Tuesday – August 13, 2024

Regular languages

1 Interchangeable subexpressions

Let's forget about FSAs for a moment, and just consider sets of strings "out of the blue." We'll connect things back to FSAs shortly. Here's a useful concept:

(1) **L-remainders**

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

Roughly, $rem_L(u)$ gives us a handle on "all the things we're still allowed to do, if we've done u so far."

(2) If L_1 = {cat, cap, cape, cut, cup, dog}, then:

a. rem_{L_1}(ca) = {t, p, pe}
b. rem_{L_1}(c) = {at, ap, ape, ut, up}
c. rem_{L_1}(cap) = {ε, e}
empty string

ex: {cv, cvv}
$rem_L(c) = \{v, vv\}$

(3) If L_2 = {ad, add, baa, bad, cab, cad, dab, dad}, then:

a. rem_{L_2}(c) = {ab, ad}
b. rem_{L_2}(d) = {ab, ad}
c. rem_{L_2}(a) = {d, dd}
d. rem_{L_2}(da) = {b, d}
e. rem_{L_2}(ad) = {ε, d}
empty string

When we notice that rem_{L_2}(c) = rem_{L_2}(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.' This is 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 thus **interchangeable subexpressions**.

(4) **L-equivalence**: Given a stringset L and two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, we define a relation $\equiv_L$ such that: $u \equiv_L v$ iff $rem_L(u) = rem_L(v)$.

"(c) and (d) L equivalent since they have same L remainder"

Some slightly more linguistics-ish examples:

alphabet

(5) Suppose that $\Sigma = \{C, V\}$, and L contains all strings that contain at least one 'V.' Then:

L equivalent

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. 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 contains 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'. create CC
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'. create VV
c. $CC \equiv_L VCVCVVCVC$ both satisfy requirement already

L remainder is any string in the alphabet for this ex

(7) Suppose that $\Sigma = \{C, V\}$, and L contains 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'. ← don't want VV
c. $CCV \not\equiv_L CCC$, because only the latter can be followed by 'VC.'
d. So, two strings are L-equivalent if they end with the same symbol.

alphabet

(8) Suppose that Σ is the set of English words, and L is the set of all grammatical English-word sequences. Then (at least from the perspective of the syntax):

a. John $\equiv_L$ the brown furry rabbit you can substitute these in to any english sentence
b. John $\equiv_L$ Mary thinks that John
c. John $\not\equiv_L$ the fact that John

"we can swap them"

2 The Myhill-Nerode Theorem

prefix or suffix
state

We can connect this idea of equivalent subexpressions back to forward values in an FSA. Recall that $fwd_M(u)(q)$ is a Boolean, True or False; so we can think of $fwd_M(u)$ 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 $fwd_M(u) = fwd_M(v)$, then $u \equiv_{L(M)} v$.

both strings L equivalent if have same forward set

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.

- According to (9), 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.

And now we can put our finger on the capacities and 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.

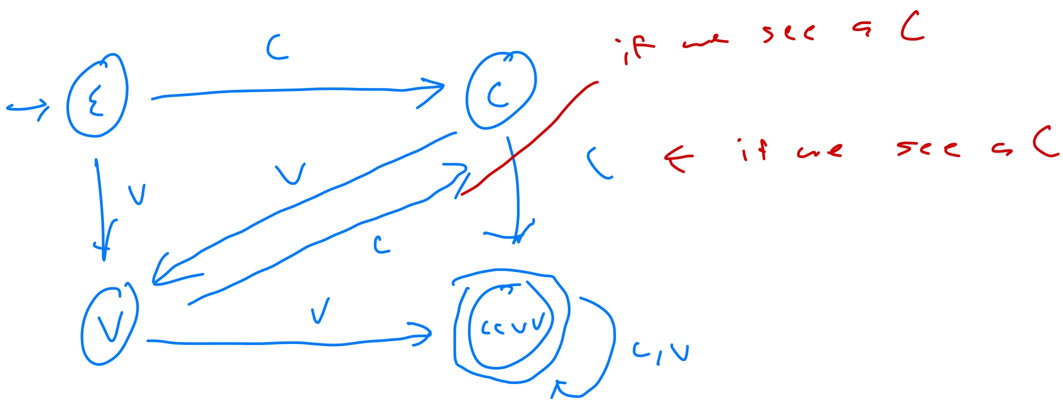
2.1 Why can any finitely-many distinctions be captured by an FSA?

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 (~~5~~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.



if two strings r equivalent,

The crucial idea here is that if $u \equiv_L v$, then $ux \equiv_L vx$ for any $x \in \Sigma$. Adding a symbol at the end can't "break" an equivalence. Similarly, for any FSA M, if $fwd_M(u) = fwd_M(v)$, then $fwd_M(ux) = fwd_M(vx)$.

finite states = finite distinctions

if two strings forward set

2.2 Why do FSAs only make finitely-many distinctions?

On the other hand, 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, namely $2^{|Q|}$ of them. This means that there are only finitely-many "candidate forward sets" for any string, so the FSA is necessarily making only those finitely-many distinctions between strings.

n partitions of a

To illustrate this concept, consider the string set $L = \{a^n b^n \mid n > 0\}$, which no FSA is able to generate. Notice that a $\not\equiv_L$ aa, and aa $\not\equiv_L$ aaa, and so on.

not regular

- In fact, any string of 'a's is non-equivalent to any other different-length string of 'a's, so this stringset sorts strings into infinitely-many buckets, one bucket for each length.
- But there is no way for an FSA M to be set up such that $fwd_M(a^j) \not\equiv_L fwd_M(a^k)$ whenever $j \neq k$; any FSA will incorrectly collapse the distinction between two such strings of 'a's.

Regular language = any language captured by FSA

3 FSAs with epsilon transitions

Epsilon transitions provide another useful way to extend standard FSAs:

states / alphabet / initial / final

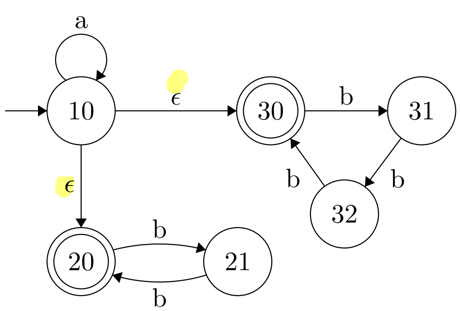
(12) A finite-state automaton with epsilon transitions (ϵ-FSA) is a five-tuple $(Q, \Sigma, I, F, \Delta)$ where:

a. Q is a finite set of states;
b. Σ, the alphabet, is a finite set of symbols;
c. $I \subseteq Q$ is the set of initial states;
d. $F \subseteq Q$ is the set of ending states;
e. $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the set of transitions. transition over empty string

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* the empty string. (Note that ϵ is a member of the set Σ^*, but it is not a member of Σ.)

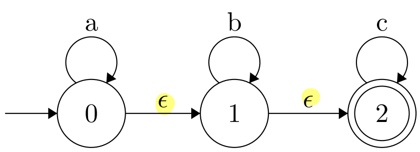
Here are two ϵ-FSAs for illustration:

(13)



epsilon closure of 10
{10, 20, 30}

(14)



epsilon closure of 0
{0, 1, 2}

The key idea in dealing with ϵ-transitions is the **epsilon closure** of a state: ex:

(15) If Δ is the transition function, $cl_\Delta(q)$ 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 an FSA with epsilon transitions can always be converted into another FSA that does not contain any epsilon transitions, but generates exactly the same stringset.

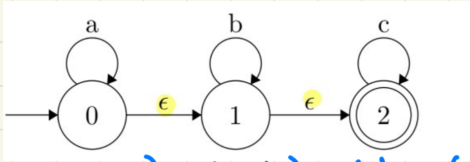
Here's the recipe for converting an ϵ-FSA into an FSA that does not contain epsilon-transitions:

(16) 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:

we

a. The new set of end states, F', contains all states q such that $cl_\Delta(q) \cap F \neq \varnothing$
b. The new transition set Δ' contains a transition (q_1, x, q_3) iff there is some $q_2 \in cl_\Delta(q_1)$ such that $(q_2, x, q_3) \in \Delta$

ex using 14



step a:

$cl_\Delta(0) = \{0,1,2\} \cap F = \{2\}$

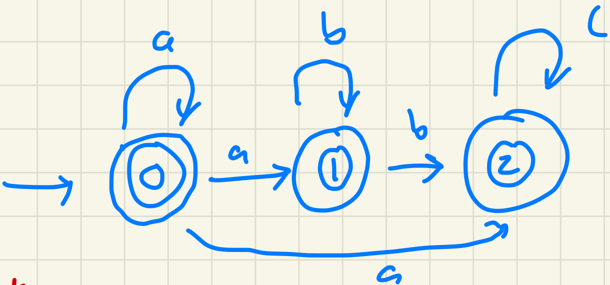
$cl_\Delta(1) = \{1,2\} \cap F = \{2\}$

$cl_\Delta(2) = \{2\} \cap F = \{2\}$

$F = \{2\}$

" look for final states

← we can create $a^x b^y c^z$



we can get from 0 to 1 and 2 by just using a

we can get from 1 to 2 just using b

4 Regular expressions and FSAs

4.1 Defining regular expressions and their denotations

First we'll define what regular expressions are. That's all we're saying in (17). It's analogous to defining what counts as a propositional formula.

{} Regular expression

(17) Given an alphabet Σ, we define RE(Σ), the set of regular expressions over Σ, as follows:

a. $\mathbf{0} \in \text{RE}(\Sigma)$

{ε} b. $\mathbf{1} \in \text{RE}(\Sigma)$

c. if $x \in \Sigma$, then $\underline{x} \in \text{RE}(\Sigma)$

or

d. if $r_1 \in \text{RE}(\Sigma)$ and $r_2 \in \text{RE}(\Sigma)$, then $(r_1 \mid r_2) \in \text{RE}(\Sigma)$

e. if $r_1 \in \text{RE}(\Sigma)$ and $r_2 \in \text{RE}(\Sigma)$, then $(r_1 \cdot r_2) \in \text{RE}(\Sigma)$

concatinated

f. if $r \in \text{RE}(\Sigma)$, then $r^* \in \text{RE}(\Sigma)$

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

- $(\underline{\text{a}} \mid \underline{\text{b}})$
- $((\underline{\text{a}} \mid \underline{\text{b}}) \cdot \underline{\text{c}})$ ← in haskell code example
- $((\underline{\text{a}} \mid \underline{\text{b}}) \cdot \underline{\text{c}})^*$

Now, any regular expression $r \in \text{RE}(\Sigma)$ denotes a stringset. We'll write $[\![r]\!]$ for the stringset denoted by r.

(18) Given a regular expression $r \in \text{RE}(\Sigma)$, $[\![r]\!] \subseteq \Sigma^*$ such that:

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

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

union concatenate

c. $[\![\underline{x}]\!] = \{x\}$

d. $[\![(r_1 \mid r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$

e. $[\![(r_1 \cdot r_2)]\!] = \{u +\!\!+ v \mid u \in [\![r_1]\!], v \in [\![r_2]\!]\}$

f. $[\![r^*]\!]$ is the smallest set such that:

- $\epsilon \in [\![r^*]\!]$
- if $u \in [\![r]\!]$ and $v \in [\![r^*]\!]$, then $u +\!\!+ v \in [\![r^*]\!]$

} zero or more replicas of r

The tricky part here is the r^* case. It says roughly that $[\![r^*]\!]$ is the set comprising all strings that we can get by concatenating zero or more strings from the set $[\![r]\!]$.

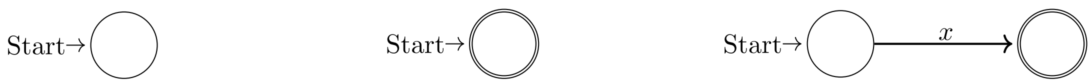
- Concatenating *zero* such strings produces ϵ, so $\epsilon \in [\![r^*]\!]$.
- Concatenating n such strings, where n is *non-zero*, really means concatenating some string u, which is in $[\![r]\!]$, with some string v, which is the result of concatenating some $n - 1$ such strings.

4.2 Relating regular expressions to FSAs

It turns out that given any regular expression r, we can construct an ϵ-FSA M such that $\mathcal{L}(M) = [\![r]\!]$. That is, we can construct an ϵ-FSA that generates exactly the stringset denoted by r. To do this, we have 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 idea for how this works.

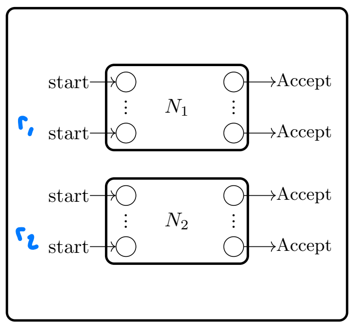
- ... $[\![\mathbf{0}]\!]$, $[\![\mathbf{1}]\!]$ and $[\![\underline{x}]\!]$, for any $x \in \Sigma$, as follows

also end state



(a) $R = \boldsymbol{0}$ **(b)** $R = \boldsymbol{1}$ **(c)** $R = \underline{x}$

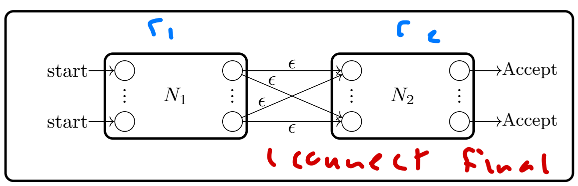
- ... $[\![(r_1 \mid r_2)]\!]$, given an ϵ-FSA N_1 that generates $[\![r_1]\!]$ and an ϵ-FSA N_2 that generates $[\![r_2]\!]$, as follows:



← New FSA has everything from both

Figure 2: *The construction used in the union of two ϵ-FSAs*

- ... $[\![(r_1 \cdot r_2)]\!]$, given an ϵ-FSA N_1 that generates $[\![r_1]\!]$ and an ϵ-FSA N_2 that generates $[\![r_2]\!]$, as follows.



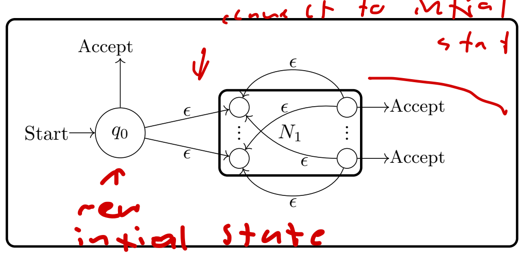
concatenated

connect final states of r_1 with starting states in r_2 using epsilon transitions

Figure 3: *The construction used in the concatenation of two ϵ-FSAs*

- ... $[\![r^*]\!]$, given an ϵ-FSA N_1 that generates $[\![r]\!]$.

epsilon connect to initial states



connect final to initial states using epsilon

Figure 4: *The construction used in the star operation*

5 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 $[\![r]\!] = S$.

language of FSA

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 that the inverse is also true: given any ϵ-FSA M, there is a method for constructing a regular expression r such that $[\![r]\!] = \mathcal{L}(M)$. So this means that any stringset that can be described by an ϵ-FSA is a regular language. [1]

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

(19)
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 equivalent prefixes, i.e. the number of buckets into which strings are sorted by the relation $\equiv_L$, is finite.
f. The number of distinct classes of equivalent suffixes is finite.[2]

one of things is true

[1] This is trickier to prove, but see e.g. pp. 33-34 Hopcroft and Ullman (1979) or pp. 67-74 of Sipser (1997).

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