4. More on FSAs: Limitations and closure properties

1 Interchangeable subexpressions

Now forget about FSAs for a moment, and just consider sets of strings "out of the blue". We'll connect things back to FSAs shortly, in section 2.

(1) Given some stringset $L \subseteq \Sigma^*$, the L-remainders of a string u are all the strings v such that $uv \in L$. I'll write $\text{rem}_L(u)$ for the set of L-remainders of u, so we can write this definition in symbols as: $\text{rem}_L(u) = \{v \mid v \in \Sigma^*, uv \in L\}$.

This definition may seem unfamiliar and a bit awkward, but the underlying idea is quite simple and *very powerful* — you will probably recognize that it's been hiding somewhere inside your existing understanding of how grammars (of any sort!) work. Roughly, $\text{rem}_L(u)$ gives us a handle on "all the things we're still allowed to do, if we've done u so far". Some examples will clarify.

(2) If $L_1 = \{\text{cat}, \text{cap}, \text{cape}, \text{cut}, \text{cup}, \text{dog}\}$, then:
   a. $\text{rem}_{L_1}(\text{ca}) = \{\text{t}, \text{p}, \text{pe}\}$
   b. $\text{rem}_{L_1}(\text{c}) = \{\text{at}, \text{ap}, \text{ape}, \text{ut}, \text{up}\}$
   c. $\text{rem}_{L_1}(\text{cap}) = \{\epsilon, \text{e}\}$
   d. $\text{rem}_{L_1}(\text{d}) = \{\text{og}\}$

(3) If $L_2 = \{\text{ad}, \text{add}, \text{baa}, \text{bad}, \text{cab}, \text{cad}, \text{dab}, \text{dad}\}$, then:
   a. $\text{rem}_{L_2}(\text{c}) = \text{rem}_{L_2}(\text{d}) = \{\text{ab}, \text{ad}\}$
   b. $\text{rem}_{L_2}(\text{a}) = \{\text{d}, \text{dd}\}$
   c. $\text{rem}_{L_2}(\text{da}) = \{\text{b}, \text{d}\}$
   d. $\text{rem}_{L_2}(\text{ad}) = \{\epsilon, \text{d}\}$

When we notice that $\text{rem}_{L_2}(\text{c}) = \text{rem}_{L_2}(\text{d})$, this tells us something useful about how we can go about designing a grammar to generate the stringset L_2: such a grammar *doesn't need to care about* the distinction between starting with 'c' and starting with 'd', because for any string v that you choose, cv and dv will either both be in L_2 or both not be in L_2. An initial 'c' and an initial 'd' are *interchangeable subexpressions*.

(4) Given a stringset $L \subseteq \Sigma^*$ and two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, we define a relation $\equiv_L$ such that: $u \equiv_L v$ iff $\text{rem}_L(u) = \text{rem}_L(v)$.

Some slightly more linguistics-ish examples:

(5) Suppose that $\Sigma = \{\text{C}, \text{V}\}$, and L is the subset of Σ^* containing all strings that contain at least one 'V'. Then:
   a. C $\equiv_L$ CC, because both can only be followed by strings that fulfill the requirement for a 'V'.
   b. VC $\equiv_L$ CV, because both can be followed by anything at all.
   c. So two strings are L-equivalent iff they either both do or both don't contain a 'V'.

(6) Suppose that $\Sigma = \{C, V\}$, and L is the subset of Σ^* containing all strings that have two adjacent 'C's or two adjacent 'V's (or both). Then
   a. C $\equiv_L$ CVC $\equiv_L$ CVCVC, because these all require remainders that have two adjacent 'C's *or* two adjacent 'V's *or* an initial 'C'.
   b. V $\equiv_L$ VCV $\equiv_L$ VCVCV, because these all require remainders that have two adjacent 'C's *or* two adjacent 'V's *or* an initial 'V'.
   c. CC $\equiv_L$ VCVCVVCVC

(7) Suppose that $\Sigma = \{C, V\}$, and L is the subset of Σ^* containing all strings that *do not* have two adjacent occurrences of 'V'. Then:
   a. CCCC $\equiv_L$ VC, because both can be followed by anything without adjacent 'V's.
   b. CCV $\equiv_L$ V, because both can be followed by anything without adjacent 'V's that does not begin with 'V'.
   c. CCV $\not\equiv_L$ CCC, because only the latter can be followed by 'VC'.
   d. In fact: two strings are L-equivalent iff they end with the same symbol!

(8) Suppose that Σ is the set of English words, and L is the set of all grammatical English word-sequences. Then (probably?):
   a. John $\equiv_L$ the brown furry rabbit
   b. John $\equiv_L$ Mary thinks that John
   c. John $\not\equiv_L$ the fact that John

2 The Myhill-Nerode Theorem

We can connect this idea of equivalent subexpressions back to *forward values* in an FSA. Recall that $\text{fwd}_M(u)(q)$ is a boolean, true or false; so we can think of $\text{fwd}_M(u)$ as a *set of states*, namely all those states reachable from an initial state of M by taking transitions that produce the string u. I'll sometimes call this a *forward set* (for lack of a better name).

Now here's the important connection:

(9) For any FSA $M = (Q, \Sigma, I, F, \Delta)$ and for any two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, if $\text{fwd}_M(u) = \text{fwd}_M(v)$ then $u \equiv_{\mathcal{L}(M)} v$.

Given any particular stringset L, we can think of the relation $\equiv_L$ as sorting out all possible strings into buckets (or "equivalence classes"): two strings belong in the same bucket iff they are equivalent prefixes. So what (9) says is that for an FSA to generate L it must be arranged so that fwd only maps two strings to the same state-sets if those two strings are equivalent prefixes; the machine can ignore distinctions between bucket-mates, but only between bucket-mates.

This idea of ignoring at least some distinctions is exactly what makes a grammar different from a list of strings. *The challenge in writing grammars is always about ignoring the "irrelevant" distinctions in order to allow creativity, while tracking the "relevant" distinctions.*[1]

And now we can put our finger on the capacities/limitations of finite-state automata.

(10) **The Myhill-Nerode Theorem:** Given a particular stringset L, there is an FSA that generates L iff the relation $\equiv_L$ sorts strings into only finitely-many buckets.

Why is this, exactly?

[1] McCulloch & Pitts (1943, pp.130–131) put this nicely in their classic paper (you should go and read it!): "our knowledge of the world including ourselves, is incomplete . . . This ignorance, implicit in all our brains, is the counterpart of abstraction which renders our knowledge useful." See also chapters 2–4 of Minsky's book *Computation: Finite and Infinite Machines* (1967) for good discussion of this very general point.

2.1 Why do FSAs make only finitely-many distinctions?

Well, if we have a particular FSA whose set of states is Q, then there are only finitely many distinct subsets of Q that fwd_M can map strings to; specifically, there are $2^{|Q|}$ of them. So there are only finitely-many "candidate forward sets", meaning that the FSA is necessarily making only those finitely-many distinctions.

Having noticed this, it's very easy to convince ourselves that no FSA can generate the stringset $L = \{a^n b^n \mid n > 0\}$. Notice that a $\not\equiv_L$ aa, and aa $\not\equiv_L$ aaa, and so on. In fact any string of 'a's is non-equivalent to each of the different-length strings of 'a's, so this stringset sorts strings into infinitely-many buckets, one bucket for each length. There is no way for an FSA M to be set up such that $\text{fwd}_M(a^j) \neq \text{fwd}_M(a^k)$ whenever $j \neq k$; any FSA will incorrectly collapse the distinction between two such strings of 'a's.

2.2 Why can any finitely-many distinctions be captured with an FSA?

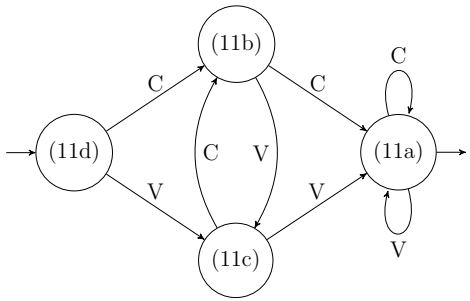
On the other hand, if we have a particular stringset L whose equivalence relation $\equiv_L$ makes only finitely-many distinctions, then there is a straightforward way to construct a minimal FSA whose states track exactly those distinctions.

Consider again the stringset from (6), consisting of all strings with either two adjacent 'C's or two adjacent 'V's (or both). This stringset's equivalence relation sorts strings into four buckets:

(11) a. a bucket containing strings that have either two adjacent 'C's or two adjacent 'V's;
b. a bucket containing strings that don't have two adjacent 'C's or 'V's, but end in 'C';
c. a bucket containing strings that don't have two adjacent 'C's or 'V's, but end in 'V';
d. a bucket containing only the empty string.

Having noticed this we can mechanically construct an appropriate FSA — known as the *minimal FSA* for this stringset — which has one state corresponding to each bucket. The crucial idea here is that if $u \equiv_L v$ then $ux \equiv_L vx$ for any $x \in \Sigma$, i.e. adding a symbol at the end can't "break" an equivalence; and similarly, for any FSA M, if $\text{fwd}_M(u) = \text{fwd}_M(v)$ then $\text{fwd}_M(ux) = \text{fwd}_M(vx)$.

(12)



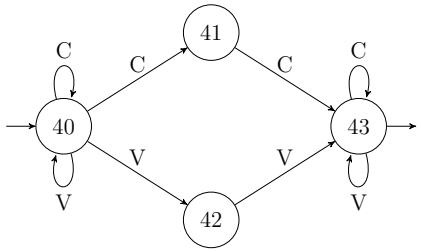
This strategy produces a specific kind of automaton, a *deterministic automaton*. In a deterministic automaton, there are never two arcs leading out of the same state that are labeled with the same symbol; each string corresponds to at most one path through the states, and so fwd only ever produces singleton sets or the empty set.

2.3 One loose end: determinization and minimization

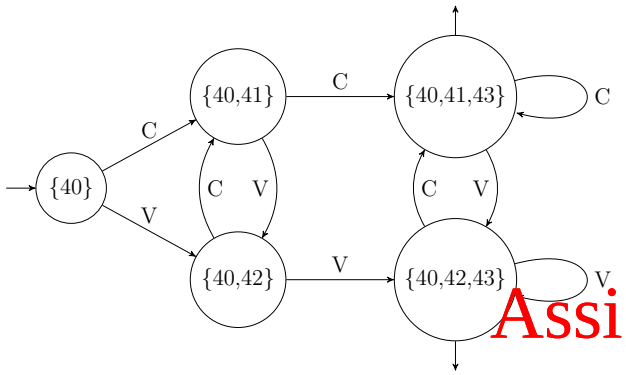
Any FSA M can be converted into an equivalent deterministic one M', by setting up the states of M' to correspond to sets of the states of M. Then, for any string u, the one state in $\text{fwd}_{M'}(u)$ will be the one corresponding to the set $\text{fwd}_M(u)$. Working out the transitions of the new automaton M' for a symbol x amounts to working out how to calculate $\text{fwd}_M(ux)$ from $\text{fwd}_M(u)$, which we saw last week.

Applying this procedure to the FSA in (13), which generates the stringset in (6), produces the new FSA in (14).

(13)



(14)

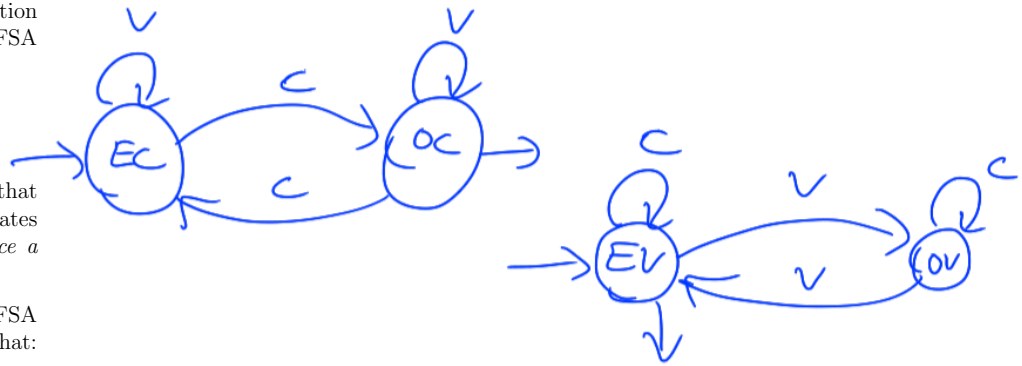


To see the connection, here's the table of forward values for the string 'CVCCV' using the FSA in (13).

(15)

State	C	V	C	C	V	
40	1	1	1	1	1	1
41	0	1	0	1	1	0
42	0	0	1	0	0	1
43	0	0	0	0	1	1

It turns out that the two states $\{40, 41, 43\}$ and $\{40, 42, 43\}$ in (14) "do the same thing": in either case, it's possible to end, it's possible to transition to state $\{40, 41, 43\}$ on a 'C', and it's possible to transition to state $\{40, 42, 43\}$ on a 'V' (and that's all). Collapsing these two states will produce the minimal FSA in (12).



CVCCV

3 Intersection of FSAs

The stringsets generated by FSAs are *closed under intersection*. This means that, if there is an FSA that generates the stringset L and there is an FSA that generates L', then there is also an FSA that generates the stringset $L \cap L'$. In other words, *intersecting two FSA-describable stringsets will never produce a non-FSA-describable stringset.*

One way to convince ourselves of this is to see that, given any two FSAs, we can construct a third FSA that generates the intersection of the original two FSAs' stringsets. The important ideas here will be that:

(16) the states of the new FSA are pairs, consisting of a state from the first FSA and a state from the second; and

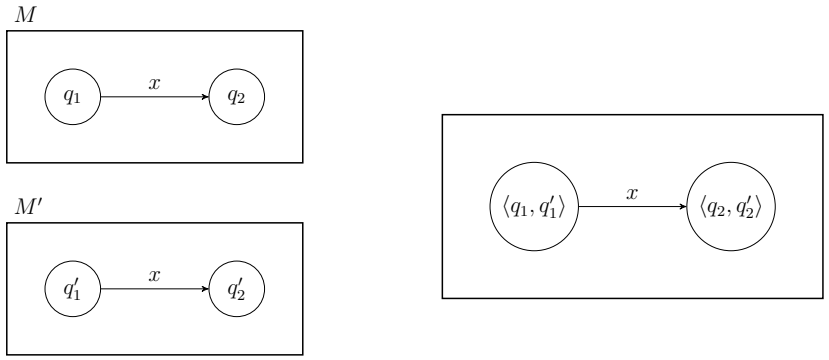
(17) each transition emitting a symbol x in the new FSA must connect two states which
- have as their first components states connected by an x-emitting transition in the first FSA, and

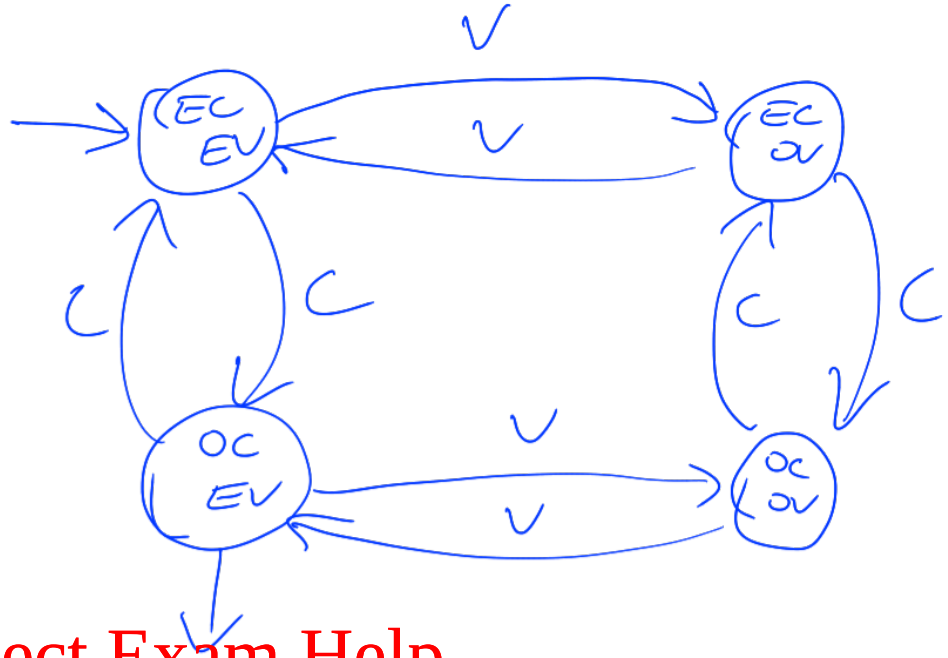
- have as their second components states connected by an x-emitting transition in the second FSA.

The new FSA will "simulate" the workings of both of the two original FSAs simultaneously.

Here's the picture to have in mind:

(18)





Here's the general "recipe":

(19) Given two FSAs $M = (Q, \Sigma, I, F, \Delta)$ and $M' = (Q', \Sigma, I', F', \Delta')$, the FSA $M'' = (Q \times Q', \Sigma, I \times I', F \times F', \Delta'')$ will generate $\mathcal{L}(M) \cap \mathcal{L}(M')$, where the new transition set Δ'' is defined by:

- $(\langle q_1, q'_1 \rangle, x, \langle q_2, q'_2 \rangle) \in \Delta''$ iff both $(q_1, x, q_2) \in \Delta$ and $(q'_1, x, q'_2) \in \Delta'$

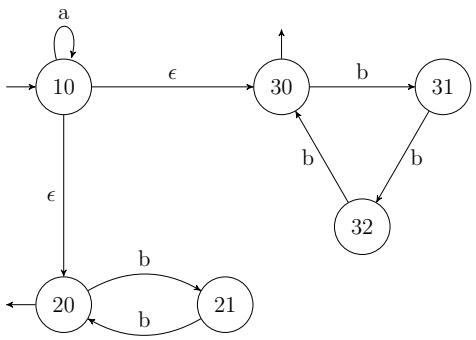
4 Finite-state automata with epsilon transitions

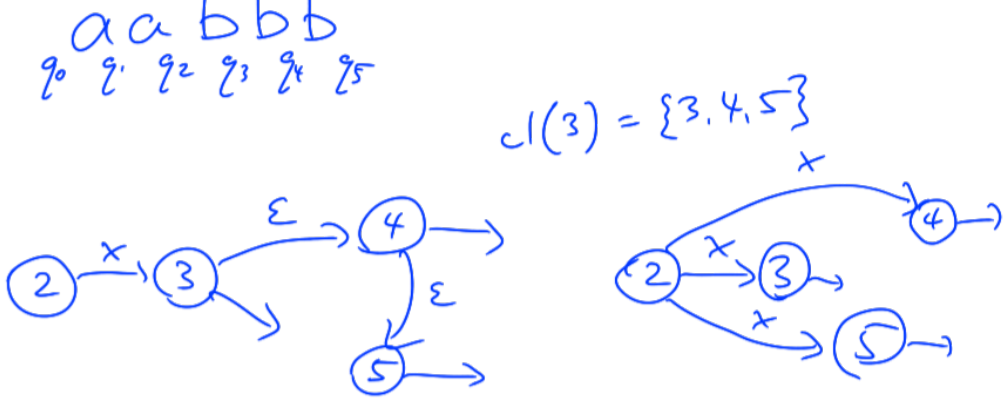
(20) A finite-state automaton with epsilon transitions (ϵ-FSA) is a five-tuple $(Q, \Sigma, I, F, \Delta)$ where:

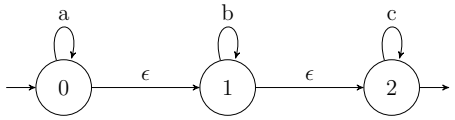
- Q is a finite set of states;
- Σ, the alphabet, is a finite set of symbols;
- $I \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of ending states; and
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the set of transitions.

The difference from what we've seen before is that transitions are labeled not with an element of Σ, but rather with *either* an element of Σ *or* ϵ. (Note that ϵ is a member of the set Σ^*, but it is not a member of Σ.)

(21)





(22) 

The key idea in dealing with epsilon transitions is the "epsilon-closure" of a state: in the automaton in (21), the epsilon-closure of state 10, which we'll write as $\text{cl}_\Delta(10)$, is the set $\{10, 20, 30\}$; and in the automaton in (22), $\text{cl}_\Delta(0) = \{0, 1, 2\}$ and $\text{cl}_\Delta(1) = \{1, 2\}$.

(23) If Δ is the transition function of some ϵ-FSA, then $\text{cl}_\Delta(q)$ is the set of all states reachable from q by a sequence of zero-or-more ϵ-transitions according to Δ.

It turns out that, given some ϵ-FSA that contains epsilon transitions, we can always construct another FSA that does not contain any epsilon transitions but generates exactly the same stringset. So the ability to include epsilon transitions doesn't bring any "extra expressive power". Allowing epsilon transitions does sometimes make automata easier to design, however.

(24) Given an ϵ-FSA $M = (Q, \Sigma, I, F, \Delta)$ (which may contain epsilon transitions), we can construct an FSA $M' = (Q, \Sigma, I, F', \Delta')$ (which does not contain epsilon transitions) that will generate the same stringset as M as follows:

- The new set of end states, F', contains all states q such that $\text{cl}_\Delta(q) \cap F \neq \varnothing$.
- The new transition table Δ' contains a transition (q_1, x, q_3) if there is some $q_2 \in \text{cl}_\Delta(q_1)$ such that $(q_2, x, q_3) \in \Delta$.

5 Regular expressions

5.1 Defining regular expressions and their denotations

First we'll define what regular expressions *are*, i.e. what counts as a regular expression. That's all we're saying in (25).

(25) Given an alphabet Σ, we define $\text{RE}(\Sigma)$, the set of regular expressions over Σ, as follows:

- if $x \in \Sigma$, then $\underline{x} \in \text{RE}(\Sigma)$
- if $r_1 \in \text{RE}(\Sigma)$ and $r_2 \in \text{RE}(\Sigma)$, then $(r_1 \mid r_2) \in \text{RE}(\Sigma)$
- if $r_1 \in \text{RE}(\Sigma)$ and $r_2 \in \text{RE}(\Sigma)$, then $(r_1 \cdot r_2) \in \text{RE}(\Sigma)$
- if $r \in \text{RE}(\Sigma)$, then $r^\star \in \text{RE}(\Sigma)$
- $\mathbf{0} \in \text{RE}(\Sigma)$
- $\mathbf{1} \in \text{RE}(\Sigma)$

So if we have the alphabet $\Sigma = \{\text{a}, \text{b}, \text{c}\}$, then here are some elements of $\text{RE}(\Sigma)$:

(26) a. $(\underline{\text{a}} \mid \underline{\text{b}})$
b. $((\underline{\text{a}} \mid \underline{\text{b}}) \cdot \underline{\text{c}})$
c. $((\underline{\text{a}} \mid \underline{\text{b}}) \cdot \underline{\text{c}})^\star$

Now, any regular expression $r \in \text{RE}(\Sigma)$ denotes a particular subset of Σ^*, i.e. denotes a stringset. We'll write $[\![r]\!]$ for the stringset denoted by r.

(27) Given a regular expression $r \in \text{RE}(\Sigma)$, we define the set $[\![r]\!] \subseteq \Sigma^*$ as follows:

a. $[\![\underline{x}]\!] = \{x\}$
b. $[\![(r_1 \mid r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$
c. $[\![(r_1 \cdot r_2)]\!] = \{u +\!\!+ v \mid u \in [\![r_1]\!], v \in [\![r_2]\!]\}$

liftA2 (\u -> \v -> u ++ v) [[r1]] [[r2]]

(T ∧ F) ∨ ¬F

[[(T ∧ F) ∨ ¬F]]

d. $[\![r^\star]\!]$ is the smallest set such that:
   - $\epsilon \in [\![r^\star]\!]$
   - if $u \in [\![r]\!]$ and $v \in [\![r^\star]\!]$, then $u +\!\!+ v \in [\![r^\star]\!]$

e. $[\![\mathbf{0}]\!] = \varnothing = \{\}$

f. $[\![\mathbf{1}]\!] = \{\epsilon\}$

The tricky part here is the $r^\star$ case. It says roughly that $[\![r^\star]\!]$ is the set comprising all strings that we can get by concatenating zero or more strings from the set $[\![r]\!]$. Concatenating *zero* such strings produces ϵ, so $\epsilon \in [\![r^\star]\!]$. Concatenating n such strings, where n is *non-zero*, really means concatenating some string u, which is in $[\![r]\!]$, with some string v, which is the result of concatenating some $n-1$ such strings.

We can use this definition to work out the stringsets denoted by the regular expressions in (26).

(28) a. $$\begin{aligned}[\![(\underline{a} \mid \underline{b})]\!] &= [\![\underline{a}]\!] \cup [\![\underline{b}]\!] \\ &= \{a\} \cup \{b\} \\ &= \{a, b\}\end{aligned}$$

b. $$\begin{aligned}[\![((\underline{a} \mid \underline{b}) \cdot \underline{c})]\!] &= \{u +\!\!+ v \mid u \in [\![(\underline{a} \mid \underline{b})]\!], v \in [\![\underline{c}]\!]\} \\ &= \{u +\!\!+ v \mid u \in \{a, b\}, v \in \{c\}\} \\ &= \{a +\!\!+ c, b +\!\!+ c\} \\ &= \{ac, bc\}\end{aligned}$$

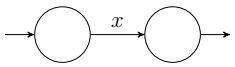
c. $[\![((\underline{a} \mid \underline{b}) \cdot \underline{c})^\star]\!] = \{\epsilon, ac, bc, acac, acbc, bcac, bcbc, acacac, acacbc, \ldots\}$

5.2 Relating regular expressions to FSAs

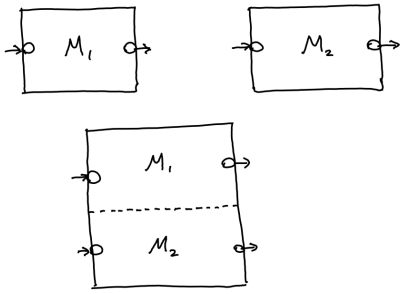
It turns out that given any regular expression r, we can construct an ϵ-FSA M such that $L(M) = [\![r]\!]$, i.e. we can construct an ϵ-FSA that generates exactly the stringset denoted by r.

To do this we need to proceed recursively on the structure of the regular expression — because there are unboundedly many regular expressions to deal with. The diagrams below give the important ideas for how this works, but to avoid clutter assume unique starting states and ending states once you understand what's going on, it's relatively easy to generalize to multiple starting and/or ending states.

(29) a. We can construct an ϵ-FSA that generates the stringset $[\![\underline{x}]\!]$, for any $x \in \Sigma$, as follows:



b. We can construct an ϵ-FSA that generates the stringset $[\![(r_1 \mid r_2)]\!]$, if we are given an ϵ-FSA M_1 that generates $[\![r_1]\!]$ and an ϵ-FSA M_2 that generates $[\![r_2]\!]$, as follows:



NB: This will usually produce an automaton that has multiple start states and multiple end states.

c. We can construct an ϵ-FSA that generates the stringset $[\![(r_1 \cdot r_2)]\!]$, if we are given an ϵ-FSA M_1

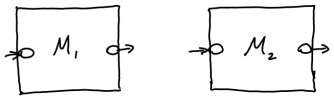
RE a → FSA Int a

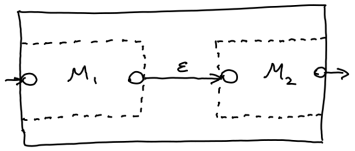
case r of

Lit a →

Alt r_1 r_2 → ... (convert r_1)

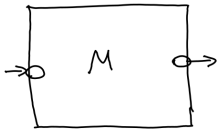
that generates $[\![r_1]\!]$ and an ϵ-FSA M_2 that generates $[\![r_2]\!]$, as follows:

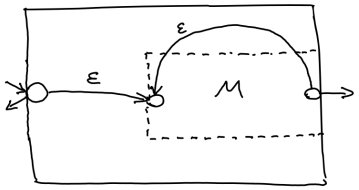




NB: This requires an ϵ-transition linking *every* ending state of M_1 to *every* starting state of M_2.

d. We can construct an ϵ-FSA that generates the stringset $[\![r^\star]\!]$, if we are given an ϵ-FSA M that generates $[\![r]\!]$, as follows:

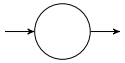




NB: This requires an ϵ-transition linking *every* ending state of M to *every* starting state of M, and an ϵ-transition linking the new state to *every* starting state of M.

e. We can construct an ϵ-FSA that generates the stringset $[\![\mathbf{0}]\!]$ as follows:

f. We can construct an ϵ-FSA that generates the stringset $[\![\mathbf{1}]\!]$ as follows:



{ε}

6 Summing up: Regular languages

A stringset $S \subseteq \Sigma^*$ is a *regular stringset* (or *regular language*) iff there is some regular expression $r \in \mathrm{RE}(\Sigma)$ such that $[\![r]\!] = S$.

We've just seen that, given any regular expression r, we can construct an ϵ-FSA M such that $\mathcal{L}(M) = [\![r]\!]$. This tells us that any stringset that can be described by a regular expression — any regular language — can also be described by an ϵ-FSA.

It turns out — surprisingly, perhaps — that we can also go the other way: given an ϵ-FSA M, there is a method for constructing a regular expression r such that $[\![r]\!] = \mathcal{L}(M)$.[2] So any stringset that can be described by an ϵ-FSA is also a regular language.

[2] This is a bit trickier to prove, but see e.g. pages 33–34 of Hopcroft & Ullman (1979), or pages 69–74 of Sipser (1997).

So for any stringset S, either all of the following are true or all are false:

(30)
a. S is a regular language.
b. There is some regular expression r such that $[\![r]\!] = S$.
c. There is some ϵ-FSA M such that $\mathcal{L}(M) = S$.
d. There is some FSA M such that $\mathcal{L}(M) = S$.
e. The number of distinct classes of S-equivalent prefixes, i.e. the number of buckets into which strings are sorted by the relation $\equiv_S$, is finite.
f. The number of distinct classes of S-equivalent suffixes is finite.[3]

For more on these topics, two good textbooks are:
- Hopcroft & Ullman 1979, *Introduction to Automata Theory, Languages and Computation*, chapters 2 and 3.
- Sipser 1997, *Introduction to the Theory of Computation*, chapter 1.

[3]This is just the obvious parallel to the idea of S-equivalent prefixes: two strings u and v are S-equivalent suffixes iff, for all strings w, wu and wv are either both in S or both not in S.