Lecture 5

Strict Locality: Alternative Characterizations

The previous chapter was all about extending bigram grammars to strictly local grammars while changing as little about our perspective as possible. This chapter does the exact opposite. Following the Keenan-Moss credo of saying something in as many ways as possible, we take a look at several distinct characterizations of strict locality. We start out simple by exploring different methods to represent and store strictly local grammars. This reinforces the message of Ch. 1 about Marr's levels of description: the further we move away from the computational level towards the algorithmic level, the more implementation details do we have to take care of that may have an effect on runtime behavior and overall performance but are ultimately immaterial for the computational properties we are interested in. The second half of the chapter then moves on to characterizations of strict locality that are less directly tied to grammars: automata and a specific fragment of propositional logic. These perspectives still add some new facets to our already well-developed understanding of strictly local languages, and they will become even more useful when we explore extensions and generalizations in the upcoming chapters.

1 Implementations of Strictly Local Grammars

1.1 Prefix Trees Revisited

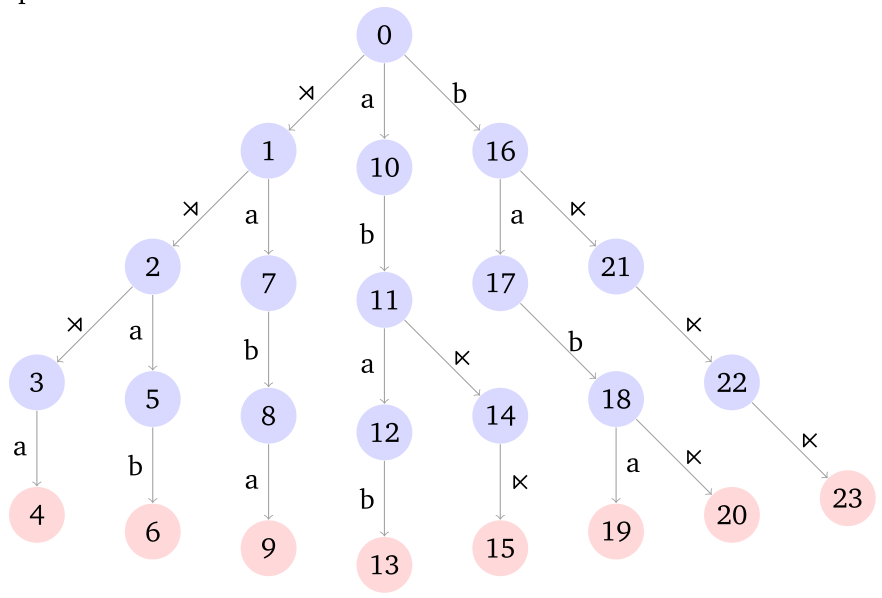
The discussion of the list phonology model in Ch. 2 spent quite some time on how a list of surface forms can be efficiently searched and stored. Prefix trees turned out to be ideal for this purpose as they provide a more compact representation but can also be searched very quickly — finding an item only requires following one specific branch for each sound in the word. It doesn't take much ingenuity to realize that strictly local grammars, too, can be stored as prefix trees.

Example 5.1 Prefix Tree for a Strictly 4-Local Grammar

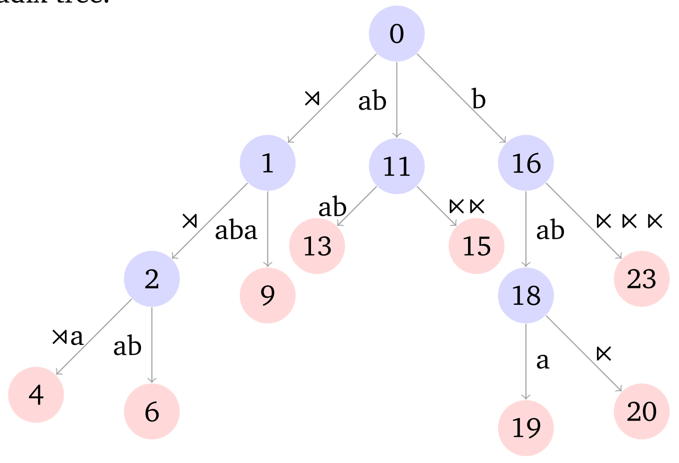
Consider the strictly 4-local grammar, which was also used in Fig .4.1.

$$\begin{array}{lll}
\rtimes\rtimes\rtimes a & abab & bab\ltimes \\
\rtimes\rtimes ab & baba & ab\ltimes\ltimes \\
\rtimes aba & & b\ltimes\ltimes\ltimes
\end{array}$$

This set corresponds to the prefix tree below, where nodes are numbered for the sake of exposition.



The prefix tree has lots of unary branches that can be compressed further to yield a radix tree.



In general, one might expect the prefix trees of strictly local grammars to be much smaller than those of a list phonology grammar, which probably needs to contain over 500,000 surface forms. But this is actually far from obvious. Suppose we have a language with 50 different phones, which is a rather conservative estimate — some languages have over 100. Then a strictly 3-local grammar can contain $50^3 = 125,000$ trigrams, and a strictly 5-local one 312.5 million! We can cut that number in half by converting from a positive grammar to a negative one, or the other way round. Remember, one is built by removing the *k*-grams of the other from the set of all possible *k*-grams, so the bigger the positive grammar the smaller the negative grammar, and the other way round. But 156.25 million 5-grams is still a shockingly large number. Are any natural language grammars actually this big? At this point, we simply do not know. Intuitively, though, it seems that individual phonotactic constraints are

In order to estimate the average number of *k*-grams used in a natural language, we would need fully worked out formal descriptions of many highly distinct languages. Unfortunately these are excessively rare, although phonology is still better of in this respect than syntax.

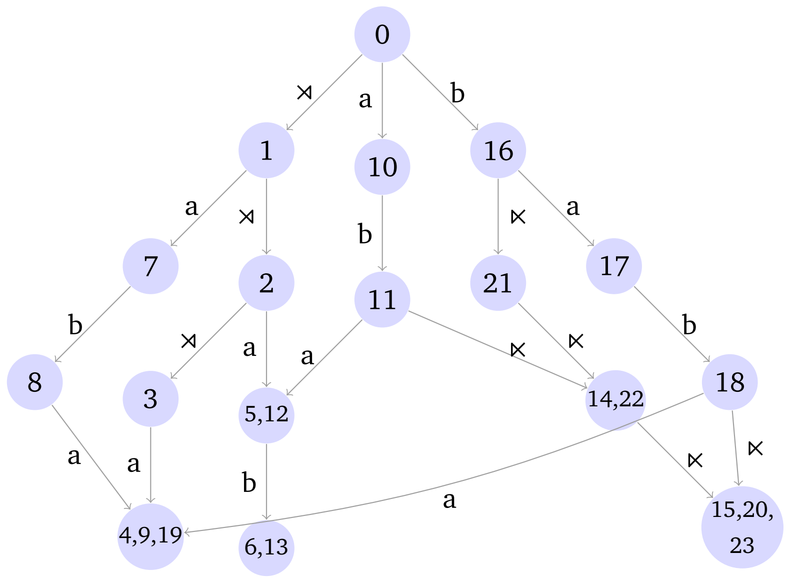
rather simple and can be captured with very small grammars. Also keep in mind that few processes require more than bigrams and trigrams, so if we have one grammar for each process and then enforce them all in parallel, we can reduce storage needs enormously compared to one monolithic strictly 5-local grammar. This is one of the many cases where factorization pays off in a big way, and thanks to closure under union we know that factorization is safe in the sense that it does not miss any strings or suddenly create new ones.

While factorization will save a lot of memory, it is also prudent to consider what minor optimizations are possible. As you can see in example 5.1 above, the prefix/radix trees for strictly local grammars differ slightly from those for the list phonology model in that they have only leaf nodes as final states. So we do not need to encode the distinction between final and non-final states, saving us a tiny amount of memory (1 bit per node in the tree).

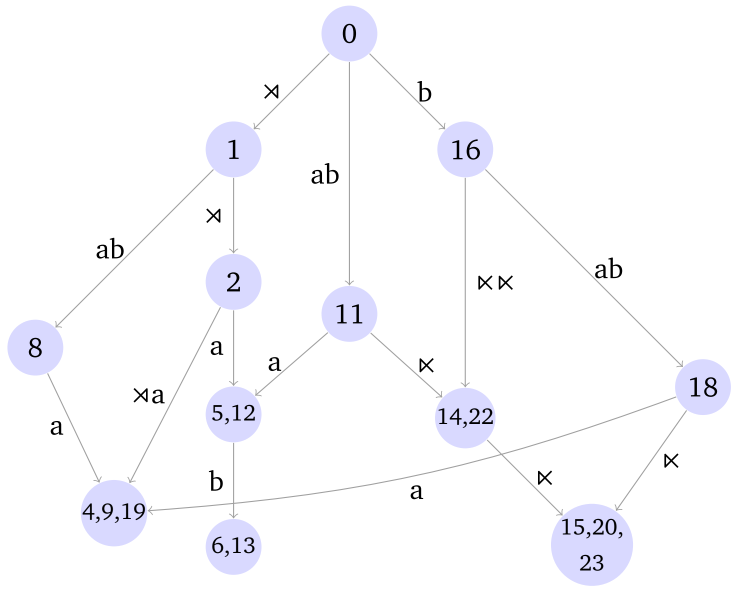
You might have also noticed that some branches are duplicated. To give but one example, the nodes 11 and 21 in the prefix tree of example 5.1 share the same sub-branch ⋉-⋉. We can combine these branches by first merging the nodes 14 and 22, and then 15 and 23. This does not change the grammar because the set of paths from a root to a leaf has not changed — in particular, we have not lost the paths a-b-⋉-⋉and b-⋉-⋉-⋉, and we have not gained any new paths. If we had merged 14 and 23 instead just because they both are reached via a ⋉arc, then we would have lost the path a-b-⋉-⋉and gained the path a-b-⋉. This would have changed the grammar to an extent where it wouldn't even be strictly 4-local anymore according to our definition. Nor is it licit to merge 7 and 17 because they both can be continued by the sub-path b-a. If we did that, we would lose no paths, but suddenly ⋊-a-b-⋉would be a possible path and the grammar would be able to generate *ab*. So it is important to verify that nodes are merged only if that does not affect the set of paths from the root to a leaf.

Example 5.2 DAG for Strictly 4-Local Grammar

The prefix tree from example 5.1 can be converted into the graph shown below. Note that nodes are no longer color-coded to distinguish final from non-final states since all leaf nodes are final, and only those.



This graph can be compacted even further by removing unary branching nodes in the same fashion that converts a prefix tree into a radix tree.



Note that the graph differs from the one one would obtain from the radix tree in example 5.1 by merging nodes. In particular, the former has 12 nodes and the latter 13.

Once we start merging trees, prefix trees are no longer trees because some nodes have more than one mother. Linguists call such trees *multi-dominance trees*. This term is unknown in computer science, and instead one speaks of *directed acyclic graphs* (DAGs). DAGs are slightly more general than multi-dominance trees because they can have multiple roots.

Definition 5.1 (DAG). A *graph* is a pair $\langle V, E\rangle$ consisting of a set V of *vertices* and a set $E \subseteq V \times V$ of *edges* connecting vertices. We also speak of *nodes* and *branches*, respectively. The reflexive, transitive closure of E is denoted E^*. A *directed acyclic graph* is a graph that satisfies the following axiom:

No cycles for all $u, v \in V$, $\langle u, v\rangle \in E$ implies $\langle v, u\rangle \notin E^*$.

A graph/DAG is *edge-labeled* iff it comes equipped with a function $\ell : E \to \Omega$ that assigns each edge $e \in E$ some symbol drawn from the alphabet Ω of edge labels.

Background Closure of a Relation

In Cha. 4 we encountered the notion of *closure* in the sense that a given object may be closed under some operation. A slightly different sense of *closure* is commonly used to construct new relations from old ones. Suppose R is some relation over set S, i.e. $R \subseteq S \times S$. Then for P some property of relations, the P-closure of R is the smallest relation R' such that $R \subseteq R' \subseteq S \times S$ and R' satisfies property P. Here are three common properties that are used in this connection:

reflexive $\langle u, u\rangle \in R$ (for all $u \in S$)

symmetric $\langle u, v\rangle \in R$ implies $\langle v, u\rangle \in R$ (for all $u, v \in S$)

transitive $\langle u, v\rangle \in R$ and $\langle v, w\rangle \in R$ jointly imply $\langle u, w\rangle \in R$ (for all $u, v, w \in S$)

The linguistic notion of proper dominance in a tree, for example, is the transitive closure of the mother-of relation. Reflexive dominance, on the other hand, is the reflexive transitive closure of the mother-of relation.

1.2 Matrices

While tree-based representations are very convenient for humans, it is far from obvious how one could implement them in some programming language. There exist specialized libraries for Python (search for *py-dag*, *biopython* or *marisa-trie*), but in general it is prudent to keep the number of dependencies for a program as small as possible without sacrificing essential functionality. In many cases, a simpler solution is to switch to a different representation format that is easier to use. For graphs there already is a well-known strategy: reencode them as *adjacency matrices*. An adjacency matrix has a row and a column for each node, and the value in row i and column j is x iff the graph contains an edge that spans from i to j and is labeled x.

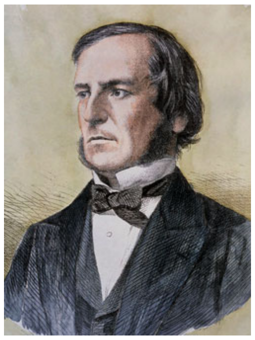
Example 5.3 Adjacency Matrix for a Radix Tree

The radix-like DAG from example 5.2 corresponds to the adjacency matrix below.

$$
\begin{array}{r|cccccccccccc}
 & 0 & 1 & 2 & 4,9,19 & 5,12 & 6,13 & 8 & 11 & 14,22 & 15,20,23 & 16 & 18 \\
\hline
0 & & \rtimes & & & & & & ab & & & b & \\
1 & & & \rtimes & & & & ab & & & & & \\
2 & & & & \rtimes a & a & & & & & & & \\
4,9,19 & & & & & & & & & & & & \\
5,12 & & & & & & b & & & & & & \\
6,13 & & & & & & & & & & & & \\
8 & & & & a & & & & & & & & \\
11 & & & & & a & & & & \ltimes & & & \\
14,22 & & & & & & & & & & \ltimes & & \\
15,20,23 & & & & & & & & & & & & \\
16 & & & & & & & & & & & & ab \\
18 & & & & & & & & & & \ltimes & &
\end{array}
$$

While Python does not include matrices as a basic data type (in contrast to other languages such as *R* or *Octave*), a 2-dimensional matrix can be treated as a list of lists. Listing 5.1 gives a Python function for converting graphs into this format and gives an example of how such lists can be queried.

While definitely useful, adjacency matrices make it very hard for humans to determine at a glance what grammar they encode. It is far from obvious that the table in example 5.3 encodes the same information as the set of 4-grams we started out with at the beginning of the chapter. That's not surprising, as the matrix is the output of a long chain of information-preserving transformations: from a set of 4-grams to a prefix tree to a DAG to a compacted DAG to an adjacency matrix. A slightly more intuitive route directly translates a strictly k-local grammar into a k-dimensional *Boolean matrix*. A Boolean matrix requires each cell to have the value True/1 or False/0. The idea is that we can map each symbol of our alphabet to a unique natural number such that if the Boolean matrix has a 1 in cell $i_1, \dots, i_k$, then the grammar contains a k-gram $s_1 \cdots s_k$ iff i_j is the natural number assigned to symbol s_j for all $1 \leq j \leq k$. That is quite a mouthful, but hopefully a quick example will make things clearer.


George Boole

The term Boolean was coined in honor of *George Boole*, one of the founding fathers of mathematical logic whose work is the theoretical foundation of the switching circuits used in computer hardware. Like many great thinkers, he died prematurely. Unlike most great thinkers, he has his wife to blame for that.

Example 5.4 Strictly k-Local Grammar as k-Dimensional Matrix

Suppose that our alphabet contains only the symbols a and b, plus the two edge markers. We randomly assign all four symbols natural numbers. The assignment below will work just fine for our purposes:

$$
\begin{array}{ccc}
\rtimes & \mapsto & 0 \\
a & \mapsto & 1 \\
b & \mapsto & 2 \\
\ltimes & \mapsto & 4
\end{array}
$$

Now consider the familiar strictly 2-local grammar $\{\rtimes a, ab, ba, b\ltimes\}$ for the string

```python
def graph2matrix(graph):
    """
    Converts graph into adjacency matrix, represented as list of lists

    Arguments:
    graph -- list of the form
             [ [list of vertices],
               [list of edges encoded as (source,target,label) tuples]
             ]
    """
    # inititalize empty matrix with a nested list comprehension
    graph_size = range(len(graph[0]))
    matrix = [['' for i in graph_size] for i in graph_size]

    # fill matrix
    for edge in graph[1]:
        source_index = graph[0].index(edge[0])
        target_index = graph[0].index(edge[1])
        edge_label = edge[2]
        matrix[source_index][target_index] = edge_label
    return matrix
```

```
>>> graph = [
...     ['A', 'B', 'C'],
...     [
...         ('A', 'B', 'e'),
...         ('A', 'C', 'f'),
...         ('B', 'B', 'g'),
...         ('C', 'A', 'h'),
...         ('C', 'C', 'i')
...     ]
... ]
>>> graph2matrix(graph)
[['', 'e', 'f'], ['', 'g', ''], ['h', '', 'i']]
>>> graph2matrix(graph)[graph[0].index('A')][graph[0].index('B')]
'e'
```

Listing 5.1: Python function for converting graphs to adjacency matrices

language $(ab)^+$. We can represent this grammar as a 2-dimensional Boolean matrix.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Here rows represent the first symbol of the bigrams, columns the second symbol. The first column is completely filled by 0s because no bigram in the grammar contains $\rtimes$ in the second position. The second column represents the possibility of a occurring in second position. As it has the form $(1, 0, 1, 0)$, a may occur in second position only if the first position is filled by $\rtimes$ or b. The remaining two columns encode the possible combinations for b and $\ltimes$ in second position.

Alternatively, we could have gone through the matrix row by row rather than column by column. In that case, the i-th row tells us whether the symbol mapped to index i can occur in first position depending on the second symbol.

Boolean matrices have several technical advantages which we will not discuss here. Quite simply, they constitute a very restricted and well-understood type of matrix, and consequently there are many efficient algorithms for working with them. For our purposes, this is yet another way of looking at strictly local grammars, but one that will gel exceptionally well with certain (probabilistic) extensions presented in Ch. 7.

2 Automata

Our initial discussion of strictly 2-local grammars in Ch. 3 was quick to point out that a grammar by itself only defines the set of well-formed structures — determining whether a specific input belongs to that set requires a recognizer. We picked scanners as the recognizers for strictly local grammars. But that does not mean that scanners are the only conceivable type of recognizer for these grammars. If one combines our previous discussion of graphs with our knowledge of scanners, one quickly discovers another type of recognizer: *automata*.

Automata can be viewed as yet another type of graph, one that is obtained by dropping the **No cycle** axiom for edge-labeled DAGs while adopting the prefix tree distinction between final and non-final nodes. Strictly local grammars correspond to a very specific automaton type.

Definition 5.2 (Strictly Local Automaton). A *strictly k-local automaton* over alphabet Σ is an edge-labeled graph $\langle V, F, E, \ell \rangle$ such that

- V is a set of k-grams over $\Sigma \cup \{\rtimes, \ltimes\})^k$,
- $F := \left\{ \sigma \ltimes^{k-1} \in V \mid \sigma \in \Sigma \cup \{\rtimes\} \right\}$ is the set of final nodes,
- $\langle u, v \rangle \in E$ iff $u = u_1 u_2 \cdots u_k$, $v = u_2 \cdots u_k \sigma$,
- for u and v as above, $\ell(\langle u, v \rangle) = \sigma$,

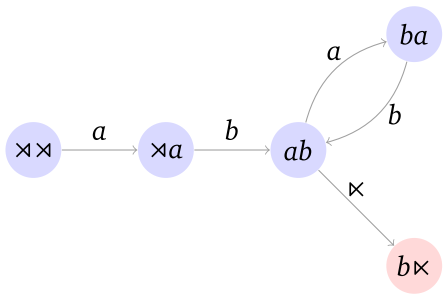
- there is a unique root $u \in V$, which must be $\rtimes^k$.

A string w is recognized by the automaton iff $\hat{w}$ is identical to a path from the root of the automaton to some final node.

This is one of the most convoluted definitions we have encountered so far, and it also distributes a very simple idea across several clauses. Intuitively, the nodes of a strictly local automaton correspond to strings that a corresponding scanner might see in its search window. An edge connects nodes u and v iff reading in the next symbol can change the content of the search window from u to v. We can take a strictly k-local scanner to always start out with a search window initialized to k left edges before reading in the first symbol of the string. Hence the root of automaton must have the very same shape. And because the last content of a scanner window always consists of a symbol followed by $k-1$ right edge markers, all final nodes must follow this pattern, too. Overall, then, an automaton is an abstract encoding of how the contents of the scanner window may change with each step while moving through the input strings from left to right.

Example 5.5 A Strictly 2-Local Automaton

The graph below shows a strictly 2-local automaton that recognizes $(ab)^+$.


$$\begin{tikzcd}
 & & & ba \arrow[dl, bend left, "b"] \\
\rtimes\rtimes \arrow[r, "a"] & \rtimes a \arrow[r, "b"] & ab \arrow[ur, bend left, "a"] \arrow[dr, "\ltimes"] & \\
 & & & b\ltimes
\end{tikzcd}$$

The direct correspondence between the nodes of the automaton and our canonical positive strictly 2-local grammar for $(ab)^+$ is apparent.

It is easy to infer that every positive strictly local grammar can be translated into a strictly local automaton that recognizes the same language. Proving the equivalence in the other direction is a little trickier, but since the proof has little additional insight to offer it is omitted here.

In sum, strictly local automata provide yet another characterization of the strictly local languages. At this point they seem to offer little advantage over positive strictly local grammars since they aren't more compact and we already have scanners as a recognition model that is easily implemented in Python. Just like Boolean matrices, they will be useful at a later point (Ch. 9) when we deal with a specific generalization of strictly local grammars.

3 Logic

Just like scanners, strictly local automata give us a view of strict locality that is rooted in serial recognition. The scanner reads an input string from left to right and eventually declares the input to be well-formed or ill-formed. The automaton slightly abstracts away from the actual processing by representing all scans of all well-formed strings in a single graph. But by virtue of encoding scans, the automaton is still about recognition. Strictly local grammars, on the other hand, completely forego recognition and just talk about licit substrings. Another perspective that directly focuses on well-formedness without specifying recognition is furnished by logical formulas.

A negative strictly local grammar lists all the k-grams that may not occur in a string. For instance, ${}^{-}G := \{\rtimes a, ab, ba, b\ltimes\}$ defines the complement of $(ab)^+$. Now let us say that a string satisfies the property p_G iff it contains the k-gram p of grammar G. So the ill-formed string bab would satisfy the properties ab, ba, and $b\ltimes$ of ${}^{-}G$, but not $\rtimes a$. More succinctly we could write this as $ab \wedge ba \wedge b\ltimes \wedge \neg\rtimes a$, where $\wedge$ means *and* while $\neg$ is short for *not*. Upon reflection, the strings that are well-formed with respect to ${}^{-}G$ must be exactly those for which the following holds:

$$\neg\rtimes a \wedge \neg ab \wedge \neg ba \wedge \neg b\ltimes$$

This statement is a formula of *propositional logic*, where each k-gram corresponds to a proposition, also called a *literal*. A literal is true of a string iff the string contains the k-gram. Full propositional logic would allow us to build logical formulas by stringing together propositions via various logical connectors. But for negative strictly local grammars, all we need is $\wedge$ and $\neg$. And the same holds in reverse: as long as a propositional formula only negates literals with $\neg$ and then connects them with $\wedge$, it can be recast as a negative strictly local grammar. So the class of negative strictly local grammars corresponds exactly to the set of formulas of propositional logic that are conjunctions of negative literals.

One advantage of this perspective is that it brings out the connection between positive and negative grammars more clearly. For the positive grammar ${}^{+}G := \{\rtimes a, ab, ba, b\ltimes\}$ a string is well-formed iff it contains $\rtimes a$ or ab or ba or $b\ltimes$, and nothing else. You might point out that a well-formed string must always contain $\rtimes a$ and $b\ltimes$ and ab, so we should not list them in a disjunction like that because that suggests that they are optional. But the obligatoriness of these bigrams can be inferred from what strings look like and hence need not be explicitly specified. So the statement might not be as informative as possible, but it is as informative as necessary.

Using $\vee$ as the propositional symbol for *or*, the disjunction part of the statement above can be recast as the formula below:

$$\rtimes a \vee ab \vee ba \vee b\ltimes$$

By DeMorgan's law, $\neg(a \vee b) = \neg a \wedge \neg b$. Using these laws, a pleasing equivalence obtains:

$$\begin{aligned}\neg(\rtimes a \vee (ab \vee (ba \vee b\ltimes))) &= \neg\rtimes a \wedge \neg(ab \vee (ba \vee b\ltimes))\\ &= \neg\rtimes a \wedge \neg ab \wedge \neg(ba \vee b\ltimes)\\ &= \neg\rtimes a \wedge \neg ab \wedge \neg ba \wedge \neg b\ltimes\end{aligned}$$

So the negation of a positive grammar yields a negative grammar. This is **not** the negative grammar that generates the same language as the positive one. Nonetheless the effect of negation reveals the deep logical connection between positive and negative grammars that our set-based translation captured only indirectly.

Notice that the disjunctive formula is not equivalent to the actual positive grammar. That's because the formula does not incorporate the *and nothing else* part. If we add this important additional restriction, we get a conjunction of two disjunctions that can be simplified quite a bit:

$$\begin{aligned}
&(\rtimes a \vee ab \vee ba \vee b\ltimes) \wedge \neg(\rtimes b \vee \rtimes\ltimes \vee aa \vee bb \vee a\ltimes) \\
=&(\rtimes a \vee ab \vee ba \vee b\ltimes) \wedge (\neg\rtimes b \wedge \neg\rtimes\ltimes \wedge \neg aa \wedge \neg bb \wedge \neg a\ltimes) \\
=&\neg\rtimes b \wedge \neg\rtimes\ltimes \wedge \neg aa \wedge \neg bb \wedge \neg a\ltimes
\end{aligned}$$

The second line is obtained from the first one using once again DeMorgan's law that $\neg(a \vee b) = \neg a \wedge \neg b$. The third line is not a universally valid inference of propositional logic, but it is true in this case: a string of a strictly 2-local language over alphabet $\{a, b\}$ that does not contain the bigrams $\rtimes b$, $\rtimes\ltimes$, aa, bb or $a\ltimes$ must necessarily contain $\rtimes a$, ab, ba, or $b\ltimes$. Since the truth of the second conjunct implies the truth of the first conjunct, the latter can be completely omitted, leaving us with the much shorter formula on the third line. But this formula is a conjunction of negative literals, which we just identified as the logical equivalent of negative strictly local grammars! What does this mean? It means that from a logical perspective, negative grammars are more basic than positive grammars because the logical formula characterizing a positive grammar must already include the negative grammar as part of the description (the second conjunct in the first line above).

Show that if we negate the formula including the *and nothing else* part, we still get a propositional formula characterizing some negative strictly local grammar.

The logical perspective has many other advantages that we will learn to appreciate in later chapters. In particular once we reach higher levels of expressivity that coincide with first-order logic, it becomes incredibly useful in modeling specific linguistic proposals.

4 Equivalent Characterizations of Strict Locality

This chapter has yielded a plethora of new views on strict locality. We now have a variety of descriptions of one and the same class of languages that we can switch between as we see fit.

Theorem 5.3. Let L be some string language over alphabet Σ. Then the following are equivalent:

- L is strictly k-local,
- L is closed under k-local substring substitution closure,
- L is generated by a positive strictly k-local language,
- L is generated by a negative strictly k-local language,
- L is recognized by a strictly k-local scanner,
- L is recognized by a strictly k-local automaton,
- L is the set of models for a disjunction of positive literals,
- L is the set of models for a conjunction of negative literals. ⌟