Lecture 3

Strictly Local Dependencies

The list phonology model turned out to be untenable on a computational level due to its inability to capture essential properties of natural language phonology. All its failures, however, can be taken to arise from the fact that it is not an analytic model — instead of treating phonological well-formedness as contingent on interacting properties of words, it simplifies it to an atomic and completely arbitrary notion with not internal rational. But perhaps the list model can be salvaged by combining it with a compositional view of well-formedness.

1 Phonological Processes as Lists

Let's return to the specific phenomenon of final devoicing for a moment. This process can be captured by straying not too far from our idea of phonology as a list, but without running into the issues the latter faces. In order to verify that a word satisfies word-final devoicing, it suffices to look at the very last phone; the word is phonologically well-formed only if the last sound is not voiced. Since every language has only a finite number of phones, we can write a list of phones that may occur in word-final position, and those that may not.

word-final	p	t	k	s	...
not word-final	b	d	g	z	...

This list will vary between languages, but it fully characterizes which sounds may occur at the end of a word.

But final devoicing is only one of myriads of local processes in phonology, so unless the idea above can easily be generalized to other local processes it is of very little practical use. Remember, treating phonology as a list of word pronunciations may not be elegant or enlightening, but at least it is guaranteed to always yield correct output forms.

Consider a simple process like nasal assimilation in English, where a nasal consonant (m, n, ŋ) that is followed by a plosive (p, b, t, d, k, g) changes into the nasal whose place of articulation is closest to that of the plosive. For instance, /bænk/ is pronounced bæŋk — the alveolar n turns into velar ŋ by virtue of k being velar. In this case we can also write a list, but we need more than two categories.

before p **or** b	m
before t **or** d	n
before k **or** g	ŋ

This list is not quite correct however, because plosives can be preceded by sounds besides nasals. So a more accurate listing would also need to contain vowels and other consonants.

before p **or** b	m	a	s	...
before t **or** d	n	a	s	...
before k **or** g	ŋ	a	s	...

It looks like we now have list-based accounts for two distinct processes, but how can we be sure that the two are actually similar accounts? For instance, the following list should strike you as a lot stranger than the previous two.

after a perfect number of sounds	p	t	k	z	...
after an imperfect number of sounds	b	d	g	s	...

You may also know perfect numbers as the topic of a great joke in Dave Gorman's Live at the Bloomsbury Theatre stand-up routine, which is available on Netflix.

A *perfect number* is a number that is identical to the sum of its remainder-free divisors, e.g. $6 = 1 + 2 + 3$ or $28 = 1 + 2 + 4 + 7 + 14$. The list above, then, tells us that certain consonants may be voiced iff their position in the word is a perfect number. Of course no natural language behaves like that. Consequently we need an explanation as to why the first two kinds of lists are phonologically natural, whereas the third one is not.

The most obvious answer is that there is a difference in how these lists pick out positions. The first two regulate the shape of a sound depending on what it is followed by. The nasalization process can easily be compiled out into a list of well-formed sequences of two sounds: mp, mb, nt, nd, ŋk, ŋg, ap, ab, and so on. We call such sequences of two sounds *bigrams*. Crucially the list does not contain bigrams like nk, so a word where n is followed by k is not phonologically well-formed. The same compiling-out procedure can be used for word-final devoicing if we use the special symbol ⋉ to mark the end of the word. The list contains the bigrams p⋉, t⋉, k⋉, s⋉, but not b⋉, d⋉, g⋉, or z⋉. Thus the first two processes share the property that they can be described via a finite list of bigrams.

The hypothetical perfect-number phenomenon, on the other hand, cannot be described in terms of bigrams. A sequence like ɪt is licit in the string tɹænzɪt, but illicit in ædmɪt. The way sounds are regulated no longer depends on adjacency to a specific sound but rather a highly abstract counting mechanism that needs to take the whole word into account. So one idea we may explore for now is that all phonological processes can be captured via finite lists of bigrams. In other words, phonology is a bigram grammar.

2 Bigram Grammars and Recognizers

2.1 The Intuition Behind Bigram Grammars

A *bigram grammar* is a just finite set of bigrams. Each bigram encodes a licit sequence of two sounds. So a bigram grammar *G* considers a string *w* well-formed iff every sequence of two adjacent symbols in *w* is a bigram of *G*. We refer to these sequences as bigrams, too. In other words, a string is well-formed iff it consists only of bigrams that are licensed by the grammar. We also say that the bigram grammar *generates* this string.

Example 3.1 Strings licensed by a small bigram grammar

Suppose grammar G contains the bigrams $\rtimes a$, ab, $b\ltimes$, where $\rtimes$ and $\ltimes$ mark the left and right edge of the string, respectively. Now consider the string ab, which we may also write $\rtimes ab\ltimes$ for the sake of explicitness. This string consists of the bigrams $\rtimes a$, ab, and $b\ltimes$. Since all of those bigrams are part of G, the string is well-formed.

The minimally different ba, on the other hand, is built from the bigrams $\rtimes b$, ba, and $a\ltimes$. Not a single one of those bigrams is contained in G, and consequently G does not generate this string.

The string abb is not generated by G either. Even though three of its bigrams are part of G — namely $\rtimes a$, ab, and $b\ltimes$ — its fourth bigram bb is not among G's bigrams. So even though 75% of this string's bigrams are well-formed, it is still not generated by the grammar. A single mistake is enough to render it illicit.

Notice that G only generates the finite language $\{ab\}$. Suppose that we extend G so that it also generates $abab$. This requires adding the bigram ba to G. But this allows G not only to generate the string $abab$, but also $ababab$, $abababab$, and so on. The addition of a single bigram pushed the language generated by G from a finite one to an infinite one. This also tells us that not all finite languages can be generated by bigram languages. For some finite languages L, a bigram grammar that generates all strings of L must generate an infinite superset of L.

2.2 Recognition via a Bigram Scanner

A subtle point that is easy to lose sight of is that the grammar is just a specification of which strings are well-formed, it does not include a procedure for verifying whether a given string is actually well-formed. That is easy to see if we think about how we would implement a bigram grammar in Python. Given the definition above, a bigram grammar is simply a set of bigrams. This is straight-forwardly translated into Python.

```
1 bigram_grammar = set(['La', 'ab', 'ba', 'bR'])
```

We could omit `set()` from the definition and thus treat bigram grammars as lists instead of sets. This will have no effect on the code in the rest of the chapter.

Clearly the definition of a variable is not enough to produce a useful program. What we need is a mechanism that takes the grammar as a parameter and then determines if a given string is generated by the grammar. Such a mechanism is called a *recognizer* as it recognizes whether a string is well-formed.

There's many different ways one may construct a recognizer, but for a bigram grammar the simplest model is that of a *scanner*. The scanner has a window that is exactly two symbols wide. It moves this window through the string from left to right, and at each step it checks whether the sequence of symbols it sees in its window is part of the bigram grammar (see Fig. 3.1 on the following page for an illustration). If the answer is no, it rejects the string. Otherwise it moves the window one symbol further to the right. If the window cannot be moved further to the right, the scanner accepts the string as well-formed.

Notice that the scanner as described above queries the grammar during every step it takes through the string. This is somewhat wasteful — after all, if we've already verified once that ab is a licit bigram, why should we have to do so again at a later point. An alternative is to first move the scan window through the entire string while

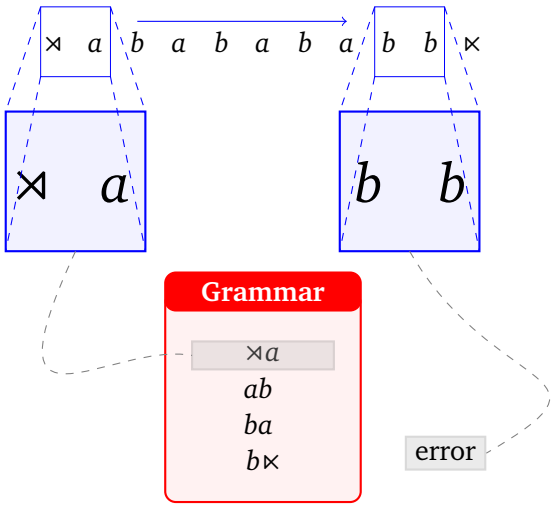

Figure 3.1: Scanner for a bigram grammar

keeping track of the bigrams one encounters. That way one obtains the set of all bigrams in the string (note that no bigram occurs more than once in this set). Once the scanning steps have concluded, we simply check whether the set of bigrams is a subset of the grammar. If that is not the case, then the string contains some bigram that is not part of the grammar, wherefore we are dealing with an ill-formed string. Otherwise the string is well-formed. A Python implementation of this approach is given in Listing 3.1 on the next page.

2.3 Positive and Negative Bigram Grammars

Contemporary phonology does not think of phenomena like final devoicing as a process but rather as a constraint: certain sounds are banned from occurring word-finally. This can be represented as below using an Optimality Theory (OT) tableau.

/ʀɑːd/	*[+voice]⋉
[ʀɑːd]	!*
[ʀɑːt]	

Similarly, assimilation is nowadays analyzed as the result of a ban against certain sound sequences.

This constraint-based perspective can be accommodated in a very natural way in the bigram framework. Right now, we interpret the bigrams of a grammar as licit sequences, that is, a string is well-formed iff all its bigrams are licensed by the grammar. But we could also view things the other way round such that the grammar's bigrams list illicit sequences. Each bigram then encodes a specific constraint of the grammar, and consequently a string is well-formed iff none of its bigrams are listed by the grammar. We call these two types of bigram grammars *positive* and *negative*, respectively.

```
def augment_string(w):
    """Add edge markers to string."""
    return 'L' + w + 'R'


def string_to_bigrams(w):
    """Convert string into non-repeating list of bigrams."""
    bigram_list = []
    for i in range(len(w)-1):
        current_bigram = w[i] + w[i+1]
        if current_bigram not in bigram_list:
            bigram_list.append(current_bigram)
    return bigram_list


def bigram_comparison(grammar, bigram_list):
    """Check whether a bigram grammar subsumes a list of bigrams."""
    return set(bigram_list).issubset(grammar)


def bigram_scanner(grammar, w):
    """Recognizer for bigram grammar given string w."""
    if bigram_comparison(grammar, string_to_bigrams(augment_string(w))):
        return True
    else:
        return False
```

```
>>> test_grammar = ['La', 'ab', 'ba', 'bR']
>>> string_to_bigrams('ababab')
['ab', 'ba']
>>> string_to_bigrams('ababa')
['ab', 'ba']
>>> string_to_bigrams(augment_string('ababab'))
['La', 'ab', 'ba', 'bR']
>>> string_to_bigrams(augment_string('ababa'))
['La', 'ab', 'ba', 'aR']
>>> bigram_scanner(test_grammar, 'ababab')
True
>>> bigram_scanner(test_grammar, 'ababa')
False
```

Listing 3.1: Python implementation of a set-based scanner

Example 3.2 A negative bigram grammar

Consider once more the bigram grammar G from the beginning of example 3.1, which consisted of the bigrams $\rtimes a$, ab, and $b\ltimes$. As a positive bigram grammar, G generates only the string ab. As a negative bigram grammar, it generates all strings that

- do not start with an a, and
- do not end in a b, and
- do not contain an a followed by b.

Notice that this set is infinite, as it contains the strings ba, bba, $bbba$, and so on.

Now that we have two different ways of interpreting bigram grammars, the obvious question is whether the two differ in any respect. In particular, is one more powerful than the other? This is a challenging question, after all there are infinitely many grammars of each type, so we can't just compare them one by one. What we need is a way to talk about these two infinite classes of grammars in a precise and rigorous manner. We need math.

3 Analyzing Bigram Grammars

3.1 Equivalence of Positive and Negative Bigram Grammars

Mathematics and its tools require a certain amount of precision and explicitness, with respect to concepts as well as notation. This is not too different from programming, where you have to use the correct syntax, define your variables and functions, and so on. In many respects math is actually less verbose and pedantic because it allows us to abstract away from details that aren't relevant for our purposes (for instance what kind of data structure to use for sets). While we have been fairly explicit about bigram grammars, it is nonetheless prudent to make all our terms fully explicit before moving on.

Definition 3.1 (String languages). A *string alphabet* Σ is a finite, non-empty set of symbols. We usually drop the "string" part and simply speak of alphabets. A *string* over alphabet Σ is a finite sequence of symbols drawn from Σ. A string is of *length* n iff it is a member of

$$\Sigma^n := \underbrace{\Sigma \times \cdots \times \Sigma}_{n \text{ times}}.$$

The *empty string* ε is the unique string whose length is 0. Given two strings u and v, we denote by $u \cdot v$ the *concatenation* of the two, which is the string uv. It holds for every string u that $u \cdot \varepsilon = \varepsilon \cdot u = u$. We denote by $\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$ the set of all finite strings that can be built from symbols in Σ. A set L is a *string language* over Σ iff $L \subseteq \Sigma^*$, i.e. L is a subset of Σ^*.

Some scholars would consider this definition of string still too sloppy since we did not specify what we mean by a sequence of symbols (we did not define sequences or the cross-product notation for sets used above). For our purposes the definition is good enough, but if you're curious here's a more technical one: a string over Σ is an initial

subset of the natural numbers with a labeling function ℓ from $\mathbb{N}$ to Σ. This definition exploits the fact that the natural numbers are string-like (1 before 2, 2 before 3, 3 before 4) and then simply labels the first n natural numbers with symbols drawn from Σ.

Quite generally, mathematical objects can be defined in many different ways. This is not a pointless exercise; very often a good definition is the most important step towards discovering new facts about an object. In fact, if you can think of only one way to define an object, that is a clear-cut sign that you do not understand the object very well. Keenan and Moss (2012:10) express this succinctly yet with the appropriate force:

> If you can't say something two ways you can't say it at all.

This credo may seem odd to linguists, who usually take a strongly realist stance according to which the grammar formalism fully specifies the grammar, even down to notation. For example, privative (= single-valued) and binary features are regarded as vastly different objects that make very different claims about phonology. Yet we will see at a later point that they are but two definitions of the same computational object. That does not rule out that one of the two is a more useful way of thinking about phonology, but that is a matter of epistemology rather than ontology.

Returning from our brief methodological excursus, we now have to define bigrams, which in turn will allows us to define bigram grammars.

Definition 3.2 (Bigrams). Given a string w over alphabet Σ, its *augmented* counterpart $\hat{w} := \rtimes \cdot w \cdot \ltimes$ is obtained by adding the left and right edge markers $\rtimes$ and $\ltimes$ to w, where $\rtimes$ and $\ltimes$ are distinguished symbols not contained in Σ. Furthermore, 2-grams$(w) := \{ab \mid \exists u, v \in \Sigma^* \text{ s.t. } u \cdot ab \cdot v = \hat{w}\}$ denotes the set of *bigrams* over $\hat{w}$, i.e. the smallest set that contains all substrings of $\hat{w}$ that consist of exactly 2 symbols.

Definition 3.3 (Bigram Grammar). A *bigram grammar* G over alphabet Σ is a finite set of bigrams over $\Sigma \cup \{\rtimes, \ltimes\}$. If G is a *positive bigram grammar* (denoted ${}^{+}G$), then it generates the language $L(G) := \{w \mid 2\text{-grams}(w) \subseteq G\}$. If G is a *negative bigram grammar* (denoted ${}^{-}G$), then it generates the language $L(G) := \{w \mid 2\text{-grams}(w) \cap G = \emptyset\}$.

With all these definitions under our belt, we can finally move on to producing a new insight: positive and negative bigram grammars are equally powerful. That is to say, if some language is generated by a positive bigram grammar, then it can also be generated by some negative bigram grammar, and the other way round.

Theorem 3.4. The class of languages that are generated by positive bigram grammars is exactly the class of languages that are generated by negative bigram grammars.

In order to show that this theorem is indeed correct, we establish two simpler propositions — called lemmata — which jointly imply the theorem.

Lemma 3.5. For every positive bigram grammar there is a negative bigram grammar that generates the same language.

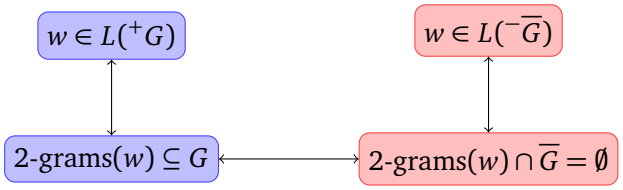

Figure 3.2: Reasoning chain for the equivalence of positive and negative bigram grammars

Proof. Let ^{+}G be a positive bigram grammar. We show that $^{-}\overline{G}$ defines the same language as ^{+}G, where $\overline{G}$ consists of all bigrams over $\Sigma \cup \{\rtimes, \ltimes\}$ that are not contained by G.

Pick some arbitrary string $w \in L(^{+}G)$. By definition, every bigram of w is a member of G, which immediately implies that no element of 2-grams(w) is contained in $\overline{G}$. But if none of the bigrams of w belong to $\overline{G}$, then 2-grams$(w) \cap \overline{G} = \emptyset$, wherefore $w \in L(^{-}\overline{G})$. Since w was arbitrary, this result holds for every string generated by ^{+}G, establishing $L(^{+}G) \subseteq L(^{-}\overline{G})$.

The same argument can be applied in the other direction to show $L(^{+}G) \supseteq L(^{-}\overline{G})$, wherefore $L(^{+}G) = L(^{-}\overline{G})$. □

This is a so-called *constructive* proof: we do not just show that an equivalent negative bigram grammar exists, we also explain how this grammar can be constructed from the positive bigram grammar. Constructive proofs are the most useful kind of proof because they provide procedures and strategies that can be implemented and run automatically.

In the case at hand, all we have to do in order to construct an equivalent negative bigram grammar is to take the set-theoretic complement of the positive bigram grammar. After all, the set of all bigrams over Σ is given by $\Sigma \times \Sigma$, and the positive bigram grammar $^{+}(G)$ is some subset thereof (*modulo* edge markers, which are treated as part of Σ here to avoid notational clutter). Bigrams that belong to G may occur in a string, bigrams that do not must not. So the bigrams not belonging to G are the illicit bigrams, which means those are exactly the bigrams the equivalent negative bigram grammar must contain. In one sentence: taking the complement of G is like taking its negation, and the switch from a positive grammar to a negative one undoes this negation.

So now we know that the negative bigram grammars are at least as powerful as the positive bigram grammars since the latter can be translated into the former. It only remains for us to show that the same holds in the other direction.

Lemma 3.6. For every negative bigram grammar there is a positive bigram grammar that generates the same language. ⌟

Proof. Left as an exercise to the reader. □

Figure 3.2 gives a pictorial representation of the implications used in these proofs.

3.2 The How and Why of Proofs

Proofs are a difficult art at every level of expertise. For the beginner, the biggest challenge is often to sort out their train of thought and present it in a clear manner: where do I start, and how do I proceed from there step by step to reach the conclusion?

There are no clear-cut rules here, but it is often helpful to break up the problem into smaller ones, as we did with Thm. 3.4. If two sets A and B need to be shown to be equivalent, one can first prove $A \subseteq B$ and then $B \subseteq A$. And a statement of the form "ϕ iff ψ" can be broken up into "ϕ entails ψ" and "ψ entails ϕ". We will encounter several basic proof techniques throughout the course (proof by induction, indirect proofs), but don't worry too much about specific techniques for now. Instead, make sure to work through the proofs we discuss multiple times until you understand how they work. Always try to answer the following questions for yourself:

- Why is the initial assumption valid?
- How does each conclusion follow from the previous one?
- Why does the final conclusion show that the theorem/lemma is correct?

You may wonder why we need proofs in the first place. Linguists don't work with proofs yet they have discovered a lot of interesting things about language. Programmers don't have much use for proofs either. There's several answers, the simplest one being that some questions can only be addressed conclusively via proofs. Consider the following alternative to the proof of Lem. 3.5: we could have implemented the translation procedure from positive to negative grammars and then tested it on a large sample of positive bigram grammars (at least several thousand). Eventually the program would have told us that the negative grammars generate the same string languages as their positive counterparts. So if the conversion works correctly on every single one of thousands of positive bigram grammars, isn't that enough to posit that positive and negative bigram grammars are interchangeable?

The answer is No. First of all, checking that two grammars generate the same language is hardly trivial if both languages are infinite (we can't just compare all their respective members). More importantly, though, the thousands of test grammars may accidentally share a special property that is essential for the correctness of the conversion. This is a real risk if all those test grammars were automatically generated by a script because true randomness is very hard to achieve with computers, if not impossible. While experiments and simulations have their place — they are a valid last resort where proofs are hard to come by — a proof is always the preferred solution where possible.

Proofs are preferred not only because they are safe from the pitfalls of simulations, but also because they provide genuine insight. In fact, proofs are often more important and enlightening than the theorems they establish. Testing the correctness of the bigram conversion via automated experiments could at best show us whether the procedure is correct (if there's only finitely many cases to test), but it does not tell us **why**. A proof is an explicit record of how certain properties entail others. In the case at hand, it is the close connection between set-theoretic complementation and the switch from positive to negative grammars that does all the work. Notice all the properties of bigram grammars that the proof does not depend on: that bigrams are strings, that bigrams consist of exactly two symbols, and that bigram grammars are finite. Yet these properties necessarily hold during any test procedure, so we would not be able to tell which one of them is a prerequisite for the correctness of the conversion. Thanks to the proof, we know which properties matter, which in turn might come in handy in the study of other formalisms.

In a few lines, a proof can establish unassailable truths, show us why they hold, but also save us hours of work compared to running simulations. So even though you

may initially find them hard and time consuming, proofs are actually the tool of choice for the lazy scientist.

4 Linguistic Evaluation

The bigram grammar model improves on the list phonology model in various respects. Variation across languages is now much more restricted because phonology is just a collection of highly local constraints (negative bigram grammars) or permissions (positive bigram grammars). Bigram grammars also account for linguistic creativity, i.e. that speakers can form new words according to the rules of their own language, and that they recognize whether nonce words are well-formed. As a consequence, they do not predict that the lexicon is finite, either. Whether the lexicon is finite or infinite is immaterial for bigram grammars since they determine well-formedness in a compositional manner. As long as each word is finite, its grammaticality can easily be determined via a bigram scanner.

Bigram grammars do still exhibit egalitarianism. Since a grammar is a list of bigrams, all these bigrams should have the same status and there should be no distictions between easy and difficult processes. For example, we have seen that the language $(ab)^+$ can be generated by a bigram language. If we identify a with C and b with V, we get the word template for languages that only allow CV syllables. Clearly we could just as well have mapped a to V and b to C, generating a language where all words consist of VC syllables. While many languages can have VC syllables, all languages with VC syllables also allow for CV syllables. This implicational universal is completely unexpected if phonology is just a bigram grammar.

Isolationism is also a problem as processes still cannot apply across words. For *phone bill*, the phonological representation is presumably something like $\rtimes \text{foʊn} \ltimes \rtimes \text{bɪl} \ltimes$. Since n and b are not adjacent, a negative bigram grammar with *nb* does not have the desired result. This could be fixed with a bigger search domain, though, which also seems to be necessary for slightly less local processes like intervocalic voicing, where we want to block a voiceless sound only if it occurs between two vowels.

Overall bigram grammars are doing very well on a conceptual level and are easy to implement, but they are not expressive enough for English. In the next chapter we will see how we can keep all their attractive properties while increasing their power to a more adequate level.