Regular Expressions (Kleene's Theorem)

The above Myhill-Nerode Theorem provides a complete characterization of properties that can be determined with finite-state machines. A totally different characterization is based on "regular expressions," which you may know as search patterns in text editors.

Regular languages and expressions are defined inductively as follows.

Definition 3 (Regular languages) *1. The empty set $\emptyset$ is regular.*

2. Any set containing one character is regular.

3. If A and B are regular then so are $A \cup B$, AB, and $A^\star$.

[The union, concatenation and Kleene star operations are defined as usual:

$$A \cup B = \{x : x \in A \text{ or } x \in B\},$$

$$AB = \{xy : x \in A \text{ and } y \in B\},$$

and

$$A^\star = \{x_1 x_2 \cdots x_k : k \geq 0 \text{ and } x_i \in A \text{ for all } i\}$$

.]

Observation 4 *Finite languages are regular.*

Definition (Regular expressions) *1. The symbol $\emptyset$ is a regular expression and $L(\emptyset) = \emptyset$.*

2. Any single character c is a regular expression and $L(c) = \{c\}$.

3. If α and β are regular expressions then so are $\alpha \cup \beta$, $\alpha\beta$, and $\alpha^\star$, and $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha\beta) = L(\alpha)L(\beta)$, and $L(\alpha^\star) = (L(\alpha))^\star$.

Observation 5 *A language L is regular if and only if there is a regular expression E with $L = L(E)$.*

The important result in this section is the following.

Theorem 16 (Kleene's Theorem) *The following two statements are equivalent.*

1. L is regular.

2. L is accepted by some finite-state machine.

Proof Both parts of this proof are inductive constructions.

To construct regular expressions from finite-state machines we number the states and carry out an inductive construction, with the induction being over this numbering of the states. This is a nontrivial example of a proof by induction and a nice illustration of the fact that inductive proofs often become easier once you decide to prove a stronger result than you really wanted.

To construct finite-state machines from regular expressions we do an "induction over the structure" of the expression. This is a nice example of the fact that induction, while strictly speaking a technique that is tied to natural numbers, can be thought of as extending to other sets as well.

Here are the details.

(2) ⇒ (1) To construct a regular expression from a given finite-state machine M, first number the states of the machine $1, \ldots, q$, with state 1 being the start

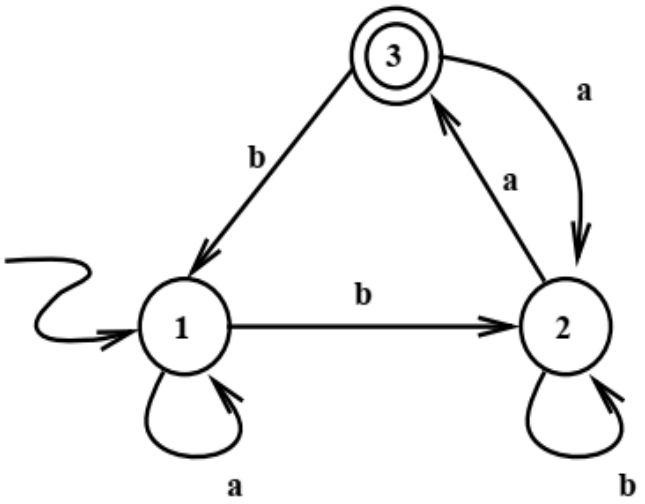


Figure 2.18: Illustrating the notion of a string making a machine "pass through" states

state. Define L_{ij}^k to be the set of all strings which make the machine go from state i into state j without ever "passing through" any state numbered higher than k. For example, if the machine of Figure 2.18 happens to be in state 1 then the string ba will make it first go to state 2 and then to state 3. In this sequence $1 \to 2 \to 3$ we say that the machine "went through" state 2. It did not "go through" states 1 or 3, which are only the starting point and the endpoint in the sequence. Since it did not go through any state numbered higher than 2 we have $ba \in L_{13}^2$. Other examples are $babb \in L_{32}^2$ and $a \in L_{23}^0$, in fact $\{a\} = L_{23}^0$. In this example, the language accepted by the machine is L_{13}^3. In general, the language accepted by a finite-state machine M is

$$L(M) = \bigcup_f L_{1f}^q \tag{2.11}$$

where the union is taken over all accepting states of M. What's left to show is that these languages L_{ij}^k are always regular. This we show by induction on k.

For $k = 0$, L_{ij}^0 is finite. It contains those characters that label a transition from state i to state j. If $i = j$ the empty string gets added to the set.

For $k > 0$,

$$L_{ij}^k = (L_{ik}^{k-1}(L_{kk}^{k-1})^\star L_{kj}^{k-1}) \cup L_{ij}^{k-1}. \tag{2.12}$$

This equation amounts to saying that there can be two ways to get from state i to state j without going through states numbered higher than k. Either we go through k, or we don't, corresponding to the subexpression before the union and the subexpression after the union. Figure 2.19 illustrates this. Figure 2.20 shows some of the steps in the construction of a regular expression from the finite-state machine of Figure 2.18. We simplify the subexpressions as we go. Expression E_{ij}^k describes language L_{ij}^k: "$L(E_{ij}^k) = L_{ij}^k$." Some parentheses are omitted. λ is short for $\emptyset^\star$, hence $L(\lambda) = \{\lambda\}$, by the definition of the *-operation. (We wouldn't want to miss a chance to be obscure, would we?)

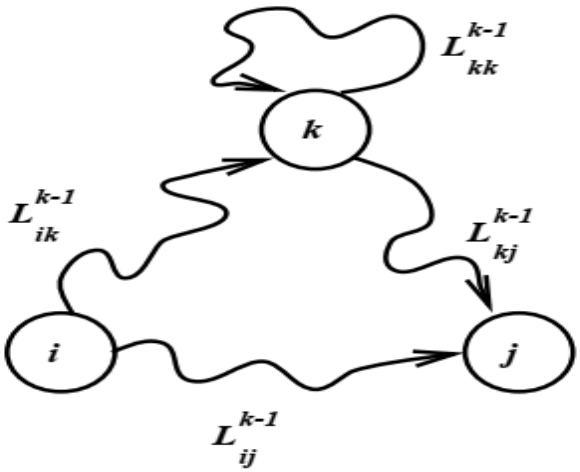


Figure 2.19: The pattern of induction for constructing the regular expressions L_{ij}^k in Kleene's Theorem

(1) ⇒ (2) To construct finite-state machines from regular expressions we use what is called "induction on the structure of the expression" or "structural induction." This means that finite-state machines are first constructed for small subexpressions, then combined to form machines for larger subexpressions and ultimately for the whole expression.

For example, if we want to convert the expression

$$E = (abb) \cup (b(aa)^\star bab)$$

a machine $M_{(aa)^\star}$ for the subexpression $(aa)^\star$ is shown in figure 2.21.

For simplicity let's adopt the convention that if a transition goes to a rejecting state from which the machine cannot escape anymore then we won't draw the transition, nor that "dead" state. This simplifies the picture for $M_{(aa)^\star}$, see Figure 2.22. A machine M_b is easier yet, see Figure 2.23. We can combine M_b and $M_{(aa)^\star}$ to get $M_{b(aa)^\star}$, shown in Figure 2.24. Adding parts that correspond to subexpressions *abb* and *bab* completes the construction, see Figure 2.25. This seems easy enough, but what if the expression had been just slightly different, with an extra b in front:

$$E' = (babb) \cup (b(aa)^\star bab)$$

Then our construction would have gotten us this machine $M_{E'}$ in Figure 2.26.

This machine $M_{E'}$ has a flaw: there are two transitions labeled b out of the start state. When we run such a machine what will it do? What should it do, presented with such an ambiguity? *Can we define the "semantics," i.e., can we define how such a machine is to run, in such a way that the machine $M_{E'}$ accepts the very same strings that the expression E' describes?*

$$
\begin{array}{lcllcllcl}
E^0_{11} & = & \lambda \cup a & E^0_{12} & = & b & E^0_{13} & = & \emptyset \\
E^0_{21} & = & \emptyset & E^0_{22} & = & \lambda \cup b & E^0_{23} & = & a \\
E^0_{31} & = & b & E^0_{32} & = & a & E^0_{33} & = & \lambda
\end{array}
$$

$$E^1_{32} = (E^0_{31}(E^0_{11})^\star E^0_{12}) \cup E^0_{32} = (b(\lambda \cup a)^\star b) \cup a = ba^\star b \cup a$$

$$
\begin{array}{lcllcllcl}
E^1_{11} & = & a^\star & E^1_{12} & = & a^\star b & E^1_{13} & = & \emptyset \\
E^1_{21} & = & \emptyset & E^1_{22} & = & \lambda \cup b & E^1_{23} & = & a \\
E^1_{31} & = & ba^\star & E^1_{32} & = & ba^\star b \cup a & E^1_{33} & = & \lambda
\end{array}
$$

$$E^2_{13} = (E^1_{12}(E^1_{22})^\star E^1_{23}) \cup E^1_{13} = (a^\star b(b^\star)^\star a) \cup \emptyset = a^\star bb^\star a$$

$$
\begin{array}{lcllcllcl}
E^2_{11} & = & a^\star & E^2_{12} & = & a^\star b^+ & E^2_{13} & = & a^\star bb^\star a \\
E^2_{21} & = & \emptyset & E^2_{22} & = & b^\star & E^2_{23} & = & b^\star a \\
E^2_{31} & = & ba^\star & E^2_{32} & = & ba^\star b^+ & E^2_{33} & = & (ba^\star b \cup a)b^\star a
\end{array}
$$

$$E^3_{11} = (E^2_{13}(E^2_{33})^\star E^2_{31}) \cup E^2_{11} = (a^\star bb^\star a((ba^\star b \cup a)b^\star a)^\star ba^\star) \cup a^\star$$

Figure 2.20: Converting the finite-state machine from Figure 2.18 into a regular expression

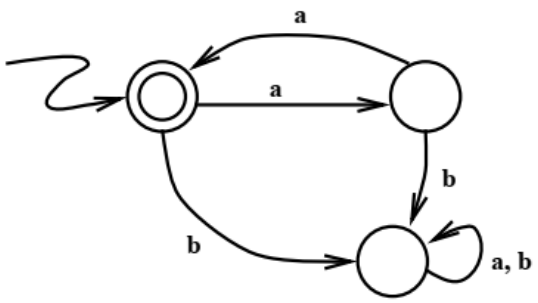


Figure 2.21: The machine $M_{(aa)^\star}$

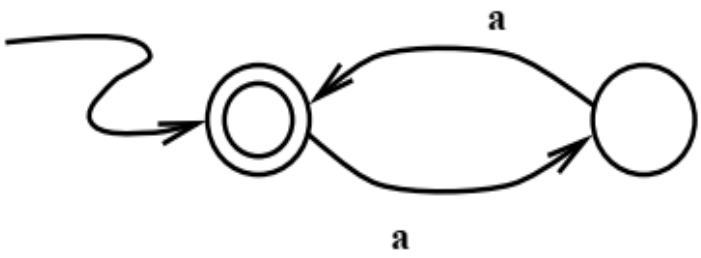


Figure 2.22: The machine $M_{(aa)^\star}$, a simplified drawing

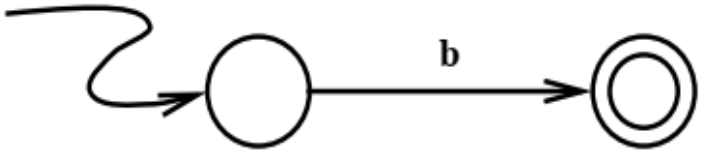


Figure 2.23: The machine M_b

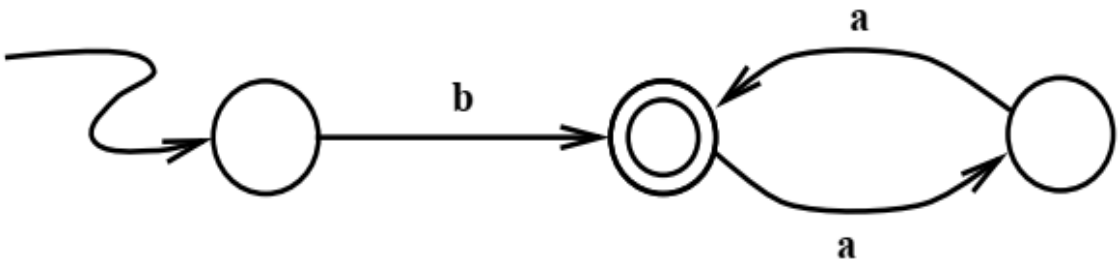


Figure 2.24: The machine $M_{b(aa)^\star}$

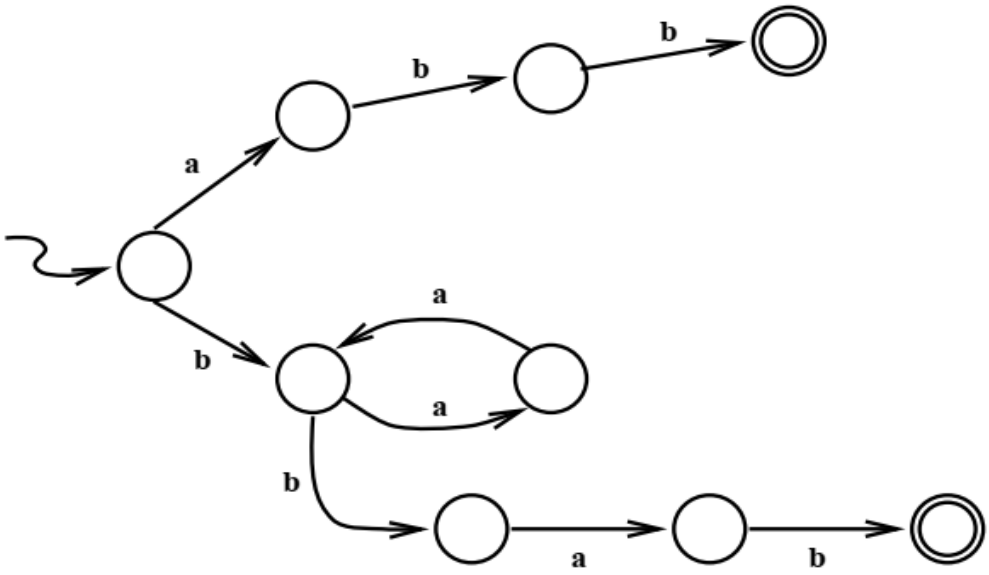


Figure 2.25: The machine M_E

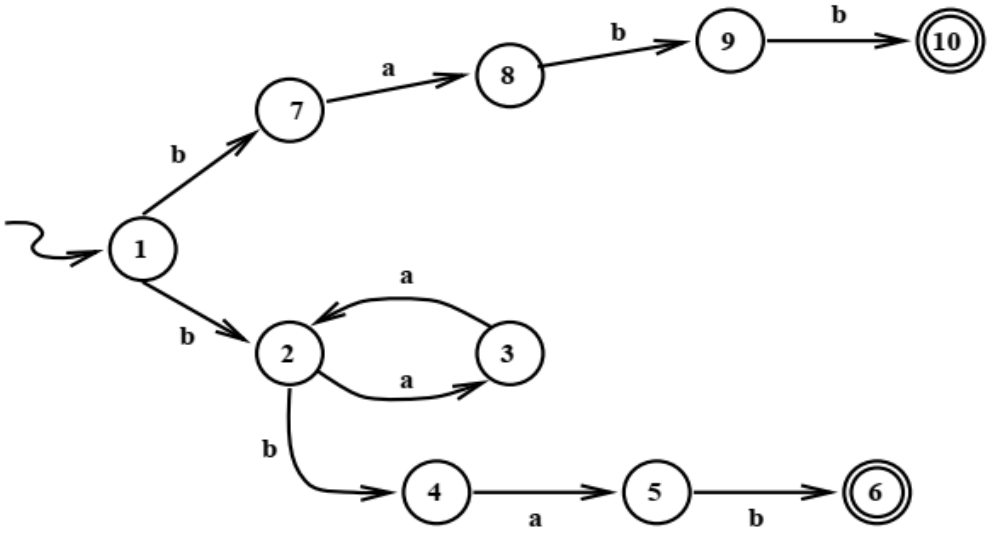


Figure 2.26: The machine $M_{E'}$

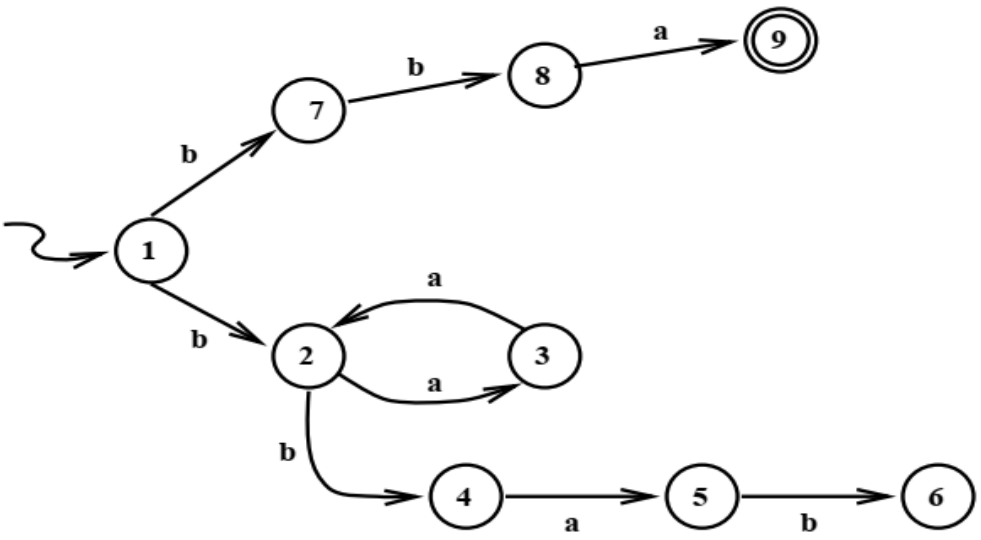


Figure 2.27: The machine $M_{E''}$

After reading an initial b, should $M_{E'}$ be in state 2 or in state 7? We want to accept both *babb* and *baabab*. Being in state 7 after the first b give us a chance to accept *babb*; and being in state 2 gives us a chance to accept *baabab*. This suggests that the right way to run this machine is to have it be *in both states* 2 *and* 7 after reading the first b.

With this way of running the machine, which states would it be in after reading an a as the second character? In states 3 and 8. After reading a b as the third character, it can only be in state 9. [7] After reading another b — for a total input string of *babb* — $M_{E'}$ is in state 10 and accepts the input.

There is one more issue that this example does not address. What if some such machine ends up in a set of states some of which are accepting and some are rejecting? The expression

$$E'' = (bba) \cup (b(aa)^\star bab)$$

provides an example. Our construction will result in the machine $M_{E''}$ of Figure 2.27. After reading *bba*, the machine is in state 5, which is rejecting, and in state 9, which is accepting. Since *bba* is a string described by E'', it should be accepted by $M_{E''}$. That settles the issue: *a string is accepted if at least one of the states it puts the machine in is an accepting state.*

Our conventions about running these machines and knowing when a string is accepted are the "semantics of nondeterministic finite-state machines." These

[7] This could be up to some discussion depending on how we view our convention of not drawing a "dead" state, i.e., a rejecting state from which the machine cannot escape. If we take the view that the state is there — we just don't draw it — then at this point we should say that the machine would be in state 9 and that invisible state, say 11. This is clumsy. It is more elegant — although ultimately of no important consequence — to say that the machine is only in state 9.

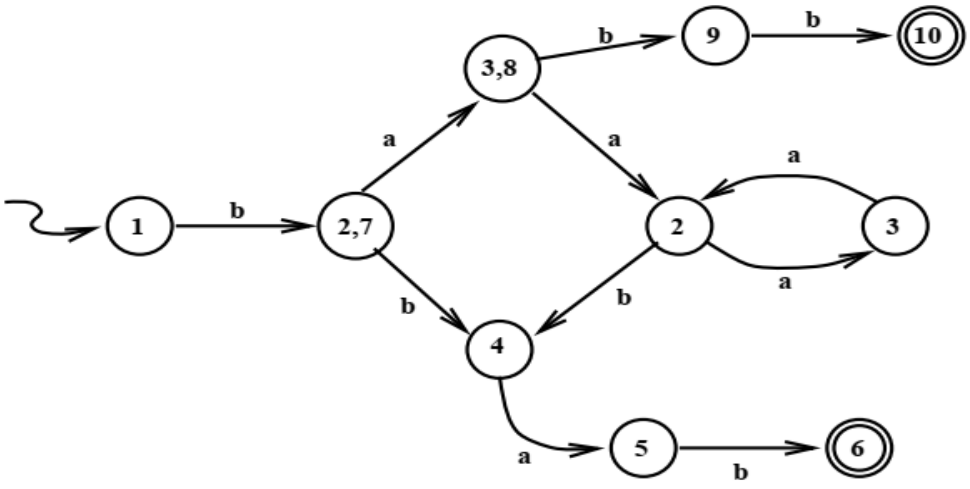


Figure 2.28: Converting $M_{E'}$ into a deterministic machine

conventions can be applied to more powerful programs as well (and we will do so in a later chapter on computational complexity, where the concept of nondeterminism will play a crucial role in defining a class of computational problems). "Nondeterministic C," for example, could be C with the extra feature of a nondeterministic branch:

```
( nondet-branch; x = 0; x = 1; )
```

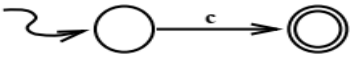
The way to think about the execution of such nondeterministic programs is that there is more than one path of execution. Tracing the execution results in a tree and a program is said to accept its input if at least one of the paths leads to acceptance.

Can we convert this new kind of machine into a standard (deterministic) finite-state machine? This task amounts to a search of the "execution tree," which we do in a breadth-first approach. Figure 2.28 gives an example, converting the machine $M_{E'}$ into a deterministic one. The states of the deterministic machine are the *sets* of states of $M_{E'}$. (Many of these sets of states may not be reachable from the start state and can be discarded.) In the deterministic machine, the state reached from a state A via a transition c, $\delta(A, c)$, is the set of those states b of $M_{E'}$ to which there is a c-transition (in $M_{E'}$) from some state $a \in A$.

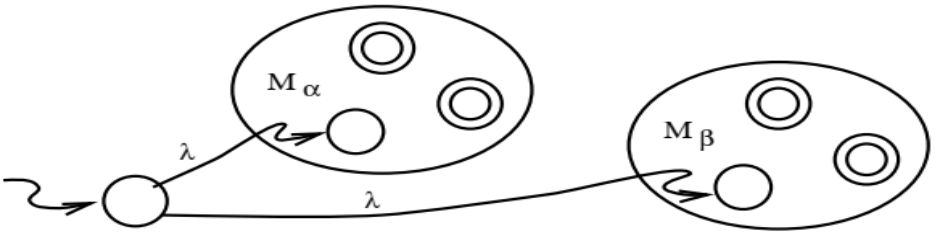
The algorithm for constructing a nondeterministic machine from a regular expression is easier to describe if we allow yet another form of nondeterministic behavior: making a "spontaneous" transition, i.e., a transition that does not consume an input character. The standard term is a "λ-transition." The inductive algorithm for constructing a (nondeterministic) finite-state machine M_E from an arbitrary regular expression E then works as shown in Figure 2.29. An example that uses all the parts of this algorithm is the machine shown in Figure 2.30.

1. $M_\emptyset$ is the following machine:

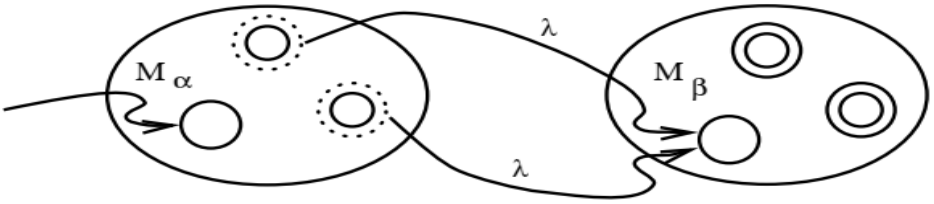
2. For any single character c, M_c is the following machine:



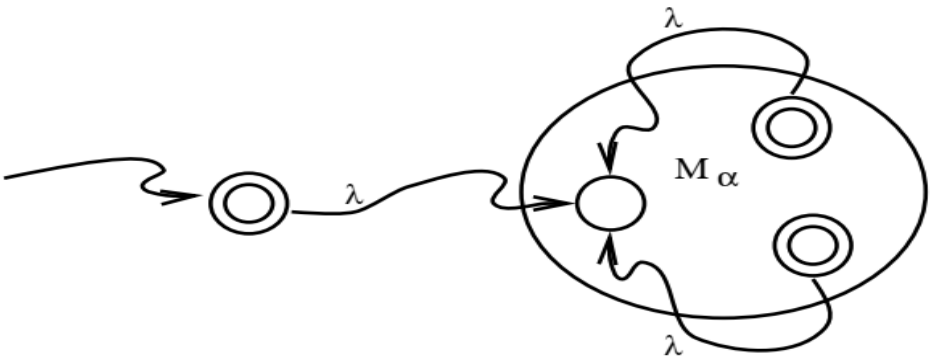
3. If α and β are regular expressions then $M_{\alpha\cup\beta}$, $M_{\alpha\beta}$, and M_{α^*} are constructed from M_α and M_β as shown below.



Constructing $M_{\alpha\cup\beta}$ from M_α and M_β



Constructing $M_{\alpha\beta}$ from M_α and M_β



Constructing M_{α^*} from M_α

Figure 2.29: An algorithm for converting regular expressions to finite-state machines

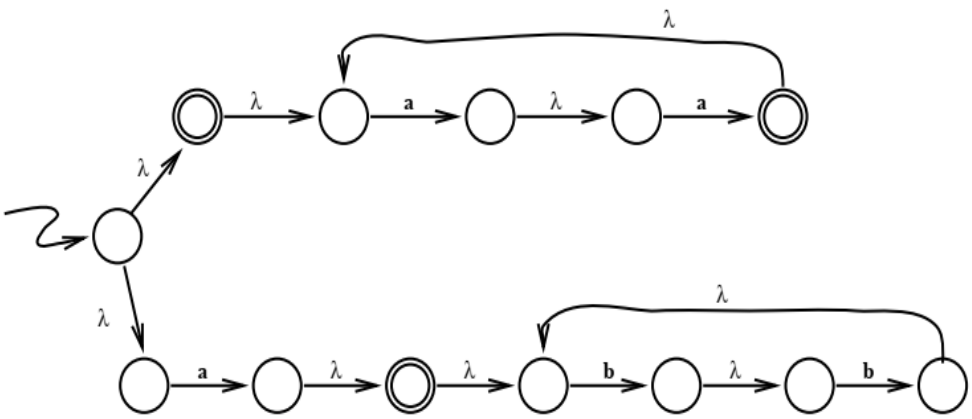


Figure 2.30: The machine $M_{(aa)^\star \cup (a(bb)^\star)}$

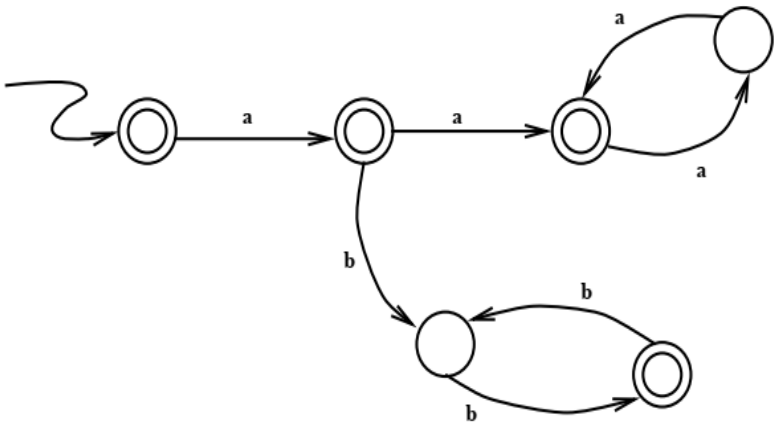


Figure 2.31: A deterministic version of $M_{(aa)^\star \cup (a(bb)^\star)}$

Nondeterministic machine with λ-transitions can be turned into deterministic ones with the same set-of-states approach used earlier. For example, the machine in Figure 2.30 turns into the machine in Figure 2.31.

This concludes the proof of Kleene's Theorem. □

Finite-State Machines or Regular Expressions?

Given that finite-state machines and regular expressions are equivalent concepts, we can choose to approach some problem in terms of one or the other. Some questions are easier to deal with in terms of finite-state machines, others in terms of regular expressions. Here are some examples, all of them variations of the question of which languages are regular:

Question 1. Is the set of binary numbers that are divisible by 2 regular?

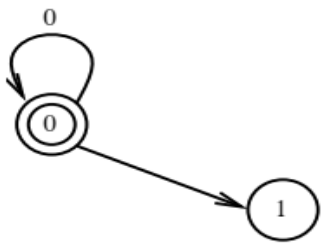


Figure 2.32: A piece of a finite-state machine

This is easy in either domain. The right regular expression is $(0 \cup 1)^*0$.

Question 2. Is the set of binary numbers that are divisible by 3 regular?

The answer is yes, but it seems nearly impossible to build a suitable regular expression. After all, what is the common pattern of 11, 110, 10000000010, 1001? Yet it is not all that hard to build a finite-state machine for this language, with the right approach anyway. The Myhill-Nerode Theorem tells us that we would have to figure out exactly what it is that we want to keep track of as the input gets processed.

If we know that the input so far was a number that was divisible by 3, then we know that after reading one more 0, the number is again divisible by 3. (This is so because reading that 0 amounted to multiplying the previous number by 2.) What if a 1 had been read instead? This would have amounted to multiplying the previous number by 2 and adding 1. The resulting number, when divided by 3, would yield a remainder of 1. This translates into a piece of a finite-state machine shown in Figure 2.32. The complete machine is shown in Figure 2.33. The four equivalence classes of $\equiv_L$ (and hence the states of the minimal machine for L) are $\{\lambda\}$ and the three sets of binary numbers which, when divided by three, yield a remainder of 0, 1, and 2. These remainders are used as the labels of the three states.

Question 3. Are regular languages closed under union? I.e., if A and B are regular, is their union $A \cup B$ regular as well?

The answer is yes. It follows from the definition of regular sets and expressions. In terms of finite-state machines, this requires an argument involving nondeterministic machine (as used in the proof of Kleene's Theorem).

Question 4. Are regular languages closed under complement? I.e., if A is regular, is $\overline{A} = \{x : x \notin A\}$ regular as well?

This is most easily dealt with in terms of finite-state machines. If M is a finite-state machine for A then a finite-state machine for $\overline{A}$ can be obtained by turning accepting states into rejecting ones, and vice versa. There is no equally simple argument in terms of regular expressions.

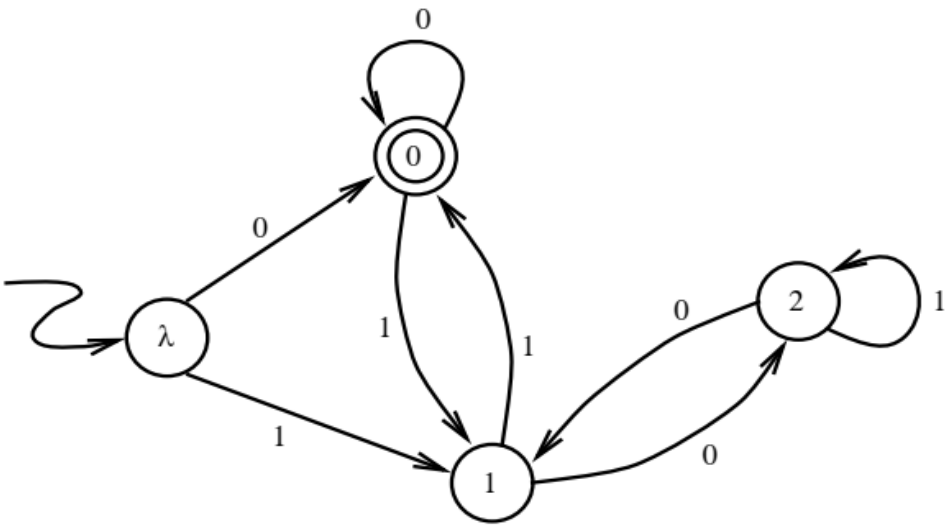


Figure 2.33: A finite-state machine for binary numbers divisible by 3

Question 5. Are regular languages closed under set difference? I.e., if A and B are regular, is their difference $A - B = \{x : x \in A \text{ and } x \notin B\}$ regular as well?

There is a construction based on finite-state machine that proves this closure result and is interesting in its own right.

Question 6. Is the set of syntactically correct C programs regular?

It is not, and the only proof method presented so far is to argue that there are infinitely many equivalence classes induced by $\equiv_L$. Here is an infinite list of nonequivalent strings:

```
main () { int x; x = (
main () { int x; x = ((
main () { int x; x = (((
main () { int x; x = ((((
main () { int x; x = (((((
...
```

This shows that parsing programs, which is a necessary part of compiling them, cannot be done with an amount of memory that does not depend on the input that is being parsed. As we will see in the next chapter, it takes a "stack" to parse.