Chapter 17

Finite Automata, Regular Languages and Type 3 Grammars

17.1 Finite automata

A finite automaton (fa), or finite state automaton (fsa), is an abstract computing device that receives a string of symbols as input, reads this string one symbol at a time from left to right, and after reading the last symbol halts and signifies either acceptance or rejection of the input. At any point in its computation a fa is in one of a finite number of *states*. The computations of a fa are directed by a "program," which is a finite set of instructions for changing from state to state as the automaton reads input symbols. A computation always begins in a designated state, the *initial state*. There is also a specified set of *final states*; if the fa ends up in one of these after reading the input, it is accepted; otherwise, it is rejected.

It may help to visualize a finite automaton as composed of (1) a control box, which at any point in the computation can be in one of the allowed internal states, and (2) a reading head, which scans a single symbol of the input. In Fig. 17-1 we have represented a fa in its initial state, q_0, at the beginning of its computation of the input string *abaab*.

Let us suppose further that the set of states for this fa is $\{q_0, q_1\}$ and that the set of final states is $\{q_1\}$ (We could have made the initial state a final state also, but we have not chosen to do so here.) We specify the program

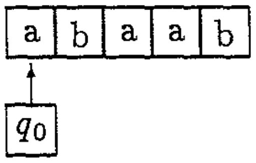


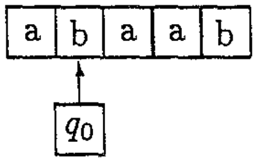
Figure 17–1.

for this fa as a set of triples of the form (q_i, x, q_j), where q_i and q_j are states and x is a symbol of the alphabet—here, $\{a, b\}$. Instructions are interpreted in the following way: when the fa is in state q_i reading a symbol x on the input tape, it changes to state q_j (possibly identical to q_i) and advances the reading head one symbol to the right. The instruction for the now current state and symbol is then carried out, and the process is repeated until there are no more symbols to be read. Here are the instructions for our example:

(17–1) (q_0, a, q_0)
(q_0, b, q_1)
(q_1, a, q_1)
(q_1, b, q_0)

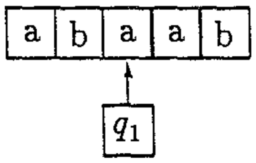
Thus, from the initial situation shown in Fig. 17-1 the fa would first execute the instruction (q_0, a, q_0) and find itself in the following situation:

(17–2)



Now the instruction (q_0, b, q_1) is applied to produce:

(17–3)



and so on. You should now be able to verify that after reading the final symbol b, the fa is in state q_0, and since this is not a final state, the input is rejected. It should also be easy to determine that the input *ab* would be accepted, while *aa* is rejected.

Problem: Describe the set of all tapes accepted by this fa

What would happen if the fa of our example were given the empty string as input? In such a case, the input tape has no symbols on it, and so no instructions can be applied—there being nothing to read. Thus, the initial situation is identical to the final situation, and since the initial state, q_0, is not a final state, this input is rejected. Note that to say that a machine accepts the empty string as part of its language is far different from saying that the language accepted is empty. The latter means that it accepts neither the empty string nor any other string An automaton with no final states would, for example, accept no strings and thus would be said to accept the empty language.

One might wonder how a finite automaton would behave if there were no instruction applicable at a particular point or if there were more than one instruction which could be applied. Such questions will arise with the so-called *non-deterministic* automata, which we consider below For now, we will be concerned only with *deterministic* fa's, in which there is one and only one instruction for each combination of state and scanned symbol. As the name suggests, the behavior of such an automaton is completely determined, given the input tape and the initial state.

17.1.1 State diagrams of finite automata

A convenient representation for a fa, called a *state diagram*, can be constructed in the following way. Each state is represented by a circle labelled with the name of the state For each instruction (q_i, x, q_j) an arrow is drawn from the q_i circle to the q_j circle and labelled with symbol x. Final states are enclosed by an additional circle, and the initial state is marked by a caret The state diagram for our example fa is shown in Fig. 17-2.

With such a diagram it is easy to trace the steps of a computation like that for *abaab* in the example above The fa starts in state q_0 and returns to q_0 reading an a. The b takes it to state q_1; the next two a's leave it in q_1; and the final b returns it to q_0. Since q_0 is non-final, the string is not accepted.

It is also somewhat easier to see from the state diagram than from the list of instructions in (17-1) that this fa accepts exactly the strings over the alphabet $\{a, b\}$ containing an odd number of b's

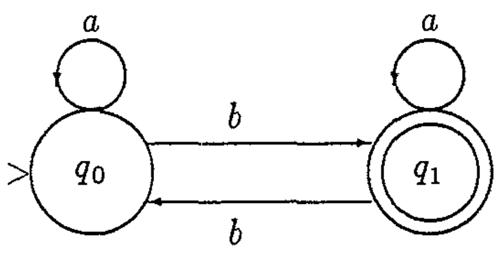


Figure 17-2.

17.1.2 Formal definition of deterministic finite automata

DEFINITION 17 1 *A* deterministic finite automaton (dfa) M *is a quintuple* $\langle K, \Sigma, \delta, q_0, F\rangle$, *where*

> K *is a finite set, the set of* states
>
> Σ *is a finite set, the* alphabet
>
> $q_0 \in K$, *the* initial state
>
> $F \subseteq K$, *the set of* final states
>
> δ *is a function from* $K \times \Sigma$ *into* K, *the* transition function *(or* next-state function*)*.

■

Note that the property of determinism is expressed in this definition by the fact that δ is a function; that is, for all $q \in K$ and all $\sigma \in \Sigma$, $\delta(q, \sigma)$ has a unique value.

In order to express formally what it is for a dfa to accept an input string, we introduce the notion of a *situation* of a dfa. This is intended to be essentially a "snapshot" of the dfa and its input tape at any point during a computation. We represented situations above by diagrams such as Fig. 17-1, but we now want a more compact notation. The essential information to be captured is (1) the current state of the automaton; (2) the input tape; and (3) the position of the reading head. A convenient representation of this information is in the form of a triple (x, q, y), where q is the current state and x and y are the portions of the input string to the left and right of the reading head, respectively In this notation, the symbol being scanned is the leftmost symbol, if any, of y. Thus, the diagram in (17-3) would be represented as (ab, q_1, aab), and the sequence of situations in the computation of $abaab$ would be as follows:

(17–4) $(e, q_0, abaab) \vdash (a, q_0, baab) \vdash (ab, q_1, aab) \vdash (aba, q_1, ab) \vdash (abaa, q_1, b) \vdash (abaab, q_0, e)$

Here we have used the symbol $\vdash$ (the 'turnstile') to indicate that one situation leads to another by a single move of the automaton. We will define this formally below. Note that at the beginning of the computation the string to the left of the reading head is empty, and likewise for the string to the right at the end of the computation. In formal terms, a situation of a dfa is defined as follows:

DEFINITION 17.2 *Given a dfa $M = \langle K, \Sigma, \delta, q_0, F\rangle$, a* situation *of M is a triple (x, q, y), where $q \in K$ and $x, y \in \Sigma^*$.* ■

This definition allows situations for a given M which are not actually attainable in the course of any computation by M. For example, (aa, q_1, abb) would be a situation of our example dfa, but it is not a situation which can be reached from the initial situation $(e, q_0, aaabb)$ by the given transition function of the automaton. It is convenient, nonetheless, to define the notion of situation in this overly broad way and to focus on attainable situations in our definition of acceptance.

Let us define, for a given dfa M, a binary relation on situations which we will call *produces-in-one-move.* Situation A produces situation B in one move just in case by applying one instruction in δ to A we produce situation B. Any two adjacent situations in (17-4) would stand in this relation, for example. Formally, we have:

DEFINITION 17.3 *Given a dfa $M = \langle K, \Sigma, \delta, q_0, F\rangle$, a situation (x, q, y)* produces *situation (x', q', y')* in one move *iff (1) there is a $\sigma \in \Sigma$ such that $y = \sigma y'$ and $x' = x\sigma$ (i.e., the reading head moves right by one symbol), and (2) $\delta(q, \sigma) = q'$ (i.e., the appropriate state change occurs on reading σ).* ■

Problem: In general, is the produces-in-one-move relation reflexive? symmetric? transitive?

As noted above, we indicate by the turnstile that two situations stand in this relation; thus $(x, q, y) \vdash (x', q', y')$

Once again, this definition is permissive in that it allows pairs of situations to stand in the produces-in-one-move relation whether or not either is attainable from some initial situation in the course of a computation For example, in the dfa above, $(aa, q_1, abb) \vdash (aaa, q_1, bb)$, despite the fact that neither situation could arise from $(e, q_0, aaabb)$.

As a final step before giving a formal definition of acceptance, we extend the previously defined relation to a new binary relation: "produces in zero or more steps." We say that a situation A produces situation B in zero or more steps (or simply *A produces B*) iff there is a sequence of situations $S_0 \vdash S_1 \vdash \ldots \vdash S_k$ such that $A = S_0$ and $B = S_k$ $(k \geq 0)$ (If $k = 0$, there is only one situation in the sequence, and $A = B$; thus, every situation produces itself in zero or more moves) This relation is reflexive (as we have just seen) and transitive; in fact, it is in formal terms the *reflexive, transitive closure* of the produces-in-one-move relation. This is just the produces-in-one-move relation (for a given dfa) with enough pairs added to it to make it a reflexive and transitive relation. We will denote this relation by $\overset{*}{\vdash}$ ('turnstile star') We may also add a subscript M to this or to the turnstile if necessary to emphasize the fact that the relation is defined with respect to a particular automaton M; thus, $\vdash_M$ or $\overset{*}{\vdash}_M$. Referring again to 17-4, we see

(17–5) $(a, q_0, baab) \overset{*}{\vdash} (aba, q_1, ab)$ and $(ab, q_1, aab) \overset{*}{\vdash} (ab, q_1, aab)$

but neither of these would be true if $\overset{*}{\vdash}$ were replaced by $\vdash$. Acceptance of a string by dfa is now easy to define formally:

DEFINITION 17 4 *Given a dfa $M = \langle K, \Sigma, \delta, q_0, F\rangle$ and a string $x \in \Sigma^*$, M* accepts *x iff there is a $q \in F$ such that $(e, q_0, x) \overset{*}{\vdash}_M (x, q, e)$.* ■

And finally:

DEFINITION 17.5 *Given a dfa $M = \langle K, \Sigma, \delta, q_0, F\rangle$, the* language accepted *by M, denoted $L(M)$, is the set of all strings accepted by M.* ■

17.1.3 Non-deterministic finite automata

We now consider what happens if we relax the requirement that the next move of a fa always be uniquely determined. Departures from determinism can occur in two ways:

(i) for a given state symbol pair, there may be more than one next state
(ii) for a given state symbol pair, there may no next state at all

There is an additional generalization from the deterministic case which, while it is strictly speaking not a departure from determinism, is often included in the definition of non-deterministic fa's:

(iii) transitions of the form (q_i, w, q_j) are allowed, where $w \in \Sigma^*$; i.e., the fa can read a string of symbols in one move; and in particular,
(iv) transitions of the form (q_i, e, q_j) are allowed; i.e., the fa can change state without moving the reading head.

An example of a non-deterministic fa which illustrates all four of these conditions is shown in Fig 17-3.

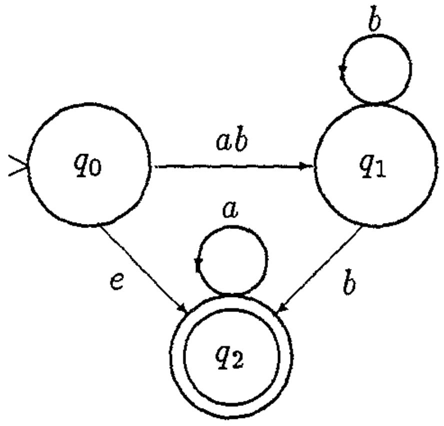


Figure 17–3.

The behavior of a non-deterministic fa is defined as follows: an input tape is accepted iff there is *some* path through the state diagram which begins in the initial state, reads the entire input, and ends in a final state. In the fa of Fig. 17-3, for example, *abb* is accepted by virtue of the path from q_0 (reading *ab*) to q_1 and then to q_2 (reading *b*). The fact that there is also a path reading *abb* which ends in q_1 is irrelevant; only the existence of at least one accepting path is required. On the other hand, *ba* is not accepted since there is no path through the state diagram which succeeds in reading the entire string. Likewise, *aba* is not accepted. Note, however, that the string *a* is accepted by the path leading from q_0 to q_2 (reading no input) and then again to q_2 (reading an *a*).

17.1.4 Formal definition of non-deterministic finite automata

Formally, a non-deterministic fa is identical to a dfa except that the transition function becomes a relation

DEFINITION 17.6 *A* non-deterministic finite automaton (nfa) M *is a quintuple* $\langle K, \Sigma, \Delta, q_0, F\rangle$, *where* K, Σ, q_0, *and* F, *are as for a dfa, and* Δ, *the* transition relation, *is a finite subset of* $K \times \Sigma^* \times K$ *(i.e., a relation from* $K \times \Sigma^*$ *into* K*).* ■

The fact that Δ is a relation, but not necessarily a function, allows for conditions (i) and (ii) above. The fact that it is a relation from $K \times \Sigma^*$ rather than from $K \times \Sigma$ allows for condition (iii) and its special case (iv). Because of the infiniteness of Σ^*, a relation from $K \times \Sigma^*$ to K is itself potentially infinite; we stipulate that it must be a *finite* subset of $K \times \Sigma^* \times K$ in order to retain the notion of a finite machine; i.e., an automaton with a finite number of states and a finite number of instructions in its program. Note that by this definition dfa's are a proper subclass of nfa's.

The definitions of situation, produces-in-one-move, etc. are similar to those for dfa's.

DEFINITION 17.7 *A* situation *is any triple in* $\Sigma^* \times K \times \Sigma^*$. $(x, q, y) \vdash (x', q', y')$ *is true iff there exists* a string $z \in \Sigma^*$ *such that* $x' = xz$, $y = zy'$, *and* $(q, z, q') \in \Delta$. ■

DEFINITION 17.8 Produces, *i.e.,* $\overset{*}{\vdash}$, *is the reflexive, transitive closure of the* $\vdash$ *relation, and, as before, an nfa* M accepts *a string* $x \in \Sigma^*$ *iff* $(e, q_0, x) \overset{*}{\vdash}_M (x, q, e)$ *for some* $q \in F$. ■

The language accepted is, of course, the set of all strings accepted.

17.1.5 Equivalence of deterministic and non-deterministic finite automata

One might expect that when fa's are allowed the extra degrees of freedom inherent in non-determinism, significantly more powerful devices would be the result. Surprisingly, this is not the case. nfa's accept exactly the same class of languages as dfa's; or in other words, for every nfa there is an

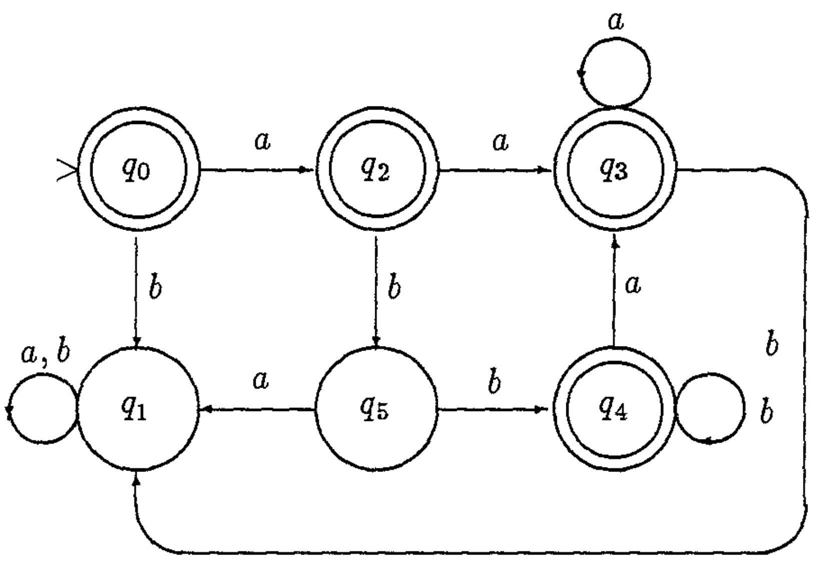


Figure 17–4.

equivalent dfa—equivalent in the sense that both accept exactly the same set of strings. (The equivalence in the other direction is trivial, every dfa being *a fortiori* a nfa.) An example of a dfa which is equivalent to the nfa in Fig. 17-3 is shown in Fig. 17-4.

A dfa will typically have more states that an nfa to which it is equivalent. The dfa works by essentially keeping track, in its states, of the *set* of states that the nfa could be in if it followed all possible paths simultaneously on a given input. There is in fact an algorithm for converting any nfa into an equivalent dfa, but it is too long to be included here. It can be found in Chomsky and Miller (1958), Rabin and Scott (1959), Hopcroft and Ullman (1979), and Lewis and Papadimitriou (1981).

In view of the equivalence of dfa's and nfa's, one might wonder why we bother to consider nfa's at all. For one thing, nfa's are generally easier to construct than dfa's Thus, it might be simpler to show that a given language is of the sort accepted by a fa by devising an nfa which accepts it. We will in fact make use of this convenience in proving certain theorems about fa's below. For another thing, determinism and non- determinism are notions which arise in connection with other classes of automata to be considered later, and, as we will see, the two varieties are not always equivalent in these cases.

17.2 Regular languages

We will say that a language is a *finite automaton language* (fal) just in case there is some fa which accepts it. We know, for example, that $\{x \in \{a,b\}^* \mid x \text{ contains an odd number of } b's\}$ is such a language by virtue of the fact that we exhibited an fa accepting this language in Fig 17-2 above. Consider, however, the general problem of deciding whether a given language L is a fal or not. Suppose we try to construct a fa accepting L, but all our attempts result in failure. We would not be justified in concluding that L is *not* a fal, of couse, since we might succeed in our attempts with renewed persistence or perhaps a bit of luck. It would be useful if we had another way to characterize this class of languages which does not depend on our ingenuity, or lack of it, in constructing fa's. To that end, we define a class of languages, called *regular languages*, which turn out to be provably identical to the class of fal's. Since these languages are defined (recursively) by reference to operations on sets of strings rather than to acceptance by automata, we have another interesting and potentially useful approach to these languages. In the next section, we prove a theorem which is used primarily to show that a given language is *not* an fal We will need a preliminary definition:

DEFINITION 17.9 *Given two sets of strings,* A *and* B, *the* concatenation *(*or set product*) of* A *and* B, *denoted* $A \cdot B$ *(*or *just* AB*), is the set of strings* $\{x^\frown y \mid x \in A \text{ and } y \in B\}$. ■

For example,

(17–6) if $A = \{a,b\}$ and $B = \{cc,d\}$, then $AB = \{acc, ad, bcc, bd\}$

Note that the concatenation of two sets of strings is itself a set of strings, in contrast to the Cartesian product of two sets (of anything), which is a set of ordered pairs. We should also note that, according to the definition, if one of the sets is empty, the concatenation is also empty.

Recall also that the notation A^* is used to denote the set of all strings formed over the alphabet A. This is a special case of an operation called *closure* or *Kleene star* on a set of strings: given a set of strings A, the Kleene star or closure of A, denoted A^*, is the set formed by concatenating members of A together any number of times (including zero) in any order and allowing repetitions For example, (17–7) $\{a, bb\}^* = \{e,\ a,\ bb,\ aa,\ abb,\ bba,\ bbbb,\ aaa,\ aabb,\ abba, \ldots\}$.

Note that our original notation treated the members of the alphabet as strings of length 1. We are now ready to give the definition of the *regular languages*.

DEFINITION 17.10 *Given an alphabet* Σ:

1. $\emptyset$ *is a regular language.*
2. *For any string* $x \in \Sigma^*$, $\{x\}$ *is a regular language.*
3. *If* A *and* B *are regular languages, so is* $A \cup B$.
4. *If* A *and* B *are regular languages, so is* AB.
5. *If* A *is a regular language, so is* A^*.
6. *Nothing else is a regular language unless its being so follows from 1–5.*

■

For example,

(17–8) Let $\Sigma = \{a, b, c\}$. Then since aab and cc are members of Σ^*, $\{aab\}$ and $\{cc\}$ are regular languages. So is the union of these two sets, *viz.*, $\{aab, cc\}$, and so is the concatenation of the two, *viz.*, $\{aabcc\}$. Likewise, $\{aab\}^*$, $\{cc\}^*$, and $\{aab, cc\}^*$, are all regular languages, etc.

Another way to state the definition is to say that the regular languages (over a given alphabet) are just those which can be obtained from the empty language and the 'unit' languages (those containing just one string) by repeated application of the operations of union, concatenation, and Kleene star. Thus, to show that a given language is in fact regular, we indicate how it can be built up out of empty or unit languages by these operations. For example, the language $\{x \in \{a, b\}^* \mid x \text{ contains an odd number of } b\text{'s}\}$ is a regular language since an equivalent representation of this language is $\{a\}^* \cdot \{b\} \cdot \{a\}^* \cdot (\{b\} \cdot \{a\}^* \cdot \{b\} \cdot \{a\}^*)^*$. In writing such expressions, it is usual to render them less cumbersome by suppressing the braces around sets and the dots in concatenation; extra parentheses can also be dispensed with in view of the associativity of union and concatenation. The previous expression in this pared-down notation would be: $a^*ba^*(ba^*ba^*)^*$ Such expressions are called *regular expressions* We note also that the set of all strings in $\{a,b\}^*$ which contain exactly two or three b's is a regular language since it can be represented (as a regular expression) as $a^*ba^*ba^* \cup a^*ba^*ba^*ba^*$, or equivalently as $a^*ba^*ba^*(e \cup ba^*)$. Note, finally, that $\{e\}$ is a regular language since it is equal to $\emptyset^*$.

Having thus characterized the regular languages, we want to show that they are in fact identical to the finite automaton languages

THEOREM 17.1 *(Kleene) A set of strings is a finite automaton language if and only if it is a regular language.* ■

We will sketch the proof of one half of this theorem; i.e , that every fal is a regular language. The proof of the converse is too complex to give here, but can be found in works such as Hopcroft and Ullman (1979) and Lewis and Papadimitriou (1981).

First, we show that the empty language and the unit languages (for a given Σ) are fal's. The empty language is accepted by the one-state fa in Fig. 17-5(a), and for each x in Σ^*, a fa of the form shown in Fig. 17-5(b) accepts the language $\{x\}$. (Note that these fa's are in general non-deterministic.)

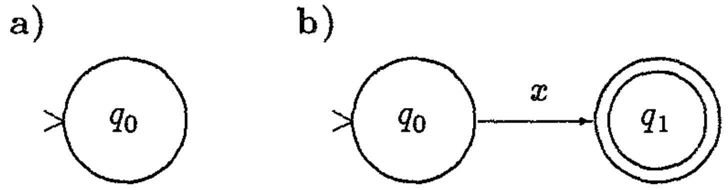


Figure 17–5.

Next we show that the fal's are closed under the operations of union, concatenation, and Kleene star. From this it will follow that the fal's are included in the regular languages.

We will indicate how, given any two fa's accepting languages L_1 and L_2, we can construct fa's accepting, respectively, $L_1 \cup L_2$, L_1L_2, and L_1^*.

Suppose, for example, we are given the following fal's:

$L_1 = ab^*a$; that is, all strings beginning and ending with an a with any number of b's between them.

L_2 = all strings in $\{a, b\}^*$ containing exactly two b's.

These are fal's since they are accepted by the fa's in Figs. 17-6 and 17-7, respectively.

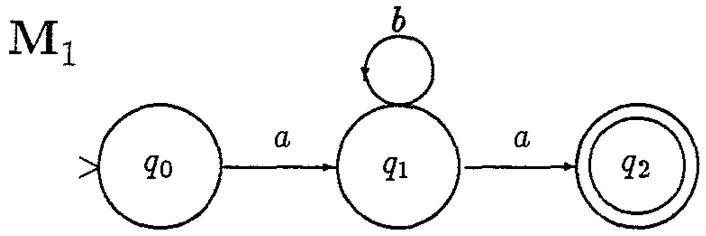


Figure 17-6.

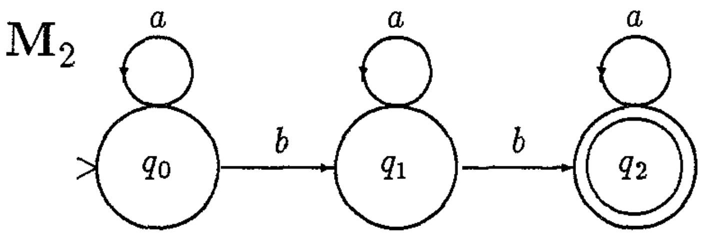


Figure 17-7.

To form a fa accepting the union of L_1 and L_2 we first relabel the states of one of the fa's so that all have distinct names—let us suppose that we add primes to the states of M_2. Now we introduce a new start state, q_0'', and establish e-transitions, i.e., changes of state reading the empty string, from q_0'' to the old start states, q_0 and q_0'. Everything else, including the final states, remains the same. The resulting automaton M_3 is shown in Fig. 17-8.

M_3 of course is non-deterministic. From its initial state it can go without reading any input to q_0, from which point it acts like M_1, or it can go to q_0' and then behave like M_2. Given a string x which is in $L_1 \cup L_2$, there will be an accepting path in M_3 corresponding to one (or both) of these possibilities. If x is not in $L_1 \cup L_2$, it is not accepted on any path in M_3.

It should not be difficult to see that the method of construction is general and can be applied to any two fa's. This is the basis of the proof that the

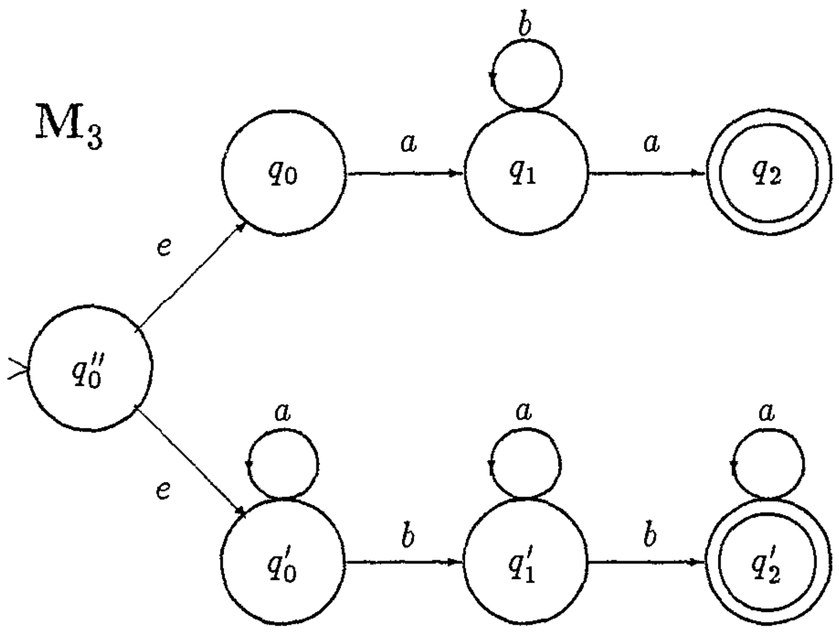


Figure 17–8.

fal's are closed under union.

The demonstration that the fal's are closed under concatenation is similar, except that the automata are hooked together "in series" rather than "in parallel." To construct a fa accepting $L_1 \cdot L_2$, we relabel the states of M_2, if necessary, to make them distinct from those of M_1 and then run e-transitions from all final states of M_1 to the initial state of M_2. Final states of M_1 now become non-final, but final states of M_2 remain. The result for our examples M_1 and M_2 above, would be as shown in Figure 17-9.

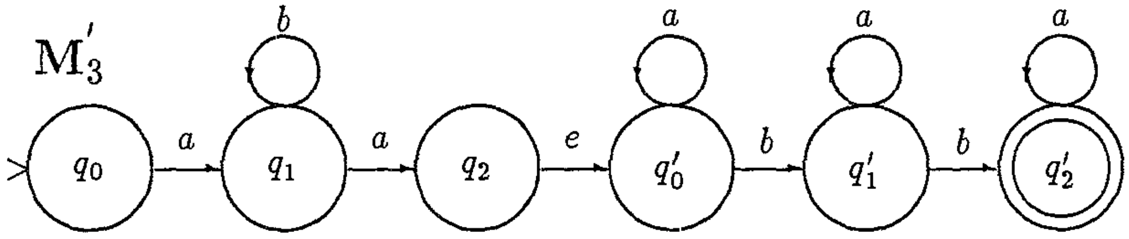


Figure 17–9.

M_3' accepts $L_1 \quad L_2$ as follows Given x, it will be accepted just in case $x = wz$, where w would be accepted by M_1 (in state q_2) and z would be accepted by M_2 (going from state q_0' to state q_2'.) If x is not in $L_1 \quad L_2$, there will be no factorization of x into wz such that w would be accepted by M_1 and z accepted by M_2 and thus no accepting path through M_3'. Again, the method is general and does not depend on the particular characteristics of automata M_1 and M_2 Note, however, that if M_1 had more than one final state, we would have e-transitions from *each* of them to the initial state of M_2, and all these states would become non-final in the resulting fa

Finally, we want to show that the fal's are closed under the Kleene star operation; that is, we want to take an automaton accepting L and convert it into an automaton accepting L^*. The strategy here is to establish e-transitions from all the final states back to the initial state so that the new fa can "recycle," accepting an input string $x_1x_2 \ldots x_n$ just in case the original fa would have accepted x_1 and $x_2 \ldots$ and x_n. There is a slight difficulty, however, in connection with the acceptance of the empty string, which is of course a member of L^* for any language L.

One would naturally want to insure that e is accepted by making the initial state a final state if it isn't already However, in certain cases this can lead to trouble Consider, for example, the following fa accepting the language $a(b \cup baa)^*b$:

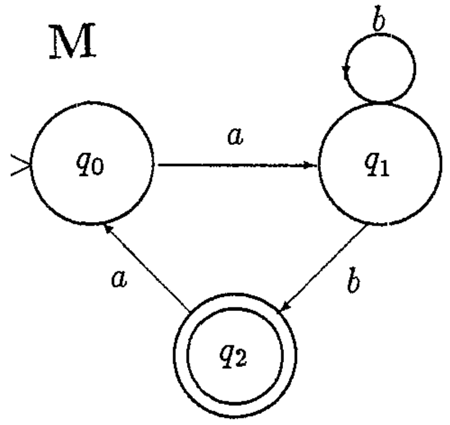


Figure 17–10.

If we were simply to add an e-transition from q_2 to q_0 so that the fa could "recycle" and make q_0 a final state, we would have:

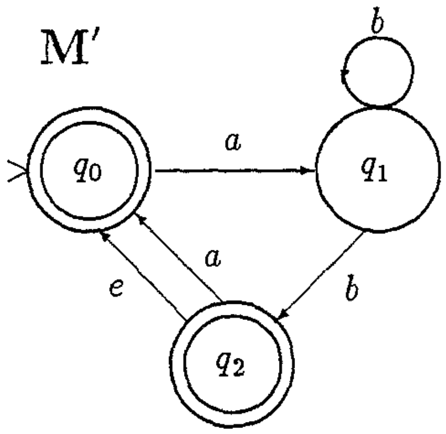


Figure 17–11.

M' accepts e, as required, but it also accepts aba, which is not in $(a(b \cup baa)^*b)^*$ Rather, what we should do is add a new initial state to M, which will also be a final state, and establish an e-transition from q_2 back to the *old* initial state, q_0:

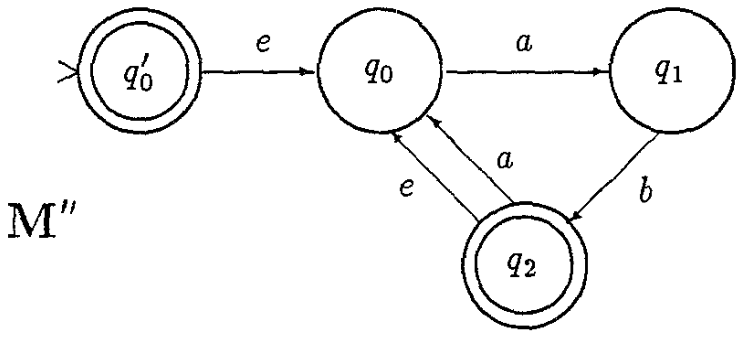


Figure 17–12.

Returning now to the examples M_1 and M_2 above, we see that this method of construction would produce the fa's in Fig. 17-13, accepting L_1^* and L_2^*, respectively.

This completes our informal demonstration that every finite automaton language is a regular language. As we have said, we will not attempt to show the converse here. It relies essentially on a procedure for extracting from any given fa a representation of the language it accepts. This representation can

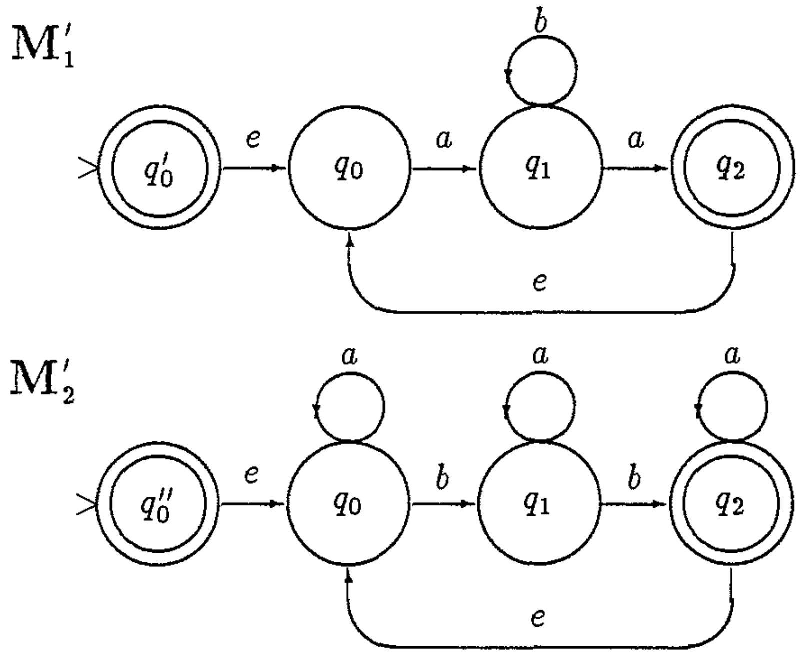


Figure 17–13.

be shown to involve only the empty and unit languages together with the operations of union, concatenation, and Kleene star, and thus, that every fal is a regular language.

17.2.1 Pumping Theorem for fal's

Consider an infinite fal L. By definition, it is accepted by some fa M, which, again by definition, has a finite number of states. But since L is infinite, there are strings in L which are as long as we please, and certainly L contains strings with more symbols than the number of states in M. Thus, since M accepts every string in L, there must be a loop in M—in particular, a loop which lies along a path from the initial state to some final state. In other words, in accepting a string longer than the number of states of M, M must enter some state more than once, and a path leading from such a state back to that state constitutes a loop along an accepting path.

Let x be the string of symbols which M reads on going from its initial state to the state at the beginning of some loop (call it q_i). Let y be the string read by M in going around the loop once, i.e., from q_i to the first re-entry to q_i. Finally, let z be the string read on going from q_i to some final state. Thus, xyz is accepted by M

But now notice that any loop lying along an accepting path can be traversed any number of times—zero or more—and the result will still be an accepting path. Therefore, if M accepts xyz in the way indicated, it will also accept $xz, xyyz, xyyyz, \ldots$, in fact all strings of the form xy^nz for $n \geq 0$. Finally, we observe that if L is an infinite language (so that there is no upper bound on sentence length), there has to be *some* loop along an accepting path in M such that $y \neq e$ Otherwise, M could not accept strings of length greater than the number of states of M. These observations are summarized in the following theorem, known as the Pumping Theorem (for fal's) because the string y is said to be "pumped", i.e., repeated with each traversal of the loop recognizing it.

THEOREM 17.2 *If L is an infinite fal over alphabet Σ, then there are strings $x, y, z \in \Sigma^*$ such that $y \neq e$ and $xy^nz \in L$ for all $n \geq 0$.* ∎

As an example, consider the language

(17–9) $\{x \in \{a, b\}^* \mid x \text{ contains an odd number of } b\text{'s}\}$

Since this is an infinite fal, the Pumping Theorem applies. Hence, there exist strings x, y, and z ($y \neq e$) such that $xy^nz \in L$ for all $n \geq 0$ Many examples of such strings could be found; to take just one, let $x = e$, $y = bb$, and $z = ab$. Then it is true that $ab, bbab, bbbbab, bbbbbbab, \ldots$, are all in L; that is $(bb)^nab \in L$ for all $n \geq 0$. For some choices of x, y, and z, this will not be true, but that doesn't matter: the theorem guarantees only that at least *one* choice for x, y, and z exists such that the specified condition holds.

The usefulness of this theorem lies in its application to languages which are not fal's. Suppose we have a language L which is infinite and for which we could somehow show that for *no* choice of x, y, and z ($y \neq e$) whatsover is it the case that $xy^nz \in L$ for all $n \geq 0$. In such a situation we would be justified in concluding that L is *not* a fal. Here we are using the theorem in its contrapositive form. As stated, it is a conditional: If L is an infinite fal, then so-and-so. The contrapositive is: if not-so-and-so, then L is not an infinite fal. For example, consider the language

(17–10) $L = \{a^nb^n \mid n \geq 0\}$

and let us show that it is not a fal using the Pumping Theorem. If L were a fal, there would be some x, y, and z ($y \neq e$) such that $xy^nz \in L$ for all $n \geq 0$ We show that no such x, y, and z exist.

The string xyz would have to be in L, so what could y consist of? It can't be empty, so it would have to consist of (1) some number of a's, or (2) some number of b's, or (3) some number of a's followed by some number of b's. It is easy to see that (3) is impossible because any string that contains more than one repetition of y, e.g., $xyyz$, will contain b's preceding a's—the b's at the end of the first y and the a's at the beginning of the next—which could not be a string in L. So such a choice of y is not pumpable.

What about case (1)? Here all the b's are contained in the z part, and as y is pumped, the number of a's in the string will increase while the number of b's remains constant Thus, we will continually be producing strings which have more a's than b's in them, which cannot be in the language a^nb^n.

Case (2) is parallel, but here the number of b's outstrips the number of a's. These are the only logical possibilities for the choice of y, and since none meet the condition laid down in the Pumping Theorem, we conclude that no such x, y, z exist for this language Conclusion: a^nb^n is not a fal.

What we have just done, then, is to show that there is no fa accepting a^nb^n without actually attempting to construct such an automaton.

One should also note that the Pumping Theorem does not yield particularly useful information when one shows that the consequent of the conditional is true. If we were to consider the language $L = \{x \in \{a,b\}^* \mid x$ contains equal numbers of a's and b's in any order $\}$ and observe that for $x = e$, $y = ab$, and $z = e$ it is the case that $xy^nz \in L$ for all $n \geq 0$, this would tell us nothing about whether L is a fal or not. Given $A \rightarrow B$ and B, we can conclude nothing about the truth or falsity of A. As it happens, the language just mentioned is *not* a fal. The moral is that the Pumping Theorem may be useful in showing that certain languages are not fal's but may not prove useful in other cases, even though the languages in question are in fact not fal's.

17.3 Type 3 grammars and finite automaton languages

We now want to examine the fal's from the point of view of grammars which generate them. Recall our previous discussion of formal grammars as consisting of V_T, the terminal vocabulary; V_N, the non-terminal vocabulary; S, the initial symbol; and R, a set of rules or productions. The various types of grammars differ in the form of productions they may contain, and here we want to focus our attention on Type 3 grammars, also called *right linear* grammars, in which each production is either of the form $A \to xB$ or $A \to x$, where A and B are in V_N and x is any string in V_T^*. That is, each rule of a Type 3 grammar has a single non-terminal on the left side, and on the right a string of terminals (possibly empty) followed by at most one non-terminal symbol An example is shown in (17-11)

(17–11) $G = \langle V_T, V_N, S, R \rangle$, where $V_T = \{a, b\}$; $V_N = \{S, A, B\}$; and

$$R = \begin{Bmatrix} S \to aA \\ A \to aA \\ A \to bbB \\ B \to bB \\ B \to b \end{Bmatrix}$$

An example of a derivation by this grammar is shown in (17-12):

(17–12) $S \Rightarrow aA \Rightarrow aaA \Rightarrow aabbB \Rightarrow aabbbB \Rightarrow aabbbb$

The phrase-structure tree associated with this derivation is shown in Fig. (17-14).

It is evident from this tree why the Type 3 grammars are also called right linear: the non-terminal symbols form a single linear sequence down the right of the tree. (There is also a class of grammars called *left linear* in which every rule is of the form $A \to Bx$ or $A \to x$. The more general class of *linear* grammars has every rule of the form $A \to xBy$ or $A \to x$, i.e., the right side of each rule has at most one non-terminl symbol but it need not to be left-most or right-most in the string. What we will say here about right-linear grammars could equally well be formulated in terms of left-linear grammars—they are equivalent in generative capacity. Linear grammars, however, generate a larger class of languages.)

Note that in a derivation by a right-linear grammar there is exactly one non-terminal symbol at the right end of each string until the last one, at which point a rule of the form $A \to x$ is applied and the derivation terminates. This observation suggests an analogy to finite automata: the one non-terminal symbol at the right side of a string is like the state of a fa

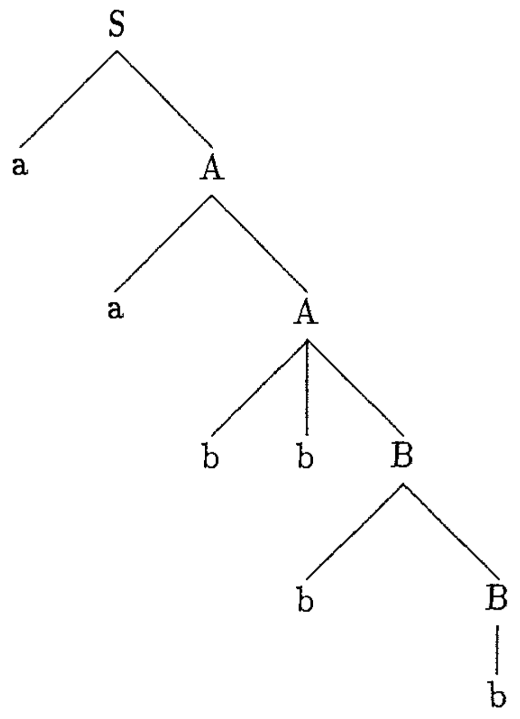


Figure 17–14.

in that the future course of the derivation or computation can depend only on the identity of that state or symbol and the given productions of the device in question. In particular, the past history, i.e., the string already read by the fa or the string already generated by the grammar, has no influence on the future course of events.

Let us then associate with each rule of a Type 3 grammar of the form $A \rightarrow xB$ a transition in a (non-deterministic) fa from state A to state B reading x. Further, let us associate each rule of the form $A \rightarrow x$ with a tranisition from state A reading x to a designated final state F. The initial state of the automaton will, of course, be S. Carrying out this construction for the grammar in (17-11) gives:

It should be reasonably easy to convince oneself that this fa accepts the same language as that generated by the grammar in (17-11). Moreover, the method is general and can be applied to any given Type 3 grammar to produce an equivalent fa. 'Equivalent,' of course, means that the language accepted by the fa is the same as the language generated by the Type 3 grammar.

On further consideration, we see that there is no reason why we should

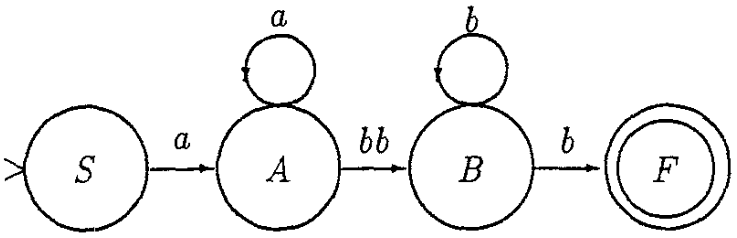


Figure 17–15.

consider fa's only as acceptors of languages. We might as well think of an fa as a language generator which starts in an initial state, moves from state to state emitting, or writing, symbols on an output tape, and halting at will in either a final or non-final state. If the fa halts in a final state, the output string is said to be generated; otherwise, it is not generated. The state diagram of a fa is the same whichever way we want to look at it, and the same language would be accepted by a given fa acceptor as that generated by the fa regarded as generator. From this point of view, then, non-deterministic fa's and Type 3 grammars are virtually isomorphic representations.

Problem: How could one construe a Type 3 grammar as accepting rather than generating strings, i.e., how would the rules and derivations be interpreted?

What we have just argued (without giving a formal proof) is that every Type 3 language is a fal. To show the converse is equally easy, but the construction proceeds in the opposite direction. Given a fa, we use its instructions to create the rules of a Type 3 grammar in the following way. For each transition (q_i, x, q_j), we put in the grammar a rule $q_i \to xq_j$. Thus, the states of the fa become non-terminal symbols of the grammar, and the alphabet of the fa becomes the terminal alphabet of the grammar Finally, for each transition (q_i, x, q_j) where q_j is a final state, we also add to the grammar the rule $q_i \to x$. If we carry out this construction on the fa in (17-1), we get the following grammar:

(17–13) $G = \langle V_T, V_N, q_0, R\rangle$, where $V_T = \{a, b\}$; $V_N = \{q_0, q_1\}$; and

$$R = \begin{Bmatrix} q_0 \to a\, q_0 & q_1 \to b\, q_0 \\ q_0 \to b\, q_1 & q_0 \to b \\ q_1 \to a\, q_1 & q_1 \to a \end{Bmatrix}$$

A derivation of the string *aaba* by this grammar would be as follows:

(17–14) $q_0 \Rightarrow aq_0 \Rightarrow aaq_0 \Rightarrow aabq_1 \Rightarrow aaba$

The reader may find it instructive to compare this with an accepting computation for *aaba* by the fa in (17-1).

To be rigorous we would have to give a proof that the method of construction just outlined does indeed produce a grammar equivalent to the original fa, but we will not do so here since the equivalence is intuitively evident The main point is that we now have three quite different characterizations of the same class of languages: the languages accepted (or generated) by (deterministic or non-deterministic) finite automata, the regular languages, and the languages generated by Type 3 grammars It is always useful to view mathematical objects from different perspectives; our understanding is enhanced, and new methods of proof are opened up. We also come to realize that we are dealing with a coherent, and in some sense "natural" mathematical class.

17.3.1 Properties of regular languages

We also gain in understanding of mathematical objects when we ascertain their behavior under various sorts of operations. Since languages are sets, it is natural to ask how they behave when subjected to certain set-theoretic manipulations. We have already seen, for example, that the class of fal's (= regular languages = Type 3 languages) is closed under the operation of union: i.e., the union of any two fal's is also a fal. Similarly, we know that the fal's are closed under concatenation and Kleene star. What about the operations of complementation and intersection?

Given a regular language $L \in \Sigma^*$, its complement, i.e., $\Sigma^* - L$, is also regular How can we show it? Given our equivalent characterizations of this class of languages, we can make use of whichever one is most convenient for what we want to prove. In this case, the desired result is easiest to show with finite automata.

Let M be a deterministic fa accepting L. Construct a new fa M' from M by interchanging final and non-final states. That is, M' is identical to M except that all final states are now non-final and vice versa. M' is also deterministic. Now M and M' read any input string in the same way, in the sense that for a given string they go through the same state transitions. The only difference is that when M accepts (ends in a final state), M' rejects (ends in a non-final state), and when M rejects, M' accepts. Thus, M' accepts the complement of L, which is therefore also a fal.

For example, applying this construction to the fa of (17-1), we obtain the following deterministic fa which accepts $\{x \in \{a, b\}^* \mid x$ contains an even number of b's$\}$ This is clearly the complement of the original language.

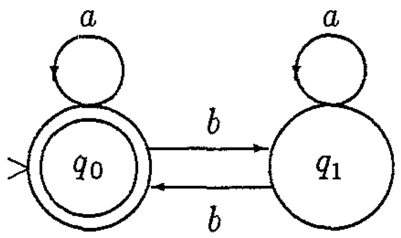


Figure 17–16.

Problem: Why wouldn't this procedure work in general if the original fa were not deterministic?

It now follows that the regular languages are also closed under intersection, since for any sets X and Y, $X \cap Y = (X' \cup Y')'$ by DeMorgan's Laws In more detail, if X and Y are regular languages, then so are their complements, X' and Y', as we have just seen The union of the latter, $X' \cup Y'$, is also regular, and the complement of the last set is also regular, i.e., $(X' \cup Y')'$, which is equal to $X \cap Y$.

Given then that the regular languages are closed under union, intersection and complementation, and that the empty language and Σ^* (for any given alphabet Σ) are regular, we have the result that the regular languages over any fixed alphabet form a Boolean algebra (see Ch. 12). This gives us some information about the class of regular languages but does not provide a complete characterization since there are other sets of languages which also form Boolean algebras (e.g., the set of *all* languages over a given alphabet) which are not regular.

Another frequently encountered problem concerning mathematical objects is this: What questions about them can be answered by algorithm? Or to put it another way, do there exist procedures which can be applied mechanically to any instance of one of these objects to yield an answer to a particular sort of question in a finite time?

An example of this sort of consideration as applied to finite automata would be: Given a fa M and a string x, can it be determined whether M accepts x or not? The answer in this case is yes One procedure for answering this question would be the following. Given M, convert it to an equivalent deterministic fa M' (if M is not already deterministic.) There is an algorithm for performing this conversion, as we mentioned earlier If $x \notin \Sigma^*$, we know it cannot be accepted, since it contains symbols not in the alphabet of M' If $x \in \Sigma^*$, trace the computation of x by M'. There is a unique path through the state diagram of M' reading x which ends in some state q_i If q_i is a final state, x is accepted; otherwise, x is rejected. Since the first part of the algorithm will yield an equivalent deterministic fa in a finite amount of time, the number of states of M being finite, and the second part will be accomplished in a finite number of steps, viz , the number of symbols of x, the procedure outlined is guaranteed to terminate after a finite time with the correct answer This is an algorithmic solution to the so-called *membership question* for fal's.

Another example of a question about fa's which has an algorithmic solution is the *emptiness question*: given an fa M, does it accept any strings at all? One could proceed as follows If M is not deterministic, make it so. The result is M', which necessarily has a finite number of states and a finite number of transitions. M' accepts at least one string just in case there is a path in its state diagram from the initial state to some final state Furthermore, if there is any accepting path in M', there is an accepting path without loops in it (Any path with loops can also be traversed by going through each loop zero times.) Since the number of states and state connections is finite, there are only a finite number of loop-free paths to examine to determine whether any ends in a final state One could imagine systematic ways of looking at the paths, but the essential part here is not the relative efficiency of the process but only that it is a finite task. Thus, there is an algorithmic solution to the emptiness question for fa's.

Similarly, one might ask of a given fa, does it accept all strings in Σ^*? This can be reduced to the previous question by noting that Σ^* is the complement of $\emptyset$. Given M, make M deterministic if it isn't already, to produce M'. Interchange final and non-final states of M' to produce M''. Apply the algorithm for answering the emptiness question to M''. M'' accepts $\emptyset$ iff M' accepts Σ^*.

Problem: Given two fa's, M_1 and M_2, show that there is an algorithm for answering the question, is $L(M_1) \subseteq L(M_2)$ (Hint: $X \subseteq Y$ iff $(X \cap Y') = \emptyset$.)

Is there an algorithmic solution to the question of whether two fa's accept the same language?

17.3.2 Inadequacy of right-linear grammars for natural languages

Is English a regular language? We can prove that it is not, using the Pumping Theorem and the fact that regular languages are closed under intersection.

We assume that all the following are grammatical (although in some cases surely incomprehensible) English sentences:

(17–15) (1) The cat died.
(2) The cat the dog chased died.
(3) The cat the dog the rat bit chased died.
(4) The cat the dog the rat the elephant admired bit chased died.
etc.

These are all of the form:

(17–16) $(\textit{the} + \text{common noun})^n\ (\text{transitive verb})^{n-1}$ intransitive verb

Let us take some finite set A of common noun phrases of the form *the* + common noun:

(17–17) $A = \{$the cat, the dog, the rat, the elephant, the kangaroo, . . $\}$

Let us also choose a finite set B of transitive verbs:

(17–18) $B = \{$chased, bit, admired, ate, befriended, . . .$\}$

Thus, the strings illustrated in (17-15) are all of the form:

(17–19) $x^n y^{n-1}$ *died*, where $x \in A$ and $y \in B$.

The language L consisting of all such strings, of which the sentences in (17-15) are members, is easily shown not to be regular. The proof uses the

Pumping Theorem and is very similar to the proof that $\{a^n b^n \mid n \geq 0\}$ is not regular.

L is the result of intersecting English (considered as a set of strings) with the *regular* language $A^* B^* \{died\}$. Since the regular languages are closed under intersection, if English were regular, L would be also. Thus, English is not regular

The demonstration that English is not a finite automaton language was one of the first results to be achieved in the nascent field of mathematical linguistics (Chomsky, 1956; 1957, Chapter 3), although Chomsky did not use this particular method of proof, nor did he focus on the particular subset of English exemplified in (17-15). Rather, he pointed out that English has a certain number of constructions such as *either . . . or*, *if . . then*, and the agreement between the subject of a sentence and the main verb, which can be thought of as obligatorily paried correspondences or *dependencies*. (In sentences of the form *Either* S_1 *or* S_2, we cannot substitute *then* or *and* for *or*, for example, and similarly, we cannot replace *then* in *If* S_1 *then* S_2 by *and* or *or*, etc , and produce a grammatical sentence) Further, these dependencies can be found in grammatical sentences nested one inside the other and to an arbitrary depth. Chomsky and Miller (1963) cite the following example in which the dependencies are indicated by subscripts:

(17–20) Anyone$_1$ who feels that if$_2$ so-many$_3$ more$_4$ students$_5$ whom we$_6$ haven't$_6$ actually admitted are$_5$ sitting in on the course than$_4$ ones we have that$_3$ the room had to be changed, then$_2$ probably auditors will have to be excluded, is$_1$ likely to agree that the curriculum needs revision.

This structure of nested dependencies finds an analog in the strings of a language like $\{xx^R \mid x \in \{a, b\}^*\}$ (recall that x^R denotes the reversal of the string x) In strings of this language, the i^{th} symbol from the left must match the i^{th} symbol from the right as indicated in the diagram of Fig. 17-17:

The language xx^R can be shown not to be regular by first intersecting it with the regular language aa^*bbaa^* to give $\{a^n b^2 a^n \mid n \geq 1\}$ and showing that the latter is not regular by means of the Pumping Theorem.

This result illustrates one sort of practical result that can sometimes be obtained from the study of formal grammars and languages. A linguistic theory proposes that the grammar of every natural language is drawn from some infinite class $\mathcal{G}$ of generative devices. (This is just to say that the

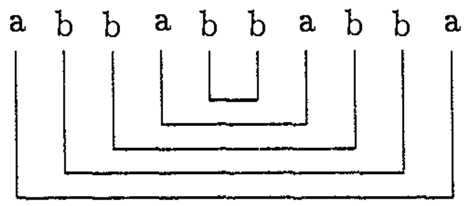


Figure 17–17

linguistic theory specifies, as it should, the form that grammars may take.) Such a theory is supported, but not of course proven true, by each succesful prediction, i e., whenever we are able to show that a grammar from $\mathcal{G}$ is adequate for some natural language. On the other hand, repeated failure to find an adequate grammar in $\mathcal{G}$ for, say, Swahili might raise doubts but would not suffice to prove the theory wrong Since $\mathcal{G}$ is an infinite set, as it will be in all interesting cases, failure after a finite number of attempts may reflect only ineptness or bad luck. In certain cases, however, we may be able to demonstrate conclusively that a linguistic theory is inadequate for one or more natural languages, as we just did for the theory of right linear grammars vis-a-vis English. We are then justified in concluding that the theory is inadequate in principle and can be removed from consideration as a viable proposal

Exercises

1. Consider the following state diagram:

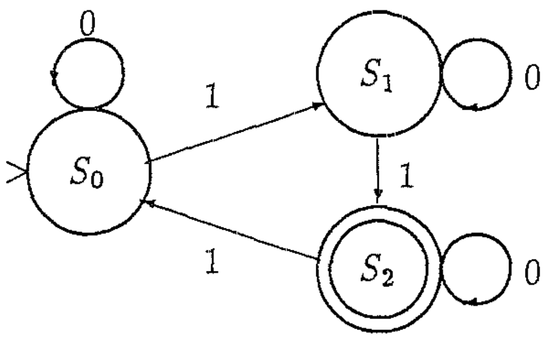


Figure 17–18.

(a) Which of the following strings are accepted by the machine?
(i) 01011
(ii) 0011
(iii) 11001101
(iv) 01010111111

(b) Describe as simply as possible the language accepted by the automaton.

2. Consider the following set of transition rules:

$(S_0, 0) \to S_0$	$(S_3, 0) \to S_0$
$(S_0, 1) \to S_1$	$(S_3, 1) \to S_4$
$(S_1, 0) \to S_2$	$(S_4, 0) \to S_0$
$(S_1, 1) \to S_3$	$(S_4, 1) \to S_5$
$(S_2, 0) \to S_0$	$(S_5, 0) \to S_0$
$(S_2, 1) \to S_1$	$(S_5, 1) \to S_5$

Final states: S_4, S_5.

(a) Draw a state diagram for this automaton

(b) Describe the set of input strings accepted by the automaton.

(c) Draw a state diagram for an automaton which is equivalent to this one but which has four states.

3. Construct state diagrams for finite automata which accept the following languages using as few states as possible:

(a) The set of all strings containing a total of n 1's, where n is congruent to 1 (modulo 3) (i.e., the remainder when n is divided by 3 is 1).

(b) The set of all strings containing a total of exactly two 1's.

(c) The set of all strings which contain a block of at least three consecutive 1's, e.g.,

010111001	but not	0101011011
001111100		0011011000
1101110111		1101100101

(Note that once such a block occurs, it is irrelevant what comes later.)

(d) The set of all strings which contain no block of more than one consecutive 0 nor any block of more than one consecutive 1, e.g.,

0	but not	0101101
e		11
101		11010
101010		
0101		

(e) The set of all strings which contain the substring 101 anywhere within them

(f)* The set of all strings in which the total number of 0's is congruent to the total number of 1's modulo 3 (see part (a) above).

4. Consider non-deterministic finite automata whose input alphabet is {*the, old, man, men, is, are, here, and*}.

 (a) Construct a state diagram for an automaton which accepts the following language: {*the man is here, the men are here*}.

 (b) Do the same for the following language: {*the man is here, the men are here, the old man is here, the old men are here, the old old man is here, the old old men are here,...*}.

 (c) Construct a state diagram for an automaton which accepts all the sentences in (b) plus all those formed by conjoining sentences with *and*, e.g., *the old man is here and the old old men are here and the men are here*.

5. (a) Construct a state diagram for an automaton which accepts the terminal language of the following grammar. (The input alphabet is $\{a, b, c\}$; it does not include S or C.)

$S \to SS \quad S \to bb$
$S \to aCa \quad C \to Cc$
$S \to bCb \quad C \to c$
$S \to aa$

 (b) Draw a diagram for an automaton whose language is that of part (a) plus the empty string

6. Show how, given any finite automaton, you can construct an equivalent one which has no transition arrows leading to the initial state

All automata asked for in exercises 7 and 8 may be non-deterministic, and of course must be finite state. The input alphabet is to be $\{0, 1\}$

7. (a) Construct an automaton A which accepts any string which contains no 1's.

(b) Construct an automaton B which accepts any string which contains an odd number of 1's, with any number (including zero) of 0's.

(c) Construct an automaton C which accepts the union $L(A) \cup L(B)$.

(d) Construct an automaton D which accepts the complement $L(C)'$. (*Caution*: first find a deterministic equivalent of C.) Describe $L(D)$ in words

8. (a) Construct an automaton A which accepts any string which contains *no* block of four or more consecutive 1's

(b) Construct an automaton B which accepts any string which contains no block of three or more consecutive 0's.

(c) Construct an automaton C which accepts the intersection $L(A) \cap L(B)$.

9. Find a regular expression for the language of:

(a) Exercise 4a (above)

(b) Exercise 4b.

(c) Exercise 4c.

(d) Exercise 3b.

(e) Exercise 3e.

10. Draw a state diagram for a finite automaton corresponding to the following regular expressions.

(a) 010^*1

(b) $(010^*1)^*$

(c) $(010^*1)^*1$

(d) $(010^*1)^*1(0 \cup 1)^*$

(e) $1(010^*1)^*$

(f) $1(010^*1)^*(0 \cup 1)^*$ (Hint: this one can be done by a simpler machine than any of the others.

11. Construct Type 3 grammars that generate each of the following languages Assume a fixed terminal vocabulary $V_T = \{a, b\}$.

(a) $L_1 = \{aa, ab, ba, bb\}$

(b) $L_2 = \{x \mid x$ contains any number of occurrences of a and b in any order$\}$

(c) $L_3 = \{x$ contains exactly two occurrences of a, not necessarily contiguous$\}$

(d) $L_4 = \{x \mid x$ contains exactly one occurrence of a, or exactly one occurrence of b, or both$\}$

(e) $L_5 = \{x \mid x$ contains an even number of a's and an even number of b's$\}$ (Zero counts as even)

(f) $L_6 = L_3 \cap L_5$

12. Construct finite automata (non-deterministic) accepting each of the languages in Exercise 11.