Unit 9

Hidden Alphabets *or* The Horrors of Abstractness

1 Adding a Hidden Alphabet

1.1 Refined Strictly Local Grammars

Definition 9.1 (Run). Let Σ and Q be overt and *hidden* alphabets, respectively. Each $q \in Q$ is called a *state*. Given a string w over Σ, a (Σ, Q)-*run* over w is a total function mapping each node n of w to some $q \in Q$. We also say that q *is assigned to* n, and we represent the set of all (Σ, Q)-runs over w by Q^w. We may omit mention of Σ and Q if they are clear from the context. Overloading our terminology, we also use run to refer to the image of w under a given run, or the string of pairs $p_1 \cdots p_n$, where each p_i consists of the i-th node of w and its image under some fixed Q-run.

Example 9.1 Runs Over a Short String

Suppose that $Q := p, q, r$ and that $w := abbca$. Then the set of Q-runs over w includes, among others, $qpprq$, $pqrpq$, or even $ppppp$, but not $qppqs$ (because $s \notin Q$), qpp (too few symbols), or $pqrpqr$ (too many symbols). If we view runs as strings of pairs, then the three runs and non-runs look as below:

q	p	p	r	q		p	q	r	p	q		p	p	p	p	p	
a	b	b	c	a		a	b	b	c	a		a	b	b	c	a	
q	p	p	q	s		q	p	p				p	p	p	p	p	r
a	b	b	c	a		a	b	b	c	a		a	b	b	c	a	

Definition 9.2 (Refined Grammar). A refined strictly k-local grammar G is a finite set of k-grams over alphabet $\Sigma \times Q$. Such a grammar generates the language $L(G) := \{w \mid \exists r \in Q^w, k\text{-grams}(r) \subseteq G\}$. A language is refined strictly k-local iff it is generated by a refined strictly k-local grammar. The class of all refined strictly k-local languages is denoted SL_k^R.

Example 9.2 A Refined Grammar for Even Counting

Recall that the language $(aa)^+$, which contains all non-empty strings over a whose length is even, is neither strictly local nor strictly piecewise. However, it can be generated by a refined strictly 2-local grammar with only a handful of bigrams.

$$\frac{eo}{\rtimes a} \quad \frac{oe}{aa} \quad \frac{eo}{aa} \quad \frac{ee}{a \ltimes}$$

This grammar generates *aaaa*, for instance, because there is a run such that each bigram of that run is licensed by the grammar. In contrast, the slightly longer *aaaaa* is not generated because every run contains at least one bigram that is not licensed by the grammar. We indicate this by not assigning a state to the node for which an irreconcilable conflict arises.

e	o	e	o	e	e
⋊	a	a	a	a	⋉

e	o	e	o	e	o	
⋊	a	a	a	a	a	⋉

1.2 Generative Capacity

Due to how refined strictly local grammars are defined, they trivially subsume the strictly local grammars as a special case where the hidden alphabet Q contains only one state. The fact that a refined strictly 2-local grammar can generate (aa^+) shows that the inclusion is proper — refined strictly local grammars are strictly more powerful than strictly local grammars without a hidden alphabet of states. But this result can be even strengthened: ever strictly local language is refined strictly 2-local.

Lemma 9.3. $\mathrm{SL} \subsetneq \mathrm{SL}_2^R$ ⌟

To see this, just keep in mind that in order to determine the well-formedness of a string with respect to a strictly k-local grammar, it suffices to memorize all the k-grams in the string and see if all of them are licensed by the grammar. This was exactly the way our first scanner implementation operated. In a refined strictly local grammar, we can use the states to keep track of all the k-grams, and the only states that can be assigned to the right edge marker ⋉ are those that represent a subset of the strictly k-local grammar.

Example 9.3 Emulating a Grammar via a Hidden Alphabet

Consider the language $(aab)^+$, which is generated by the strictly 3-local grammar $G_3 := \{\rtimes \rtimes a, \rtimes aa, aab, aba, baa, ab\ltimes, b \ltimes \ltimes\}$. Then we can construct a refined strictly 2-local grammar G_2^R that uses its states to keep track of all trigrams seen so far, as well as the last two symbols preceding the current symbol (so that the three can be combined into a new trigram if necessary).

The grammar is bound to be large, since merely keeping track of all seen k-grams already requires $2^{|G_3|} = 2^7 = 128$ distinct states. There are also $|\Sigma| \cdot (k-1) =$

$2 \cdot (3-1) = 4$ distinct $(k-1)$-grams, and since the grammar has to keep track of both the seen k-grams and the previous $(k-1)$-gram, it must use a hidden alphabet with $4 * 128 = 512$ distinct states (although not all of them occur in some bigram).

Due to its sheer size, G_2^R is not listed here, but its basic functioning can be gleaned from a few example runs. Note in particular how the second string below is rejected as ill-formed because $\ltimes$ would receive the state $\langle aa, \{\rtimes \rtimes a, \rtimes aa, aa\ltimes\}\rangle$, but since $\{\rtimes \rtimes a, \rtimes aa, aa\ltimes\}$ is not a subset of G_3, no bigram of G_2^R uses this state.

$$\underset{\rtimes}{\begin{pmatrix}\rtimes\rtimes\\ \emptyset\end{pmatrix}} \quad \underset{a}{\begin{pmatrix}\rtimes\rtimes\\ \{\rtimes \rtimes a\}\end{pmatrix}} \quad \underset{a}{\begin{pmatrix}\rtimes a\\ \{\rtimes \rtimes a, \rtimes aa\}\end{pmatrix}} \quad \underset{b}{\begin{pmatrix}aa\\ \{\rtimes \rtimes a, \rtimes aa, aab\}\end{pmatrix}} \quad \underset{\ltimes}{\begin{pmatrix}ab\\ \{\rtimes \rtimes a, \rtimes aa, aab, ab\ltimes\}\end{pmatrix}}$$

$$\underset{\rtimes}{\begin{pmatrix}\rtimes\rtimes\\ \emptyset\end{pmatrix}} \quad \underset{a}{\begin{pmatrix}\rtimes\rtimes\\ \{\rtimes \rtimes a\}\end{pmatrix}} \quad \underset{a}{\begin{pmatrix}\rtimes a\\ \{\rtimes \rtimes a, \rtimes aa\}\end{pmatrix}} \quad \ltimes$$

Proof. We already have $\mathrm{SL}_2^R \not\subseteq \mathrm{SL}$ thanks to the example of $(aa)^+$, so it suffices to show $\mathrm{SL} \subseteq \mathrm{SL}_2^R$. Let $L \in SL_k$ be a language over alphabet Σ, and G a positive strictly k-local grammar such that $L(G) = L$. The grammar G_2^R is the largest subset of $(\Sigma \times Q)^2$ such that

- Q consists of pairs $\langle g, S\rangle$, where g is a $(k-1)$-gram over Σ and S a set of k-grams over Σ,
- $\frac{pq}{ab} \in G_2^R$ only if, for $p := \langle g_p, S_p\rangle$ and $q := \langle g_q, S_q\rangle$
  - if $a = \rtimes$, then $p := \langle \rtimes^{k-1}, \emptyset\rangle$,
  - $g_q := u \cdot a$, where $u \in \Sigma^{k-2}$ and $g_p := x \cdot u$,
  - $S_q := S_p \cup \{g_q \cdot b\}$,
  - $S_q \subseteq G$.

Since S_q must be a subset of G, it follows immediately that $L(G_2^R) \subseteq L(G)$. In the other direction, suppose that some $w \in L(G)$ is not contained in $L(G_2^R)$. Then there is no $r \in Q^w$ for which k-grams$(r) \subseteq G_2^R$. However, G_2^R is a maximal subset and thus contains every $\frac{pq}{ab}$ unless one of the conditions above is violated. But since $w \in L(G)$, k-grams$(w) \subseteq G$, so g_p, g_q and S_q are well-defined, and $S_q \subseteq G$. □

The strategy above can be modified to keep track of subsequences instead of substrings, which entails that every strictly piecewise language is refined strictly 2-local. And because $(aa)^+$ is not strictly piecewise but refined strictly 2-local, we get yet another proper subsumption relation.

Lemma 9.4. $\mathrm{SP} \subsetneq \mathrm{SL}_2^R$ ⌟

At this point it shouldn't come as a surprise that even the strict threshold testable languages are refined strictly 2-local.

Lemma 9.5. STT $\subsetneq \mathrm{SL}_2^R$ ⌟

Proof. Left as an exercise to the reader. □

As you can see, the addition of a hidden alphabet has made our formalism much more powerful, to the extent where we can't even make any distinctions between local and non-local dependencies: they all belong to SL_2^R. And as we will see next, things are even worse insofar as locality plays no role at all in refined strictly local grammars.

1.3 Reduction to 2-Locality

The subsumption results established in the previous section are worrying. By now we have seen a lot of evidence that SL_k and SP_k are feasible models for local and non-local phonological processes thanks to their curtailed yet sufficient expressivity, guaranteed learnability, low cognitive requirements, and closure properties that reflect specific typological properties of phonology. Crucially, though, these properties did not fully extend to the full classes SL and SP, which are not learnable in the limit from positive text and include all finite languages. The addition of a hidden vocabulary pushes even SL_2 to a level of power beyond even the **union** of SL and SP. Since the class of refined strictly 2-local languages properly subsumes both SL and SP, it inherits their shortcomings with respect to overgeneration and non-learnability — all of sudden, we have lost many attractive properties of our formal model, and we cannot even make a principled distinction between local and non-local processes anymore because both are in SL_2^R.

But things are even worse, because this loss of locality extends even to the full class of refined strictly local languages. Once we have a hidden alphabet, the size of the k-grams becomes largely irrelevant.

Theorem 9.6. If a language is generated by a refined strictly k-local grammar with hidden alphabet Q ($k \geq 2$) then it is generated by a refined strictly 2-local grammar with hidden alphabet Q'.

The proof for this theorem is fairly cumbersome to read, but builds directly on the insights of the translation from SL_k to SL_2^R: since a refined strictly k-local grammar has only a finite number of k-grams, we can emulate the computations of the whole grammar in the hidden alphabet of some refined strictly 2-local grammar.

Example 9.4 Emulating a Refined Grammar via a Hidden Alphabet

Let L be the language of all strings over $\{a, b\}$ such that consecutive substrings of bs must have even length and may only occur after at least three as. This language includes strings like *aaa*, *aaabb*, and *aaabbbba*, but not *bbaaa* or *aaab*. This can be easily handled by a refined strictly 4-local grammar where the hidden alphabet keeps track of the length requirement while the presence of three as is enforced directly via the 4-grams. The initial 4-grams are very simple, since no b can occur at the start of a string:

$$\frac{eeee}{\rtimes\rtimes\rtimes a} \qquad \frac{eeee}{\rtimes\rtimes aa} \qquad \frac{eeee}{\rtimes aaa}$$

The final 4-grams already have to take quite a bit of variation into account:

$$\frac{eeee}{a \ltimes \ltimes \ltimes} \quad \frac{eeee}{aa \ltimes \ltimes} \quad \frac{eeee}{aaa \ltimes} \quad \frac{eoee}{abb \ltimes}$$
$$\frac{eeee}{b \ltimes \ltimes \ltimes} \quad \frac{eeee}{ba \ltimes \ltimes} \quad \frac{eeee}{baa \ltimes}$$
$$\frac{oeee}{bb \ltimes \ltimes} \quad \frac{oeee}{bba \ltimes}$$
$$\frac{eoee}{bbb \ltimes}$$

The non-initial, non-final 4-grams are also much fewer than one might expect thanks to the restricted distribution of *b*s:

$$\frac{eeee}{aaaa} \quad \frac{eeeo}{aaab} \quad \frac{eeoe}{aabb} \quad \frac{eoee}{abba} \quad \frac{eoeo}{abbb}$$
$$\frac{eoeo}{bbbb} \quad \frac{oeoe}{bbbb} \quad \frac{eoee}{bbba} \quad \frac{oeee}{bbaa} \quad \frac{eeee}{baaa}$$

Finally, we also add 4-grams to allow for the empty string:

$$\frac{eeee}{\rtimes \rtimes \rtimes \ltimes} \quad \frac{eeee}{\rtimes \rtimes \ltimes \ltimes} \quad \frac{eeee}{\rtimes \ltimes \ltimes \ltimes}$$

Let's quickly verify that this grammar generates the string *aaabb* but not *aaabbabb* or *aaabbaaab*.

$$\begin{array}{cccccccccccc} & & & e & e & e & e & e & e & o & e & e & e & e \\ & & & \rtimes & \rtimes & \rtimes & a & a & a & b & b & \ltimes & \ltimes & \ltimes \end{array}$$

$$\begin{array}{cccccccccccccc} e & e & e & e & e & e & o & e & e & e & & & & \\ \rtimes & \rtimes & \rtimes & a & a & a & b & b & a & a & b & b & \ltimes & \ltimes & \ltimes \end{array}$$

$$\begin{array}{cccccccccccccc} e & e & e & e & e & e & o & e & e & e & e & o & & & \\ \rtimes & \rtimes & \rtimes & a & a & a & b & b & a & a & a & b & \ltimes & \ltimes & \ltimes \end{array}$$

We are now going to convert this grammar into a refined strictly 2-local one. The operation is very simple to carry out: every 4-gram is first split into two 3-grams, one of which includes the first three positions, the other one the last three positions. We then use these two 3-grams as the hidden symbols for the 2-gram that consists of the last two positions of the 4-gram. For example, $\frac{eoee}{abba}$ is turned into the bigram below.

$$\frac{\frac{eoe}{abb} \; \frac{oee}{bba}}{b \quad a}$$

The whole grammar consists of the following bigrams (the translation also produces a few refined bigrams for $\ltimes\ltimes$, but those are never useful in a strictly 2-local grammar

and can safely be discarded):

$$\frac{eee\ eee}{\rtimes\rtimes\rtimes\rtimes\rtimes a}\ \frac{eee\ eee}{\rtimes\rtimes a\ \rtimes aa}\ \frac{eee\ eee}{\rtimes aa\ aaa}\ \frac{eee\ eee}{\rtimes\rtimes\rtimes\rtimes\rtimes\ltimes}$$
$$\rtimes\ a \qquad a\ a \qquad a\ a \qquad \rtimes\ \ltimes$$

$$\frac{eee\ eee}{aaa\ aa\ltimes}\ \frac{eoe\ oee}{abb\ bb\ltimes}\ \frac{eee\ eee}{baa\ aa\ltimes}\ \frac{oee\ eee}{bba\ ba\ltimes}\ \frac{eoe\ oee}{bbb\ bb\ltimes}$$
$$a\ \ltimes \qquad b\ \ltimes \qquad a\ \ltimes \qquad a\ \ltimes \qquad b\ \ltimes$$

$$\frac{eee\ eee}{aaa\ aaa}\ \frac{eee\ eeo}{aaa\ aab}\ \frac{eeo\ eoe}{aab\ abb}\ \frac{eoe\ oee}{abb\ bba}\ \frac{eoe\ oeo}{abb\ bbb}$$
$$a\ a \qquad a\ b \qquad b\ b \qquad b\ a \qquad b\ b$$

$$\frac{eoe\ oeo}{bbb\ bbb}\ \frac{oeo\ eoe}{bbb\ bbb}\ \frac{eoe\ oee}{bbb\ bba}\ \frac{oee\ eee}{bba\ baa}\ \frac{eee\ eee}{baa\ aaa}$$
$$b\ b \qquad b\ b \qquad b\ a \qquad a\ a \qquad a\ a$$

When we use this grammar to determine the well-formedness of the previous three example strings, we once again see that only the first one has a run and is thus generated by the grammar.

$$\frac{eee}{\rtimes\rtimes\rtimes}\ \frac{eee}{\rtimes\rtimes a}\ \frac{eee}{\rtimes aa}\ \frac{eee}{aaa}\ \frac{eeo}{aab}\ \frac{eoe}{abb}\ \frac{oee}{bb\ltimes}$$
$$\rtimes \qquad a \qquad a \qquad a \qquad b \qquad b \qquad \ltimes$$

$$\frac{eee}{\rtimes\rtimes\rtimes}\ \frac{eee}{\rtimes\rtimes a}\ \frac{eee}{\rtimes aa}\ \frac{eee}{aaa}\ \frac{eeo}{aab}\ \frac{eoe}{abb}\ \frac{oee}{bba}\ \frac{eee}{baa}$$
$$\rtimes \qquad a \qquad a \qquad a \qquad b \qquad b \qquad a \qquad a \qquad b \quad b \quad \ltimes$$

$$\frac{eee}{\rtimes\rtimes\rtimes}\ \frac{eee}{\rtimes\rtimes a}\ \frac{eee}{\rtimes aa}\ \frac{eee}{aaa}\ \frac{eeo}{aab}\ \frac{eoe}{abb}\ \frac{oee}{bba}\ \frac{eee}{baa}\ \frac{eee}{aaa}\ \frac{eeo}{aab}$$
$$\rtimes \qquad a \qquad a \qquad a \qquad b \qquad b \qquad a \qquad a \qquad a \qquad b \qquad \ltimes$$

As this example shows, we never need a search window bigger than 2 because the hidden alphabet can keep track of all the extra information a bigger window would provide. Turning this intuition into a proof is fairly straight-forward, but requires a loot of bookkeeping via indices.

Proof. We show that every strictly k-local grammar G_k with refined alphabet $\langle \Sigma_k, Q_k \rangle$ can be converted into a strictly 2-local grammar G_2 with refined alphabet $\left\langle \Sigma_k, (\Sigma_k \times Q_k)^{k-1} \right\rangle$ such that $L(G_k) = L(G_2)$. Let G_2 be the smallest set of bigrams such that if $g := \frac{q_1 \cdots q_k}{\sigma_1 \cdots \sigma_k}$ is a k-gram of G_k, then G_2 contains $g' := \frac{\phi\rho}{\sigma_{k-1}\sigma_k}$, where $\phi := \frac{q_1 \cdots q_{k-1}}{\sigma_1 \cdots \sigma_{k-1}}$ and $\rho := \frac{q_2 \cdots q_k}{\sigma_2 \cdots \sigma_k} \in Q_k^{k-1}$. We give an inductive proof that $L(G_2) = L(G_k)$.

Suppose $w \in L(G_k)$, and that $g_k := \frac{q_1 \cdots q_k}{\sigma_1 \cdots \sigma_k}$ is the refined k-gram spanning from the m-th to the $(m+k-1)$-th symbol of the k-augmented counterpart $\hat{w}_k$ of w.

For the base case assume $m = 1$. Then the first $k-1$ symbols of g_k are left edge markers: $\sigma_i = \rtimes$, $1 \leq i < k$, while the 2-augmented counterpart $\hat{w}_2$ lacks the first

$k-2$ symbols of $\hat{w}_k$. Given the construction above, G_2 contains the refined bigram

$$g_2 := \frac{\dfrac{q_1 \cdots q_{k-1}}{\sigma_1 \cdots \sigma_{k-1}} \cdot \dfrac{q_2 \cdots q_k}{\sigma_2 \cdots \sigma_k}}{\rtimes \qquad \sigma_k},$$

which can be assigned to the first two positions of $\hat{w}_2$. In the other direction, if g_2 is assigned to the first 2 positions of $\hat{w}_2$, then g_k can be assigned to the first k positions of $\hat{w}_k$. Since $g_2 \in G_2$ iff $g_k \in G_k$, the first 2 positions of $\hat{w}_2$ are well-formed wrt G_2 iff the first k positions of $\hat{w}_k$ are well-formed with respect to G_k.

For arbitrary m, suppose that g_k spans from the m-th to the n-th position of $\hat{w}_k$, where $n = m+k-1$. By our induction hypothesis, some bigram spans the positions in $\hat{w}_2$ that correspond to $n-2$ and $n-1$ of $\hat{w}_k$— namely $n-(k-2)-2 = m-1$ and $n-(k-2)-1 = m$, respectively. Moreover, the second component of this bigram is

$$\frac{\dfrac{q_1 \cdots q_{k-1}}{\sigma_1 \cdots \sigma_{k-1}}}{\sigma_{k-1}}.$$

By our construction, then, there is a $g_2 \in G_2$ that spans from m to $m+1$ and has the shape

$$\frac{\dfrac{q_1 \cdots q_{k-1}}{\sigma_1 \cdots \sigma_{k-1}} \cdot \dfrac{q_2 \cdots q_k}{\sigma_2 \cdots \sigma_k}}{\sigma_{k-1} \qquad \sigma_k}$$

iff $g_k \in G_k$. □

2 Regular Languages and Finite-State Automata

2.1 From Bigrams to Automata

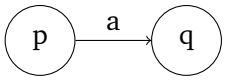
Theorem 9.6 shows that every refined strictly local language is refined strictly 2-local. Rather than the infinite hierarchies of increasing complexity that we saw with the strictly local and strictly piecewise languages, the class of refined strictly local languages is flat, at least with respect to the size of k-grams. To emphasize this flatness, we don't just drop the locality parameter — which might be mistaken as referring to a more powerful class that is the union of all refined strictly k-local languages — but coin a completely new term.

Definition 9.7 (Regular Languages). A language is *regular* iff it is generated by a refined strictly 2-local grammar.

Regular languages are one of the most important classes of string languages in computer science and have been defined in numerous equivalent ways. It simply isn't feasible for us to look at all of them, but most are covered in every decent textbook on formal language theory such as Sipser (2005), Kozen (1997), or Hopcroft & Ullman (1979) (listed in increasing order of difficulty).

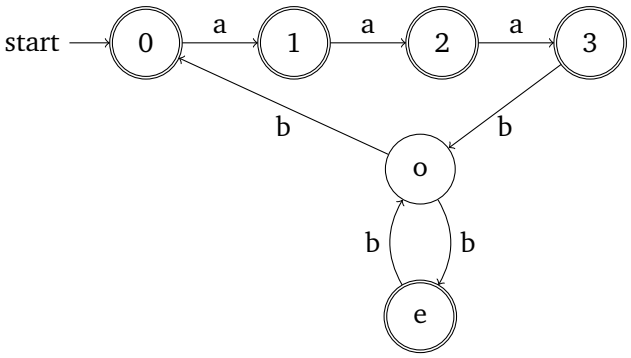
At least one of these equivalent definitions of regular languages can be inferred rather easily from example 9.4, though. There we saw that parts of a k-gram can be crammed into the hidden alphabet of a bigram. But in principle we could do the same

thing to the bigram and copy the overt symbols into the hidden alphabet, too. In this case, we can infer the state of a node purely from the label of that node and the state of the preceding node; the label of the preceding state is no longer needed. Consequently, every bigram $\frac{pq}{ba}$ satisfying this property can be represented by a directed graph.



This graph has the *vertices* p and q, and a (directed) *edge* from p to q that is labeled a. We can think of the graph as a machine or *automaton* that switches from state p to q if it reads in an a.

The whole grammar is an automaton that combines the mini-automata of all the bigrams (this combination is achieved via automata intersection, which is discussed in Sec. 2.3). For the language in example 9.4, the automaton is actually much easier to understand than the refined grammar we constructed.



The graph uses start to indicate which state(s) may be assigned to the first node (*initial states*), whereas circled (*final*) states are the only ones that may be assigned to the final node. A string w is well-formed iff there is path from an initial state to a final state such that the sequence of edge labels is identical to w. This definition leaves open whether we should think of the automaton as a *recognizer*, similar to the scanner we used for strictly local languages, or as a *generator* that is run to produce strings. The two perspectives have no impact on the formal properties of the automaton, so we will often switch between the two depending on the area of application.

As a recognizer, the automaton reads in a string and assigns each node a state. So in contrast to the formally equivalent perspective via refined grammars, we do not start out with a run and check its well-formedness but rather try to build a successful run in a step-by-step fashion. Once again, though, this has no repercussions for how we represent runs; as you can see in the examples below, the only change is that there are no edge markers, so the initial state is left "dangling". Also note that the states are different, but they still lead to the same conclusions.

	0	1	2	3	*o*	*e*
		a	*a*	*a*	*b*	*b*

0	1	2	3	*o*	0	1	2		
	a	*a*	*a*	*b*	*b*	*a*	*a*	*b*	*b*

0	1	2	3	*o*	0	1	2	3	*o*
	a	*a*	*a*	*b*	*b*	*a*	*a*	*a*	*b*

While there are many different types of automata, we are only interested in those with a finite number of states. After a few ancillary results have been established, we will be able to prove that these automata generate exactly the class of regular languages. So they are indeed just a different way of specifying the kind of dependencies that are computed by refined strictly 2-local grammars.

Definition 9.8 (Finite-State Automaton). A *finite state automaton* (FSA) is a quintuple $A := \langle \Sigma, Q, I, F, \Delta \rangle$, where

- Σ is an alphabet,
- Q is a finite set of states,
- $I \subseteq Q$ is the set of *initial* states,
- $F \subseteq Q$ is the set of *final* states,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a finite set of transition rules.

The FSA is *deterministic* iff I is a singleton set Δ contains no two $\langle p, a, q \rangle$ and $\langle p, a, r \rangle$ such that $q \neq r$ or $a = \varepsilon$. Otherwise it is non-deterministic. For deterministic FSA, we also write $\delta(q, a) = q'$ instead of $\langle q, a, q' \rangle \in \Delta$.

In our graphical representation, determinism holds iff there is never a choice as to which outgoing branch the automaton should follow. This is the case only if, first, all outgoing branches of a state have distinct labels, and second, no outgoing arc is labeled ε. The example automaton above is non-deterministic because both branches leaving the state o are labeled b.

2.2 Deterministic and Non-Deterministic Automata

Determinism is an important property of FSAs. A deterministic FSA never has to pick between multiple valid states for a given symbol, at any given node in the string the state is uniquely determined by the symbol of the current node and the state of the preceding node (if it exists). Hence there is only one valid state assignment for each string. If we think of automata as a more elaborate kind of scanner — moving through the string from left to right and assigning each node a state — this means that the well-formedness of a string can be established in a single left-to-right pass: if the automaton has not found a valid state assignment the first time around, the string must be ill-formed because there is no point at which the automaton might have made a wrong guess. Formally, this means that an automaton can accept or reject a string in $O(n)$, i.e. in linear time. That is identical to the time complexity of accepting a string of a strictly local language, since there, too, it suffices to run the scanner one time from left-to-right, with each symbol of the string adding a fixed number of steps to the computation. So at least with respect to overall processing speed, the addition of a hidden alphabet does not have any negative effects as long as we can use a deterministics FSA.

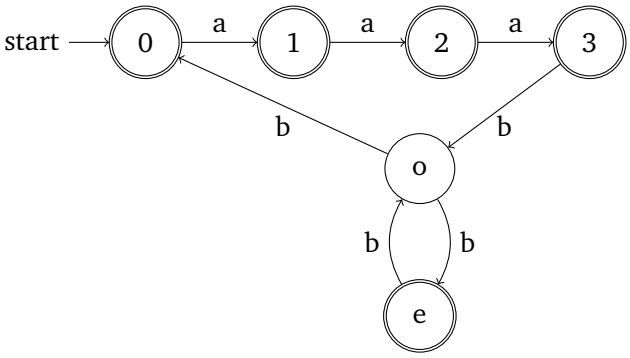
But if the phonological process we are interested in is specified via a non-deterministic FSA, do we see a slow-down in this case? Only if we use a very inefficient implementation. When faced with problems that involve non-determinism, humans tend to use

a serial strategy: at each point of non-determinism, make an educated guess about how to proceed, and if that doesn't work out go back to the start and try again with different choices. For example, when asked to draw a line through a maze from the start to the exit, you may first sketch out a patch with your finger, following a specific path at each juncture and backtracking whenever you reach a dead end. In the case at hand, such a strategy is not in $O(n)$ because the string might have to be processed as many times are there are distinct options, and this number does not grow linearly. If there are 2 decision points, each one corresponding to a binary decision, then there are 4 options, whereas with 3 decision points there are already 8 options. So in the former case determining well-formedness would take $4n$ steps, in the latter $8n$, an exponential increase with respect to the number of decision points. Crucially, the number of decision points can increase with the length of the string. Therefore a serial strategy simply isn't feasible due to how quickly the number of possible runs grows.

A parallel strategy is a viable alternative: the string is read only once from left to right, and all alternatives are explored at the same time. Returning to the intuitive example of a maze, for instance, you can cut down quite a bit on backtracking if you use two fingers to follow two paths at the same time. If you have a large number of fingers at your disposal (maybe a few friends are quite literally lending a hand?) you can explore all paths at the same time. The same strategy can be applied to non-deterministic automata, so that rather than one run at a time, all possible runs are explored in parallel. On a technical level, this amounts to constructing a deterministic automaton where each state represents all possible configurations of the non-deterministic automaton at a given point. The downside is that just like your parallel maze search requires a lot more fingers, determinizing a non-deterministic automaton in this way can induce an exponential blow-up in the number of states. The more states an automaton has, the more memory it consumes. We are dealing with a *space-time tradeoff*: we can construct a large, memory-intensive automaton that runs in linear time, or a small, memory-efficient one that runs in exponential time.

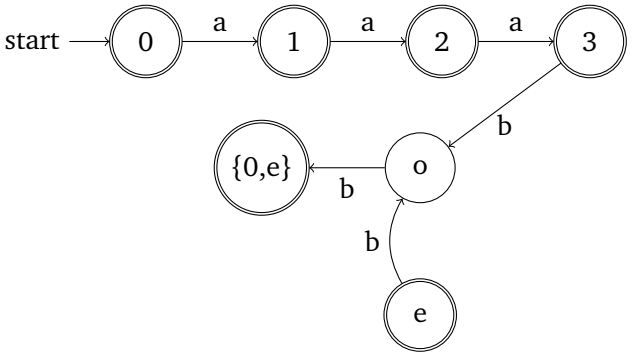
Example 9.5 Determinizing The Non-Deterministic Example Automaton

Consider once more the non-deterministic automaton for example 9.4, repeated below for your convenience.

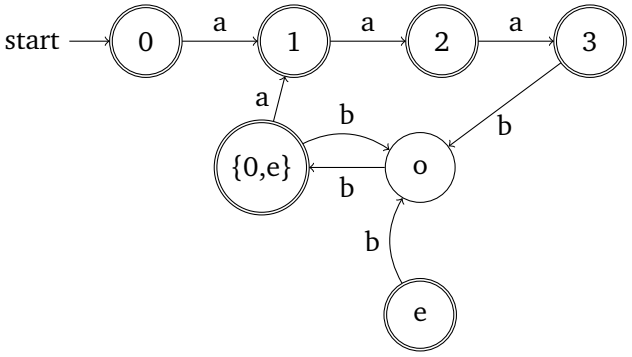


There is only one instance non-determinism, namely the two b-transitions out of state o. In order to make the automaton deterministic, we have to represent the two alternative routes that start at this point into one state. The states that can be reached from o via b are 0 and e, so we create a new state $\{0, e\}$ that is connected to o via a

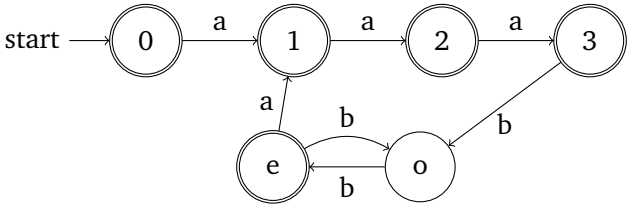
b-transition. All other b-arcs out of o are removed. Note that since $\{0, e\}$ contains at least one final state of the non-deterministic automaton, it is itself a final state.



In the next step, we have to add outgoing arcs to the newly created state. In the original automaton, one can move from 0 to 1 via a, so we add an a-edge from $\{0, e\}$ to 1. As one can also move from e to o via b, we also add a b-arc back to o.



We now have a fully deterministic automaton. Since the state e no longer has any entering arcs and is thus unreachable we can safely remove it. The final deterministic automaton now looks almost exactly the same as the original one, we can even rename $\{o, e\}$ back into e. This makes it clear that the only difference is in two transitions.



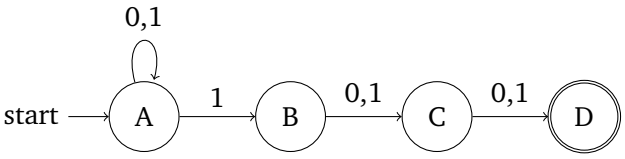
The example above is a very special case in that the determinization does not need increase the number of states and can easily be carried out with intuitive reasoning alone. In general, determinization is a more complex affair and best carried out via a specific conversion algorithm, known as the *powerset construction*.

Powerset Construction Given a non-deterministic FSA $A := \langle \Sigma, Q, \Delta, I, F \rangle$, its deterministic counterpart is $A_d := \langle \Sigma, Q_d, \Delta_d, I_d, F_d \rangle$, where

- Q_d is the powerset of Q,
- $q \in F_d$ iff $q \cap F \neq \emptyset$,
- I_d contains only the state that is identical to I,
- for all $q_d \in Q_d$ and $a \in \Sigma$, $\delta_d(q_d, a) := \{q' \mid \langle q, a, q' \rangle \in \Delta, q \in q_d\}$.

Example 9.6 Determinization via Powerset Construction

The automaton below accepts every string over 0 and 1 where the antepenultimate symbol is 1.



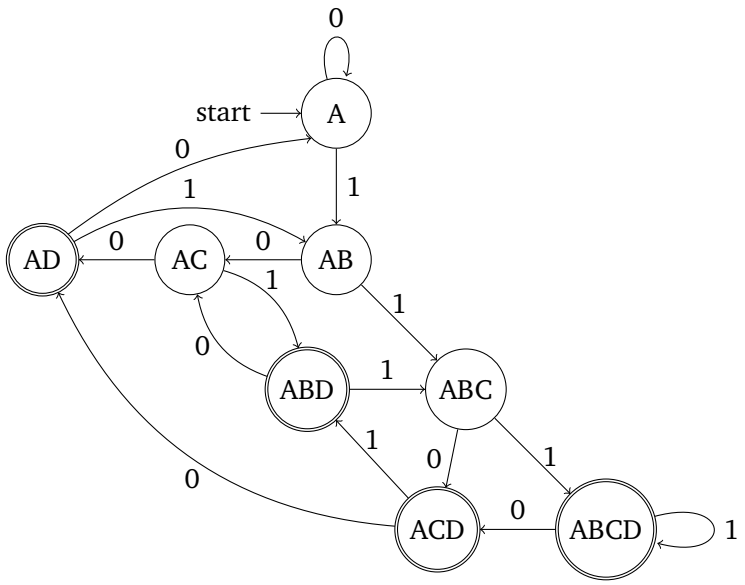
Once again we have only one case of non-determinism, where two arcs that leave A are both labeled 1. This time, however, determinization doubles the size of the state set. First we construct a transition table according to the powerset algorithm. A symbol s in row r and column c means that there is an s-arc from c to r.

	A	**B**	**C**	**D**	**AB**	**AC**	**AD**	**ABC**	**ABD**	**ACD**	**BC**	**BD**	**BCD**	**CD**	**ABCD**
A	0				1										
B			0,1												
C				0,1											
AB						0		1							
AC							0		1						
AD	0				1										
ABC										0					1
ABD						0		1							
ACD							0		1						
BC														0,1	
BD			0,1												
BCD														0,1	
CD				0,1											
ABCD										0					1

Next all empty columns and all rows that no longer have a matching column are removed from this table, and this procedure is repeated until no empty columns or rows remain:

	A	**AB**	**AC**	**AD**	**ABC**	**ABD**	**ACD**	**ABCD**
A	0	1						
AB			0		1			
AC				0		1		
AD	0	1						
ABC							0	1
ABD			0		1			
ACD				0		1		
ABCD							0	1

This table fully describes the deterministic automaton below, which 8 states instead of the original 4.



For real-world applications, one usually uses small non-deterministic automata to define the desired behavior or process, and then uses an algorithm to construct a large deterministic automaton that can be run efficiently. From a scientific perspective, it is a lot less clear what kind of automaton may be more adequate for various aspects of human cognition. On the one hand, human working memory seems to be very limited, yet at the same time most computations are carried out at remarkable speed. Things are complicated even more by the existence of a third alternative known as *hyper minimization*. This is an extension of minimization, a long-known method for converting an FSA into the smallest deterministic FSA that recognizes the same language. Hyper minimization takes this one step further and converts an FSA into the smallest deterministic FSA that recognizes almost exactly the same language. A hyperminimized FSA thus makes a certain number of mistakes, but it is also much smaller than the original automaton. So maybe those areas of human cognition that need the power of FSAs use hyperminimized FSAs, thus combining speed with manageable memory usage and adequate empirical coverage.

These psychological issues have not been explored much in the literature, and for our purposes the only important thing is that non-deterministic FSAs can be made deterministic (even if it comes at the expense of greater memory usage). This shows that the two types of FSAs have exactly the same generative capacity, so whenever we talk about arbitrary FSAs we may assume that they are deterministic.

2.3 Operations on Automata

FSAs can be combined in a variety of ways. In particular, the class of FSAs is closed under the boolean operations intersection, union, and (relative) complementation. That is to say, given two automata we can automatically construct a new automaton that computes the intersection, union, or (relative) complement of the languages defined by these automata. This section describes the relevant construction, but proofs

of their correctness are omitted.

Intersection Suppose that $A_1 := \langle \Sigma, Q_1, \delta_1, q_1, F_1 \rangle$ and $A_2 := \langle \Sigma, Q_2, \delta_2, q_2, F_2 \rangle$ are deterministic FSAs that accept the languages L_1 and L_2, respectively (remember that deterministic FSAs have only one initial state, which is listed here explicitly). Then clearly a string is in the intersection of L_1 and L_2 iff it is recognized by both A_1 and A_2. If we can construct an automaton that somehow executes both A_1 and A_2 in parallel, then this automaton will recognize exactly the intersection of L_1 and L_2. The powerset construction in the previous section has already shown us how we can use complex states to represent multiple runs of the same automaton. Intersection is a modification of this idea where the state set encodes the run of each automaton. States are now pairs, with the first component keeping track of the current state of A_1, while the second one tracks A_2. A string is well-formed iff the components of last state of the run are both final states. More precisely, $A_1 \cap A_2 := \langle \Sigma, Q, \delta, q_0, F \rangle$ where

- $Q := Q_1 \times Q_2$,
- $q_0 := \langle q_1, q_2 \rangle$,
- $F := \{\langle p, q \rangle \mid p \in F_1 \text{ and } q \in F_2\}$
- $\delta(\langle p, q \rangle, a) = \langle p', q' \rangle$ such that $\delta_1(p, a) = p'$ and $\delta_2(q, a) = q'$.

Complement The complement of L contains no string of L and all strings that are not in L. If we no longer want an automaton to recognize any strings of L, we have to convert all final states into non-final ones. Similarly, every string not in L will be recognized is all non-final states are made final. So the complement of an automaton is obtained by switching the status of final and non-final states. The relative complement $L_1 \setminus L_2$ of two languages L_1 and L_2 is the result of intersecting L_1 with the complement of L_2 and thus can be obtained by a sequence of two automata constructions: complementation followed by intersection.

Union By De Morgan's law, $L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$, so the union of two automata is the complement of the intersection of their complements. But there are also more insightful ways of obtaining the union of two automata. First, one can simply take the intersection automaton and extend the set of final states such that $F := \{\langle p, q \rangle \mid p \in F_1 \text{ or } q \in F_2\}$. This new automaton accepts a string as long as it would be recognized by at least one of the two original automata, which is exactly what union amounts to. The major advantage of this method is that it takes fewer steps than the De Morgan translation (all three complementation steps are avoided) and preserves determinism.

Alternatively, we can simply take the two automata and view them as one non-deterministic automaton with two initial states. Depending on which initial state one starts in, either the first or the second automaton is used to determine well-formedness. More formally (assuming that Q_1 Q_2 are disjoint, which can always be achieved by renaming states as necessary):

- $Q := Q_1 \cup Q_2$,
- $I := I_1 \cup I_2$,
- $F := F_1 \cup F_2$,

- $\Delta := \delta_1 \cup \delta_2$.

3 Properties of Regular Languages

3.1 Equivalence of Refined Grammars and Finite-State Automata

While the connection between refined strictly 2-local grammar and FSAs is rather obvious, we haven't given a formal proof yet that the two define exactly the same class of languages.

Lemma 9.9. For every FSA there is a refined strictly 2-local grammar that generates the same language. ⌟

Proof. W.l.o.g. let $A := \langle \Sigma, Q, \Delta, q_0, F \rangle$ be a deterministic FSA with at least one transition rule (which may be an ε-transition). Let G be a refined strictly 2-local grammar over alphabet $(\Sigma \cup \{\rtimes, \ltimes\}) \times Q$. For every $\langle p, a, q \rangle \in \Delta$,

- $\frac{pq}{\sigma a} \in G$ for every $\sigma \in \Sigma$,
- if p is an initial state, G also contains $\frac{pq}{\rtimes a}$,
- if q is a final state, G also contains $\frac{qq}{a \ltimes}$.

No other bigrams are contained by G. It is easy to see that A and G assign the same states to all nodes in a given string. □

Lemma 9.10. For every refined strictly 2-local grammar there is an FSA that generates the same language. ⌟

Proof. For G a refined strictly 2-local grammar over alphabet Σ, let $A := \langle \Sigma, G, \Delta, I, F \rangle$, where

- $I := \left\{ \frac{pq}{\rtimes a} \in G \right\}$,
- $F := \left\{ \frac{pq}{a \ltimes} \in G \right\}$,
- $\langle u, a, v \rangle \in \Delta$ for $u := \frac{pq}{xa}$ and $v := \frac{qr}{ay}$ $(x, y \in \Sigma)$,

As the states replicate exactly the grammar, a string is well-formed with respect to A iff it is well-formed with respect to G. □

The two lemmata establish a close connection between refined grammars and automata, but we can broaden this perspective by also formalizing the intuition that the languages generated by a refined grammar are essentially strictly 2-local languages over a hidden alphabet. In linguistic terms, we have a richly annotated, strictly 2-local language of underlying forms that is mapped to a language of surface forms via a simple relabeling that removes the extra annotation.

Definition 9.11 (Projection). Let Σ and Ω be two alphabets such that $|\Sigma| \geq |\Omega|$. A *projection* π is a total function from Σ onto Ω. We extend this to a function over strings in a piecewise fashion such that $\pi(a_1 \cdots a_n) = \pi(a_1) \cdots \pi(a_n)$. Given two languages L_Σ and L_Ω, L_Ω is a projection of L_Σ iff there is a projection π with $L_\Omega := \{\pi(w) \mid w \in L_\Sigma\}$. In this case we also say that L_Σ is a *cylindrifaction* of L_Ω.

Lemma 9.12. Every projection of every strictly 2-local language L is recognized by some FSA. ⌟

Proof. Suppose L is a strictly 2-local language generated by grammar G over alphabet $\Sigma \cup \{\rtimes, \ltimes\}$ and $\pi : \Sigma \to \Omega$ a projection. We construct an automaton that recognizes the image L_π of L under the projection π.

First, π^{-1} identifies with every $\omega \in \Omega$ the set $\{\sigma \mid \pi(\sigma) = \omega\}$. Now let $A := \langle \Omega, Q, \Delta, q_0, F \rangle$ be such that

- $Q := \wp(\Sigma) \cup \{\rtimes\}$,
- $q_0 = \{\rtimes\}$,
- $F := \left\{\pi^{-1}(\pi(a)) \mid a\ltimes \in G\right\}$,
- $\delta(p, a) = \left\{a_\Sigma \in \pi^{-1}(a) \mid ba_\Sigma \in G, b \in p\right\}$.

The automaton uses its states to assign every node with label l a set of symbols that might be a projection of l and can according to G, follow one of the symbols in the set of the preceding node. □

All of this shows that the power of regular languages stems from the ability to abstract away from the surface string and keep track of information that cannot be inferred from the output alphabet itself. It does not matter whether we think of this abstraction as bigram refinement, states of an automaton, or simply a mapping from rich underlying alphabets to surface alphabets. While they appear to be very different perspectives, they all grant us the same amount of power and hence define exactly the same class of languages.

Theorem 9.13. Let L be a string language. Then the following claims are equivalent:

1. L is a projection of a strictly 2-local language,
2. L is recognized by a finite-state automaton,
3. L is generated by a refined strictly 2-local grammar,
4. L is regular. ⌟

Proof. Statement 3) and 4) mutually imply each other by definition. Hence it suffices to show for 1), 2), 3) that each statement implies the next one. That 1) implies 2) and 2) implies 3) follows immediately from the previous lemmata, so it only remains to show that 3) implies 1). But this is obvious, since every refined strictly 2-local grammar G_R can be viewed as normal strictly 2-local grammar G over alphabet $\Sigma \times Q$ such that $L(G_R) := \pi(L(G))$ where π maps each $\langle \sigma, q \rangle$ to σ. □

Corollary 9.14. The class of regular languages is closed under intersection, union, and relative complement. ⌟

Proof. This follows from the analogous closure properties of the class of FSAs. □

3.2 Is Phonology Regular?

Regular languages are much more powerful than anything we have seen so far. They properly include all strictly local, strictly piecewise, and strictly threshold testable languages, and they can easily enforce complex conditions like the LHOR stress pattern, which we could only model over the alphabet $\{S, U\}$ for stressed and unstressed syllables. This raises two questions:

1. Do we need the extra power?
2. Do we need even more power?

The second question will be answered in the next lecture, so let's focus on the first one for now.

As mentioned last time, it looks like segmental phonology belongs to the union of SL_k and SP_j for some fixed j and k. Suprasegmental phonology is more demanding, but seems to fall into the same class if one assumes that I) cumulativity is factored out, II) cumulativity is computed with respect to the alphabet $\{S, U\}$, and III) the application domain of phonology is words, rather than strings of words. These are very specific assumptions that might easily be wrong. One might suspect that the power of regular languages becomes indispensable once one of them is them dropped, but this is not the case. While dropping each one of these assumptions does increase the power demands, we never hit the level of regular languages.

If cumulativity needs to be stated for a more elaborate alphabet such as {H́,Ĺ,H,L}, one has to move from strictly locally threshold testable languages to *locally threshold testable* languages. These make it possible to state conditions like "at least 1 H́ or at least 1 Ĺ", which is exactly what is needed for cumulativity (the subclass where the threshold is always 1 is also called locally testable). If one drops the assumption that the domain of phonological processes is automatically restricted to a single word, then long distance dependencies and stress patterns are no longer strictly piecewise even if cumulativity is already accounted for. That's because constraints and processes that apply within a single word now must be explicitly restricted to this domain, which is not trivial. The *star-free* languages can do this, as they enrich the locally threshold testable languages with non-local dependencies and the option to state interval conditions such as "exactly one primary stress between the smallest interval spanning from a left edge marker to a right edge marker (i.e. within a single word)". But the star-free languages are still a proper subclass of the regular languages. Are there any phonological phenomena that are regular but not star-free?

The main difference between regular and star-free languages is that the latter may exhibit mathematical restrictions on the number of symbols in a string. In particular, regular languages may involve *modulo* counting, requiring that the number of symbols is, say, a multiple of 2 or 3. We have already encountered an example of that, namely $(aa)^+$. This language is regular, but not star-free. It is unclear whether phonology involves any dependencies of this kind. It has been argued that primary stress assignment in Creek and Cairene Arabic involve modulo counting as under very specific conditions primary stress goes on the rightmost syllable that is an even number of syllables away from the left edge of the word (**Graf10PLC33**, **Graf10thesis**). But the data is rather dubious and so far there have been no attempts to replicate it under carefully controlled conditions. At this point, then, there is no conclusive evidence that the regular languages are a more appropriate model of phonology than the star-free ones, or, given a suitable factorization of the workload, the union of SL and SP.

My personal hunch is that the generalization in the literature is wrong and that the observed stress patterns follow a simpler rule.

Relevant literature for Unit 9

add more info

Hopcroft, John E. & Jeffrey D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Reading, MA: Addison Wesley.

Kozen, Dexter C. 1997. *Automata and computability*. Springer.

Sipser, Michael. 2005. *Introduction to the theory of computation*. Second. Course Technology.