 would respectively go. Since M has to accept a string whenever M_1 or M_2 does, the accepting states of M should be pairs (p,q) for which either $p \in A_1$ or $q \in A_2$. That leads to the fact that M accepts $L_1 \cup L_2$.

By now you probably came to understand that exactly the same strategy works for the cases $L_1 \cap L_2$ and $L_1 - L_2$, except for the definition of accepting states. In the first case (p,q) should be an accepting state if both $p \in A_1$ and $q \in A_2$, and in the second case (p,q) is an accepting state if $p \in A_1$ and $q \notin A_2$.

For the last case i.e. $L_1{}'$, note that we can reduce it to the third case by noting that $L_1{}' = E^* - L_1$. That solves this case. However we can solve it in an even more simple way by noting that the M which accepts $L_1{}'$ will be the same as M_1 accepting L_1 with the only difference that the accepting states of M will be $P - A_1$, i.e. exactly those states which are not the accepting states in M_1.

Exercise 3 *Let L_1 and L_2 be the following subsets of $\{0,1\}^*$:*

$$\begin{aligned} L_1 &= \{x \mid 00 \text{ \textit{is not a substring of} } x\} \\ L_2 &= \{x \mid x \text{ \textit{ends with} } 01\} \end{aligned}$$

Construct the FAs accepting L_1, L_2, $L_1 - L_2$, and $L_1 \cap L_2$.

6 From Regular Expression to an FA via NFA-ε and NFA

Given a regular expression, in most cases you will see that it can be broken into natural subparts. For each such subpart, it will be easy to construct an automaton, and then join these together to obtain a finite automaton corresponding to the given regular expression. However, you will see that it will be very natural to come up with an automaton for which there are multiple transitions from each state for a single input symbol. Moreover, there will also be states from which it will be possible to jump to other states with only the null string (ε) as input. These are what we will call *nondeterministic finite automaton with ε-transitions* (NFA-ε). If there are no ε-transitions in any state then the automaton is simply a *nondeterministic finite automaton*.

Given the NFA-ε obtained by joining together the NFA-εs corresponding to the subparts of the given regular expression, it is possible to first convert this to a nondeterministic finite automaton without the ε-transitions and then convert this to a deterministic finite automaton. How to do all this is the subject of the remaining sections.

7 Nondeterministic Finite Automata

Now we will consider machines similar to the ones described in Section 5, except that we will introduce some element of nondeterminism into them. These new machines will be capable of recognizing exactly the same languages as those recognized by deterministic finite automata (FA), but often these machines will be simpler to construct and will have fewer states than the corresponding FA.

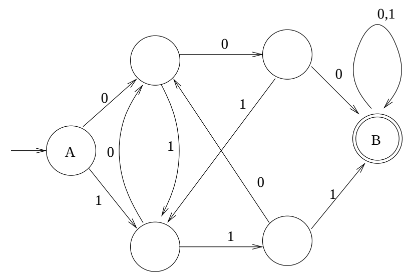

Figure 2: FA corresponding to the language in Example 5

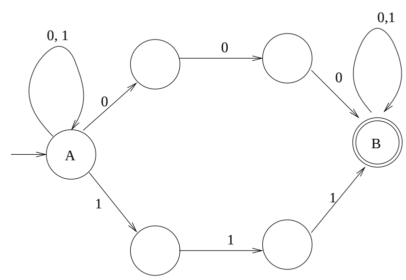

Figure 3: Nondeterministic finite automaton corresponding to the language in Example 5

The only difference these machines will have is that from each state there might be multiple transitions possible for any given input symbol, whereas in FAs for a given state and input symbol there was exactly one "next state".

Example 5 *Consider the language L corresponding to the regular expression:*

$$(0+1)^*(000+111)(0+1)^*$$

Try to convince yourself that the FA shown in Figure 2 corresponds to the language L.

The automaton shown in Figure 3 is similar to the one in Figure 2, but is not a FA since in state A, the input 0 offers possible transitions to two different states. It is unclear as to which path should be taken if the input 0 occurs in state A. However, if we decide on the rule that this automaton accepts strings for which there *exists some* path to an accepting state, then note that it reflects the structure in the regular expression much more than the FA in Figure 2 does. For any string in the language L it is easy to describe a path that leads to the accepting state B: start at A and continue looping back to A until the first occurrence of either 000 or 111, use these to go to the accepting state B, and continue looping back to B for each of the remaining symbols. Note that this is exactly what the regular expression in Example 5 also says. Further, also note that for any string not in L, there doesn't exist any path which starts in A and leads to B.

Now let us formally define machines like that described in Figure 3, which we will call nondeterministic finite automaton (NFA).

Definition 4 (Nondeterministic Finite Automaton) *A nondeterministic finite automaton (NFA) is a 5-tuple (Q, E, q_0, δ, A) where Q, E, q_0 and A are exactly the same as described in the case of a FA. The function is δ is different and is defined from $Q \times E$ to 2^Q (i.e. the set of all possible subsets of Q).*

The above definition simply says that an NFA is exactly the same as a FA (or a Deterministic FA) except that from each state there can be multiple different transitions for any given input symbol.

Note that it is also possible to define δ as a relation instead of a function, in which case $\delta \subseteq (Q \times E) \times Q)$. The two definitions are clearly equivalent.

As in the case of FA, it is also possible to extend the definition of δ to δ^* so that it accepts strings of symbols. So if for a string $y \in E^*$, symbol $a \in E$, and state $p \in Q$, $\delta^*(p, ya) = \delta(\delta^*(p, y), a)$.

Lastly, we can generalize our definition of an NFA even further by including ε transitions i.e. state transitions which require only the null string (ε) as input. We will see that this additional extension will simplify the process of finding an abstract machine for recognizing a given language. Therefore we can define an NFA with ε-transitions (denoted by NFA-ε) to be the same as an NFA, except that the transition function is defined as $\delta^* : Q \times (E^* \cup \{\varepsilon\}) \rightarrow 2^Q$.

8 Equivalence of FAs, NFAs, and NFA-εs

In an NFA-ε there are states from which it is possible to go to other states with only ε-transitions. Given any such state we need to compute all the states to which we can go from this state via ε-transitions. We will see shortly why this is important, but first we need to formalize this idea.

Definition 5 (ε-closure) *Let $M = (Q, E, q_0, \delta, A)$ be an NFA-ε. For a subset S of Q, the ε-closure of S is the subset $\varepsilon(S) \subseteq Q$ which can be defined as follows:*

1. *Every element of S is an element of $\varepsilon(S)$.*

2. *For any $q \in \varepsilon(S)$, every element of $\delta(q, \varepsilon)$ is an element of $\varepsilon(S)$.*

3. *No elements of Q are in $\varepsilon(S)$ unless they can be obtained from the above rules 1 and 2.*

So the ε-closure of a set S is simply the set of states that can be reached from the elements of S using only ε-transitions.

Algorithm to calculate $\varepsilon(S)$: Begin with $T = S$, and at each step add to T the union of all the sets $\delta(q, \varepsilon)$ for $q \in T$. Stop when the set T doesn't change any more. $\varepsilon(S)$ is the final value of T.

Now note that an FA can be considered to be a special case of an NFA, since a function from $Q \times E$ to Q can in an obvious way be identified with a function from $Q \times E$ to 2^Q, whose values are all sets with one element. Similarly, an NFA can be considered to be a special case of NFA-ε, one in which for each $q \in Q$, $\delta(q, \varepsilon) = \emptyset$. Therefore, any language that is recognized by an FA can be recognized by an NFA, and any language that is recognized by an NFA can be recognized by an NFA-ε.

The equivalence of FA, NFA, and NFA-ε in terms of the class of languages that they recognize, is proved with the additional fact that any language which is recognized by an NFA-ε can be recognized by an FA. This completes the loop, and proves that allowing nondeterminism doesn't enlarge the class of languages that FAs can recognize.

Theorem 3 *Let $L \subseteq E^*$, and suppose L is recognized by the NFA-ε $M = (Q, E, q_0, \delta, A)$. There is an FA $M_2 = (Q_2, E, q_2, \delta_2, A_2)$ recognizing L.*

We will not give a formal proof of the above theorem, but we will sketch what the proof would look like. We will do this in two parts. First we will find an NFA M_1 (without ε-transitions) recognizing L, and second we will find an FA equivalent to this NFA.

PART I. Finding an NFA $M_1 = (Q_1, E, q_1, \delta_1, A_1)$ recognizing L

(a) Defining M_1 (Eliminating the ε-transitions)
We let $Q_1 = Q$ and $q_1 = q_0$. The only thing that we have to do is define the transition function δ_1 such that, if in the machine M we can get from a state p to a state q using certain symbols together with ε-transitions, then in the machine M_1 we should be able to get from p to q using only those symbols, without the the ε-transitions. If in the machine M the initial state q_0 is not an accepting state, but it is possible to get from q_0 to an accepting state using only ε-transitions, then clearly the state q_0 in M_1 should also be labeled as an accepting state.

Hence, for any $q \in Q$ and $a \in E$, we define the transition function in M_1 as

$$\delta_1(q, a) = \delta^*(q, a)$$

Note that $\delta^*(q, a)$ is the set of all states that can be reached from q, using the input symbol a but allowing ε-transitions both before and after. Therefore, the way we have defined δ_1, if M can move from p to q using the input symbol a together with ε-transitions, then M_1 can move from p to q using the input a alone.

Finally, as we mentioned before, it might be necessary to make q_0 an accepting state in M_1. For this, we define

$$A_1 = A \cup \{q_0\} \text{ if } \varepsilon(\{q_0\}) \cap A \neq \emptyset \text{ in } M, \text{ and } A \text{ otherwise}$$

(b) For any $x \in E^*$ and $q \in Q$, if $|x| \geq 1$ then ${\delta_1}^*(q, x) = \delta^*(q, x)$
This can be proved using induction on $|x|$. Note that the basis step, with $|x| = 1$ follows from our definition of δ_1 above. So ${\delta_1}^*(q, x) = \delta_1(q, x)$, and we have already defined $\delta_1(q, x)$ to be $\delta^*(q, x)$. The rest of the proof is equally easy and we omit it here.

(c) M_1 recognizes L
Since M recognizes L, a string x is in L if and only if $\delta^*(q_0, x) \cap A \neq \emptyset$. On the other hand, x is accepted by M_1 if and only if ${\delta_1}^*(q_0, x) \cap A_1 \neq \emptyset$. First let us consider the case when $\varepsilon(\{q_0\}) \cap A = \emptyset$ in M. Here A_1 is defined to be A. If $|x| \geq 1$ then from part (b) above, $\delta^*(q_0, x) = {\delta_1}^*(q_0, x)$. So x is in L if and only if x is accepted by M_1. If $x = \varepsilon$ then x is not accepted by either M or M_1. Hence it follows that M_1 recognizes L.

In the second case, when $\varepsilon(\{q_0\}) \cap A \neq 0$, then $A_1 = A \cup \{q_0\}$. ε is accepted by both M and M_1. If $|x| \geq 1$, $\delta^*(q_0, x) = {\delta_1}^*(q_0, x)$. Either this set contains an element of A (in which case both M and M_1 accept x), or this set contains neither q_0 nor any elements of A (in which case both M and M_1 reject x). It is impossible for this set to contain

q_0 and no elements of A since if $q_0 \in \delta^*(q_0, x)$, then since $\delta^*(q_0, x)$ is the ε-closure of another set (by definition), $\delta^*(q_0, x)$ contains $\varepsilon(\{q_0\})$ and thus contains an element of A. Hence M and M_1 accept exactly the same strings.

PART II. For any NFA $M_1 = (Q_1, E, q_1, \delta_1, A_1)$ recognizing L, there is an FA $M_2 = (Q_2, E, q_2, \delta_2, A_2)$ recognizing L

(a) Definition of M_2

We are trying to eliminate the nondeterminism present in M_1, which means that in M_2 each combination of state and input symbol should result in exactly one state. Note that the transition function δ_1 takes a pair (q, a) and returns a set of elements of Q_1. Now suppose we define our notion of *state* to be a *set of elements* from Q_1. Let s be such a subset of Q_1. For any element $p \in s$, there is a set $\delta_1(p, a)$ of (possibly several) elements of Q_1 to which M_1 may go on input a. But for a single subset of elements s of Q_1 there is a single subset of elements of Q_1 in which M_1 may end up–this is the union of the sets $\delta_1(p, a)$ for all elements $p \in s$. So for each state-input pair (with our new notion of state), there is one and only one state. The machine obtained in this way clearly simulates in a natural way the action of the original machine M_1, provided the initial and the final states are defined correctly. Hence we have eliminated nondeterminism by what might be called as *subset construction*: states in Q_2 are subsets of Q_1.

From the above discussion, we can now define $M_2 = (Q_2, E, q_2, \delta_2, A_2)$ as follows.

$$\begin{aligned}
Q_2 &= \text{the set of all subsets of } Q_1 \\
q_2 &= \{q_1\} \\
\delta_2(q, a) &= \bigcup_{p \in q} \delta_1(p, a) \text{ for } q \in Q_2 \text{ and } a \in E \\
A_2 &= \{q \in Q_2 \mid q \cap A_1 \neq \emptyset\}
\end{aligned}$$

The definition of A_2 follows from the fact that for a string to be accepted in M_1, starting in q_1, the machine can end up only in sets of states which contain an element of A_1.

(b) M_2 recognizes L

Clearly this will follow if we can show that

$$\delta_2{}^*(q_2, x) = \delta_1{}^*(q_1, x) \ \text{(for every } x\text{)}$$

This can be proved using mathematical induction on $|x|$.

At this point, note that the above proof gives us two algorithms. The first one to find an NFA equivalent to a given NFA-ε, and the second one allows us to find an FA equivalent to a given NFA.

Example 6 *Consider the NFA-ε shown in Figure 4(a). Figure 4(b) shows the corresponding NFA and Figure 4(c) the corresponding FA.*

Let us consider a few steps in obtaining the NFA and the FA in the above example. Let $M = (Q, E, q, \delta, A)$ be the NFA-ε, $M_1 = (Q_1, E, q_1, \delta_1, A_1)$ be the NFA, and $M_2 = (Q_2, E, q_2, \delta_2, A_2)$ the FA.

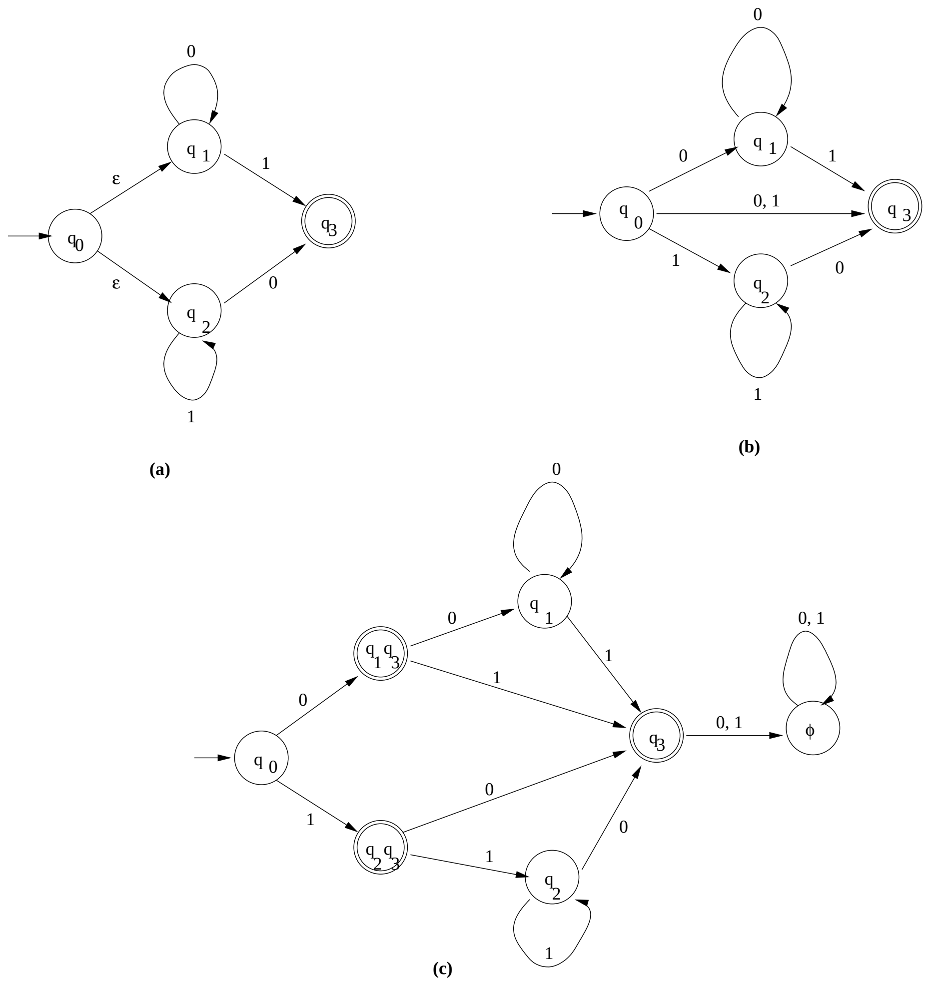

Figure 4: Obtaining an FA from a given NFA-ε. (a) denotes an NFA-ε, (b) the equivalent NFA, and (c) the equivalent FA

For finding $\delta_1(q_0,0)$, for example, we use

$$\delta_1(q_0,0) = \delta^*(q_0,0) = \varepsilon(\bigcup_{p\in\varepsilon(\{q_0\})} \delta(p,0))$$

The steps involved are: calculating $\varepsilon(\{q_0\})$, finding $\delta(p,0)$ for each p in this set, taking the union of the $\delta(p,0)$s, and calculating the ε-closure of the result. We can see that from the state q_0 with input 0, M can move to state q_1 (using a ε-transition to q_1 first), or to state q_3 (first to q_2 with ε , and then from there to q_3 with 0). There are no ε-transitions from q_1 or q_3, and so $\delta_1(q_0,0) = \{q_1,q_3\}$. A similar procedure will give the remaining values of δ_1. Finally since $\varepsilon(\{q_0\})$ does not contain q_3, $A_1 = A = \{q_3\}$. Figure 4(b) shows the NFA.

For the DFA shown in Figure 4(c), first note that it has only seven states and not sixteen, which is the number of subsets of $\{q_0,q_1,q_2,q_3\}$. This is because we create a state only when it is needed, during the construction process of the FA.

From $\{q_0\}$ the two states that can be reached with one symbol are $\{q_1,q_3\}$ and $\{q_2,q_3\}$. To compute $\delta_2(\{q_2,q_3\},0)$, we proceed as follows.

$$\delta_2(\{q_1,q_3\},0) = \delta_1(q_1,0) \cup \delta_1(q_3,0) = \{q_1\} \cup \emptyset = \{q_1\}$$

Since this state $\{q_1\}$ didn't appear yet during our construction, we add this state. Proceeding in this fashion we reach a point where for each state q that is drawn so far, $\delta_2(q,0)$ and $\delta_2(q,1)$ are both already drawn, and this indicates that we have all the reachable states.

Lastly, note that $\delta_2(\{q_3\},0) = \delta_2(\{q_3\},1) = \emptyset$. To complete the FA we add an extra state ϕ for handling this case. The resulting FA is the one shown in Figure 4(c).

9 There exists an FA for every Regular Expression

Here we want to show that if R is a regular expression over the alphabet E, and L is the language in E^* corresponding to R, then there is a finite automaton M recognizing L.

We will rely on the result of the last section where we proved the equivalence between an NFA-ε and an FA, and show that for a regular expression corresponding to a language L, we can construct an NFA-ε for recognizing L.

Recall the definition of a regular expression that was given by Definition 1. We will give an algorithm for constructing an NFA-ε. Towards this we will show how to construct an NFA-ε corresponding to the rules 4(a)–(c) in Definition 1. The NFA-εs corresponding to rules 1–3 in Definition 1 are really easy and are shown in Figure 5. Figure 5(a) shows the NFA-ε corresponding to the empty language $\emptyset$, (b) corresponding to the regular expression ε (which results in the language $\{\varepsilon\}$), and (c) corresponding to the regular expression a (which results in the language $\{a\}$).

Now we will show how to construct an NFA-ε corresponding to 4(a)–(c) in Definition 1. Note that this will give us all the essential ingredients for a proof of the fact that for every regular expression there is an NFA-ε (and hence an FA). The proof will use mathematical induction. The basis step was just shown above–that there is an NFA-ε corresponding to the regular expressions $\emptyset$, ε, and $a \in E$. The induction step will show that if r and s are regular expressions for which there exist NFA-εs then for the regular expressions obtained by applying rules 4(a)–(c) of Definition 1 we can also obtain NFA-εs.

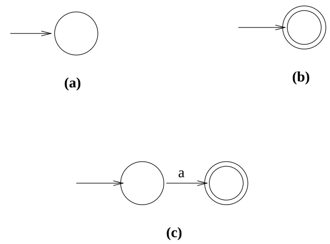

Figure 5: NFA-εs corresponding to (a) $\emptyset$, (b) ε, and (c) $a \in E$

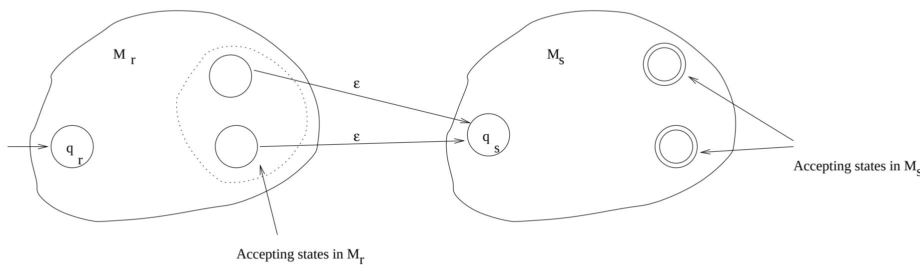

Figure 6: NFA-ε construction corresponding to *concatenation*

Case 1. Suppose $R = (rs)$. Let L_r and L_s be the languages corresponding to r and s respectively and let $L = L_r L_s$ correspond to R. Suppose the NFA-εs $M_r = (Q_r, E, q_r, \delta_r, A_r)$ and $M_s = (Q_s, E, q_s, \delta_s, A_s)$ recognize L_r and L_s. We assume that $Q_r \cap Q_s = \emptyset$ (by renaming states if necessary). We want to construct $M = (Q, E, q_0, \delta, A)$ which will recognize $L = L_r L_s$. For this we use a very simple idea: we make the initial state of M i.e. $q_0 = q_r$, the accepting states of M i.e. $A = A_s$, and we add ε transitions from every element of A_r to q_s. Strings in the language $L_r L_s$ will correspond to paths from q_r to an element in A_r, then jump to q_s using the ε-transition, and finally finish in some state in A_s. Figure 6 shows this construction.

Try to convince yourself that if $x \in L_r L_s$ then x is accepted by M, and for all strings x accepted by M, $x = x_r x_s$ where $x_r \in L_r$ and $x_s \in L_s$.

Case 2. Suppose $R = (r + s)$. L_r and L_s are recognized by $M_r = (Q_r, E, q_r, \delta_r, A_r)$ and $M_s = (Q_s, E, q_s, \delta_s, A_s)$, where as before $Q_r \cap Q_s = \emptyset$. Let $L = L_r \cup L_s$ and let $M = (Q, E, q_0, \delta, A)$ recognize L. The way we will construct M is, we take M_r and M_s, add a new initial state q_0, and add ε-transitions from q_0 to the initial states of M_r and M_s (i.e. q_r and q_s). The construction is shown in Figure 7

If $x \in L_r$ then M with x as input moves from q_0 to q_r by a ε-transition, and then from q_r to an element of A_r (and therefore to an element of A, since $A = A_r \cup A_s$). Hence $\varepsilon x = x$ is accepted by M. An exactly similar argument works for $x \in L_s$. Clearly, the converse is also true, i.e. if x is accepted by M then x either belongs to L_r or to L_s, and hen $x \in L_r \cup L_s$.

Case 3. Suppose $R = (r^*)$. Let $M_r = (Q_r, E, q_r, \delta_r, A_r)$ recognize L_r, and $L = L_r{}^*$. We want to construct $M = (Q, E, q_0, \delta, A)$ for recognizing L. This we do as follows. We

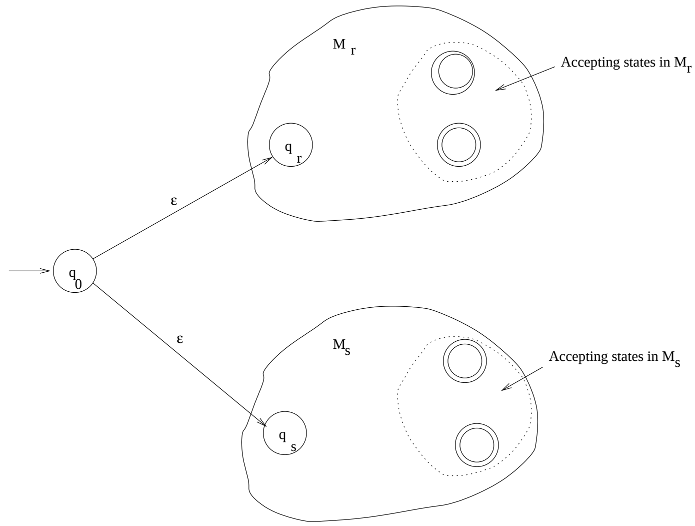

Figure 7: NFA-ε construction corresponding to *union*

assume that q_0 is a state not in Q_r.

$$\begin{array}{rcl} Q & = & Q_r \cup \{q_0\} \\ A & = & \{q_0\} \\ \text{for } \; q \in Q & \text{and} & a \in E \\ \delta(q,a) & = & \begin{cases} \delta_r(q,a) & \text{if} \quad q \in Q \\ \emptyset & \text{if} \quad q = q_0 \end{cases} \end{array}$$

for $q \in Q$

$$\delta(q,\varepsilon) \quad = \quad \begin{cases} \delta_r(q,\varepsilon)) & \text{if} \quad q \in Q_r - A_r \\ \delta_r(q,\varepsilon) \cup \{q_0\} & \text{if} \quad q \in A_r \\ \{q_r\} & \text{if} \quad q = q_0 \end{cases}$$

Now suppose $x \in L_r{}^*$. If $x = \varepsilon$ then clearly x is accepted by M. For some $m \geq 1$, let $x = x_1 x_2 \ldots x_m$, where each $x_i \in L_r$. M moves from q_0 to q_r by a ε-transition. Then for each i, M moves from q_r to an element of A_r by a sequence of transitions corresponding to those that happen in M_r for x_i, and finally M moves back to q_0 by a ε-transition. It follows that $\varepsilon x_1 \varepsilon x_2 \varepsilon \ldots \varepsilon x_m \varepsilon = x$ is accepted by M. The construction of M and this process of accepting x is illustrated in Figure 8

Clearly, for every x accepted by M there is a sequence of transitions begining and ending at q_0, and it follows that x can be decomposed to the form $x_1 x_2 \ldots x_m$ where each $x_i \in L_r$.

10 There exists a Regular Expression for every FA

Given an FA $M = (Q, E, q_0, \delta, A)$ recognizing a language L, we want to show that there exists a regular expression over E corresponding to L.

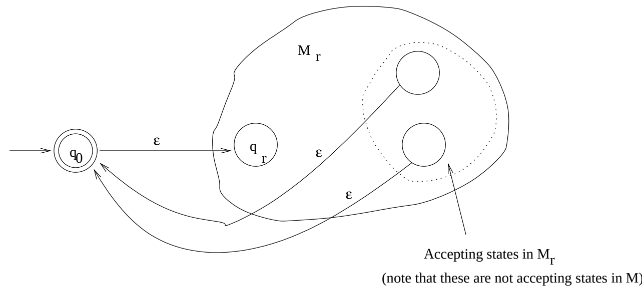

Figure 8: NFA-ε construction corresponding to *closure*

For elements p and q belonging to Q let

$$L(p,q) = \{x \in E^* \mid \delta^*(p,x) = q\}$$

$L(p,q)$ is the set of strings that allow M to reach state q if it begins in state p. If we can show that each language $L(p,q)$ correspond to a regular expression, then since the language recognized by M is a union of the languages $L(q_0,q)$ for all $q \in A$, a regular expression for L can be obtained by combining these individual regular expressions using $+$. We will give an inductive proof that each language $L(p,q)$ is regular.

On what shall we base our induction? For this, consider the elements of Q to be labeled with integers from 1 to N. Next we formalize the idea of a path *going through a state s*. If $x \in E^*$, we say x represents a path from p to q through s if x can be written in the form $x = yz$ for some y and z with $|y|,|z| > 0$, $\delta^*(p,y) = s$ and $\delta^*(s,z) = q$. For any $J \geq 0$ we define the set $L(p,q,J)$ as follows. $L(p,q,J) = \{x \in E^* \mid x$ corresponds to a path from p to q that goes through no state numbered higher than $J\}$. Note that $L(p,q,N) = L(p,q)$ since N is the highest numbered state in the FA. Thus it will be sufficient to show that $L(p,q,N)$ is regular, and the way we shall show this is by proving that $L(p,q,J)$ is regular for each J, $0 \leq J \leq N$. We will use mathematical induction over J.

For the basis step we have to show that $L(p,q,0)$ is regular. Now a path from p to q can go from p to q without going to any state numbered higher than 0 (i.e. without going through *any* other state) only if it corresponds to a single symbol, or if $p = q$ and the path corresponds to the string ε. Thus $L(p,q,0)$ is a subset of $E \cup \{\varepsilon\}$ and is regular.

Now we want to show that if the language $L(p,q,K)$ is regular then $L(p,q,K+1)$ is also regular for $1 \leq p,q \leq N$, and $0 \leq K \leq N-1$. A string $x \in L(p,q,K+1)$ represents a path from p to q that goes through no state numbered higher than $K+1$. There are two possible ways in which this can happen: the path could bypass the state $K+1$ altogether, in which case $x \in L(p,q,K)$ and is regular, or the path could go from p to $K+1$, possibly looping back to $K+1$ several times and then go from $K+1$ to q, never going to any states higher than $K+1$. In the later case, we can write x in the form x_1yx_2, where

$$\begin{aligned}
\delta^*(p,x_1) &= K+1 \\
\delta^*(K+1,y) &= K+1 \\
\delta^*(K+1,x_2) &= q
\end{aligned}$$

If we include all the looping from $K+1$ back to itself within the string y, then $x_1 \in L(p,K+1,K)$ and $x_2 \in L(K+1,q,K)$. This simply says that before arriving at $K+1$ for the first time and after leaving $K+1$ for the last time, the path goes through no state numbered higher than K. Further, if $y \neq \varepsilon$, and each separate loop in the path is represented by a string y_i, then $y = y_1 y_2 \ldots y_l$ and each y_i is an element of $L(K+1,K+1,K)$. Therefore $y \in L(K+1,K+1,K)^*$. Hence it follows from the above discussion that

$$L(p,q,K+1) = L(p,q,K) \cup L(p,K+1,K)L(K+1,K+1,K)^* L(K+1,q,K)$$

This proves that $L(p,q,K+1)$ is regular.

The results in this and the last two sections together show that a language is regular if and only if it is accepted by a finite automaton. This result is known as Kleene's theorem.

11 References

1. "Introduction to Automata Theory, Languages, and Computation", by John Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, Addison-Wesley, Reading, Massachusetts, USA, 2001.

2. "Elements of the Theory of Computation", by Harry R. Lewis and Christos H. Papadimitriou, Prentice-Hall International, 1981.

3. "Introduction to Languages and the Theory of Computation", by John C. Martin, McGraw-Hill Inc., 1991.