# Unit 2

# Implementing Phonology as a List

Linguists distinguish many different aspects of language:

**Phonetics** the physical properties or speech, in particular the physiological production of sounds (*articulatory phonetics*) and their acoustic properties (*acoustic phonetics*)

**Phonology** the rules regulating the sounds of a language: which sounds are part of a given language's inventory, in which positions may they occur, how are sounds affected by the presence of other sounds, which syllables in a word are stressed, and so on

**Morphology** the rules regulating the structure of words, in particular how a words form can change depending on certain features such as person or number (*inflectional morphology*) and how new words can be built from other words (*derivational morphology*)

**Syntax** the rules regulating the structure of sentences, e.g. word order and morphosyntactic dependencies (person, gender, number agreement)

**Semantics** the formal study of the meaning of words (*lexical semantics*) and how the logical meaning of a sentence is derived from the meaning of its words (*compositional semantics*)

**Pragmatics** the study of how a sentence's intended meaning arises from the interaction of its logical meaning and the discourse context

All these areas have been looked at by computational linguists, but we will start with phonology, for several reasons. First, phonetics involves a lot of non-discrete math that is fairly demanding for the uninitiated. Similarly, syntax, semantics and pragmatics use fairly complicated linguistic formalisms. This leaves us with phonology and morphology, and since the two are very similar from a computational perspective (many computational models even handle them both at the same time), we pick the one that linguistics students usually have had greater exposure to: phonology.

## 1 A Simple Phonology Problem

Word-final devoicing is a process that has been extensively studied by phonologists. As its name implies, final devoicing turns voiced consonants ($/\mathrm{b}/$, $/\mathrm{d}/$, $/\mathrm{g}/$, $/\mathrm{z}/$, $\ldots$) that occur at the end of a word into their voiceless counterparts (/p/, /t/, /k/, /s/, . . . ). It usually applies only to a proper subclass of a given language's full inventory of voiced sounds. Final devoicing is attested in a rich variety of languages, from Indo-European ones like Catalan (Romance), German (Germanic), and Russian (Slavic) to Turkish (Turkic, Altaic) and Wolof (Senegambian, Niger-Congo).

| **Language** | **Voiced** | **Devoiced** |
|---:|---|---|
| Catalan | *grize* 'gray (F)' | *gris* 'gray (M)' |
| German | *räder* 'bikes' | *rat* 'bike' |
| Russian | *kniga* 'book (NOM.SG.) | *knik* 'book (GEN.PL.)' |
| Turkish | *sarabi* 'wine (ACC.SG.)' | *sarap* 'wine (NOM.SG.)' |
| Wolof | does anybody know | the data? |

What kind of computational resources must a native speaker possess that has successfully learned this process for their language and can apply it correctly during speech? As innocent as this question may seem, it is actually very difficult to answer because it is unclear what kind of computational process underlies word-final devoicing.

The way it was described above — which is the standard view among phonologists — it is a process that takes a word as an input and returns an output where any word-final voiced consonants have been devoiced. That is a complex idea that involves three distinct components:

1. an input form,
2. an output form,
3. a process translating the former into the latter.

It is far from obvious that this is indeed what speakers are doing; all we can tell for sure is that speakers produce the correct output forms. So rather than jumping immediately into the deep waters of sophisticated phonological machinery, let's see what the **simplest empirically adequate** solution might be. It might well turn out that speakers are actually doing something more complicated, but at least we will have a better understanding of the problem and a computational baseline that we can compare speakers' behavior to.

Arguably the simplest conceivable solution is that there are no phonological processes at all, speakers simply memorize all the output forms.

**List Phonology Model** A speaker's knowledge of phonology is fully captured by a list of pronunciations.

This would reduce phonology to a long list of fully inflected words and speakers simply pick that item from the list that they want to pronounce. It explains speakers' ability to produce the correct output form purely via memorization, no computational machinery is required beyond a mechanism for storing and retrieving phonetic strings. Such a solution is certainly simple, and it has been used in NLP in the past. After all it only require a few people to go through a dictionary of English and write down the pronunciation for each word and all its fully inflected variants (e.g. *go*, *goes*, *went*, *gone*, *going*). Simple as it may be, though, the important question for us whether it is empirically adequate.

# 2 Storage and Retrieval Speed

## 2.1 Lists, Hash Tables, and Python Dictionaries

Let's first think about whether speakers could possibly have such a list stored somewhere in their brain. Psychologists and neuroscientists still know very little about how the brain stores information, so for the sake of argument we will look at this through the lens of Python data structures. Python has lists as a data type, so we could instantiate a phonology list for a given language, say German.

This list uses the sci.lang ASCII transliteration of IPA, which is easier to type and can be used on systems without unicode support.

```
german_phonology = ['Ra:t', 'Re:dV"', 'blint', 'blind@',\
                    'iC', 'du:', 'EV"', 'si:', 'Es']
```

The problem is that items in a list are not always easy to retrieve. If you know the index of an item, you can immediately access it via said index.

```
german_phonology[4]
```

But with long lists you do not know which position a specific item occupies. In order to retrieve this item, then, one has to search through the entire list until one finds the correct item. So the longer the list, the longer it takes to find an item towards its end. If we simply search through the list from left to right, finding the $n$-th item takes $n$ steps. There are more efficient methods such as the binary search algorithm that we looked at in Lecture 1, but with those the search time still increases with the size of the list, just not as rapidly.

This does not at all seem plausible from a psycholinguistic perspective. Experiments have shown that frequent words are retrieved more quickly than rarer ones, but the retrieval speed is independent of overall vocabulary size (see **Jurasfky03** and references therein). Having a larger vocabulary does not mean that it takes you longer to retrieve specific words, but this is what the current list implementation predicts. But this might be just a problem at the algorithmic level — something that could be fixed by changing to a different data structure while maintaining the claim that on the computational level phonology is accurately described as a process of memorization and retrieval of word pronunciations.

Quick access and storage of information is crucial in various applications, and thus it is no surprise that computer scientists have developed data structures which allow any given item to be stored and retrieved in constant time. A particularly popular one is the *hash table*. The basic idea behind a hash table is that the problem with lists is the arbitrary and volatile connection between an item in a list and its index. Not only is there no particular reason why *blint* has index 2 and *ea* index 1, their indices can also change, e.g. if elements are added to the list or if the entries in the list are reordered. In a hash table, each entry has a key that uniquely identifies it, and this key is translated into the correct index via a *hash function*. Hence the order of elements in a hash table is irrelevant for their retrieval. As long as we have the key, lookup takes only the amount of time required to compute the index from the key. A smart hash function can do this in constant time for any given key irrespective of how many keys there are or how long they are — the conversion from indices to keys always takes a

This is a very simplified presentation of hash tables, see Sec. 1.5 of Dasgupta, Papadimitriou & Vazirani (2006) for some very useful details.

fixed number of steps. It follows that lookup in hash table takes constant time, much more efficient than the linear time lookup in a list.

### Background Asymptotic Notation

Computer scientists measure performance in terms of *asymptotic worst-case complexity*. That is to say, how long does it take to solve a given task if everything goes wrong that could go wrong within the specified parameters of the problem (e.g. the item we are looking for is at the very end of the list) and if there are no restrictions on the size of the input (e.g. there is no limit on how many items a list may contain).

Asymptotic worst-case complexity is expressed via the "Big $O$" notation. If an algorithm has time complexity $O(n^3)$, this means that it takes at most $n^3$ steps for it to find a solution (or determine that there is no solution), where $n$ is the size of the input problem (e.g. the number of items in a list). The Big $O$ notation ignores parameters whose impact on complexity diminishes as the size of the problem approaches infinity. For instance, when running a search algorithm you wrote in Python there is a fixed cost for reading in the actual code you wrote. But since this only takes a fixed amount of steps, say 5, that cost is completely negligible once we deal with lists containing millions of items — it does not matter whether your computer has to do 3 million computations or 3 million and 5.

Without going into too much detail, we can rank an algorithm's efficiency according to which of the following classes it belongs to:

**constant** solving the problem always takes a fixed number of steps
e.g. $O(5) = O(1)$

**logarithmic** the amount of time is logarithmically bounded by the size of the input
e.g. $O(2\log n + 5) = O(2\log n) = O(\log n)$

**linear** the amount of time scales linearly with the size of the input
e.g. $O(5n\log n + 5) = O(5n) = O(n)$

**polynomial** the amount of time is bounded by a polynomial of the input (in simpler terms: a function where $n$ has an exponent)
e.g. $O(8n^4 + 3n^2 + 5n\log n + 5) = O(8n^4) = O(n^4)$

**exponential** the amount of time grows exponentially with input size
e.g. $O(4^n + 2^n + 8n^4 + 3n^2 + 5n\log n + 5) = O(4^n)$

**factorial** the amount of time grows factorially with the input, where the factorial of $n$ is $n! := n \times (n-1) \times \cdots \times 2 \times 1$ (for instance, $4! = 4 \times 3 \times 2 \times 1$)
e.g. $O(8n! \times n^{10^{100}}) = O(8n!) = O(n!)$

In practice, problems that can be solved in polynomial time are considered efficiently solvable. In the worst case, you may have to wait a few years for hardware to catch up, but since computing power doubles approximately every two years (*Moore's law*), you will eventually catch up with the problem. The difficulty of exponential problems, on the other hand, grows so fast with input size that it vastly outpaces technological progress and how many resources you can throw at it.

Python dictionaries are an implementation of hash tables, so we can improve the performance of our list model simply by switching to a different data structure. That requires only minor changes in the code. First, we change to curly braces to signal that we are constructing a dictionary. Then each item is assigned a unique key, for which we choose the underlying form posited by phonologists.

```
    german_phonology = {'Ra:d':'Ra:t', 'Re:d@R':'Re:dV"',\
        'blind':'blint', 'blind@':'blind@',\
        'iC':'iC', 'du:':'du:', 'ER':'EV"',\
        'si:':'si:', 'Es':'Es'}
```

But now that lookup is constant time thanks to the use of a dictionary we outperform the psycholinguistic reality that not all words are equally easy to retrieve. It is conceivable, though, that humans use a hash function that prioritizes quick retrieval of common items to the detriment of less frequent ones. After all, a hash function that takes one step on very frequent items and five steps on rare ones might be preferable to one that takes 3 steps for all of them. From the perspective of computational complexity the two are the same because $O(1) = O(3) = O(5)$, but that does not preclude that one is noticeably faster than the other in practice, at least when run on *wetware*, i.e. the human brain. Or maybe frequent words are stored in a faster type of memory (just like you might have your OS installed on an SSD while keeping your media files on a conventional hard drive). So let's graciously assume that this aspect of human performance can be modeled (and possibly explained) by the phonology list approach.

## 2.2 Memory Usage

The next question one might ask is whether it is feasible that humans do indeed store all words in their fully inflected forms. Remember that this is a crucial assumption for the model under discussion, which treats phonology as nothing more than an inventory of the well-formed outputs. We can do some rough estimates regarding memory usage.

Let's make the conservative estimate that the speaker's language has 30 different phones (ignoring length differences). So each sound in a word can be represented by a number between 0 and 29. For instance, *Ra:t* may correspond to *21 5 13*. How many bits does it take to store one of these numbers?

**Background** **Binary Numbers**

In daily life we use the *decimal number system*. In this system, 53 represents a number that we can analyze as $50 + 3 = (5 \times 10) + (3 \times 1) = (5 \times 10^1) + (3 \times 10^0)$. So in the decimal system, the digit in the $n$-th position acts as a multiplier for $10^{n-1}$, and we just sum the values encoded by each position of the number. That's why it is called the decimal system, 10 acts as the base with exponent $n - 1$.

This system can of course be used with any other natural number as the base. The simplest case is the *binary number system*, where the base is 2. The number 5, for instance, has the binary representation 101 since $5 = (1 \times 2^2) + (0 \times 2^1) = (1 \times 2^0)$. And 53 is written 110101 (you do the math!). Binary representations are important because each position has only two values, which can easily be instantiated in a physical system, for instance via the on and off states of a switch. That's exactly what computers do, and that's why computers "only know zeros and ones", as the tired old saying goes. A binary digit is also called a *bit*, which you probably know as the smallest unit of information. So if you know how binary numbers work, you can actually calculate memory usage.

The highest number we need for our example is 29 (that way we can order the 30 phones consecutively from 0 to 29). In binary this is represented as 11101, so each number (= sound) requires at most 5 bits. We will take 3 as the average number of bits that is needed per phone in an aribtrary word (assuming that some sounds are more common than others, a smart coding strategy could push this down quite a bit). We furthermore assume that the speaker's vocabulary contains about 20,000 words, which expands to about 50,000 fully inflected word forms, with an average word length of 8. Hence we have 50,000 words that take $8 \times 3$ bits on average. Overall then, we need $\frac{50000 \times 8 \times 3}{8 \times 1024} \approx 146$ Kilobytes to store all these forms. To put this number into perspective, the collected works of Shakespeare take up about 25 times as much (2 to 5 Megabytes depending on character encoding) and fit on 1500 densely printed pages. The phonology list model, then, would take up about 60 pages, which doesn't seem too outrageous — humans are certainly capable of memorizing longer passages.

The Vedic Sanskrit corpus, for instance, consists of over 1000 pages and despite being over 3000 years old, had not been written down until the first century BC. For over a thousand years, it was preserved by a purely oral tradition that relied exclusively on memorization. In fact, the oral tradition continued to dominate for the first millenium AD.

Things do not get much worse as the number of allophones increases. The language Ubykh (extinct, Northwestern Caucasian) has 84 consonant phonemes, but only 2 phonemic vowels. Ignoring allophones, we need numbers between 0 and 85 to identify each phoneme, which implies an increase in the maximum number of bits per sound to 7, so with an average of 6 bits (a very pessimistic estimate) we would end up using 292 Kilobytes, or 111 pages. Even if every phoneme has two allophones and no phone is an allophone for more than one phoneme, we have at most 172 different sounds, which fits comfortably within 8 bits; assuming an average of 7 bits per phone, the whole phonology list would take up a mere 341 Kilobytes.

As you can see, the number of representable phones increases exponentially with the number of bits, and the overall memory usage depends only on the number of bits. So a linear increase in the size of the sound inventory causes only a logarithmic increase in the amount of memory consumed by the phonology list. Compared to the size of the lexicon and the average word length, the number of sounds in a language is a negligible factor.

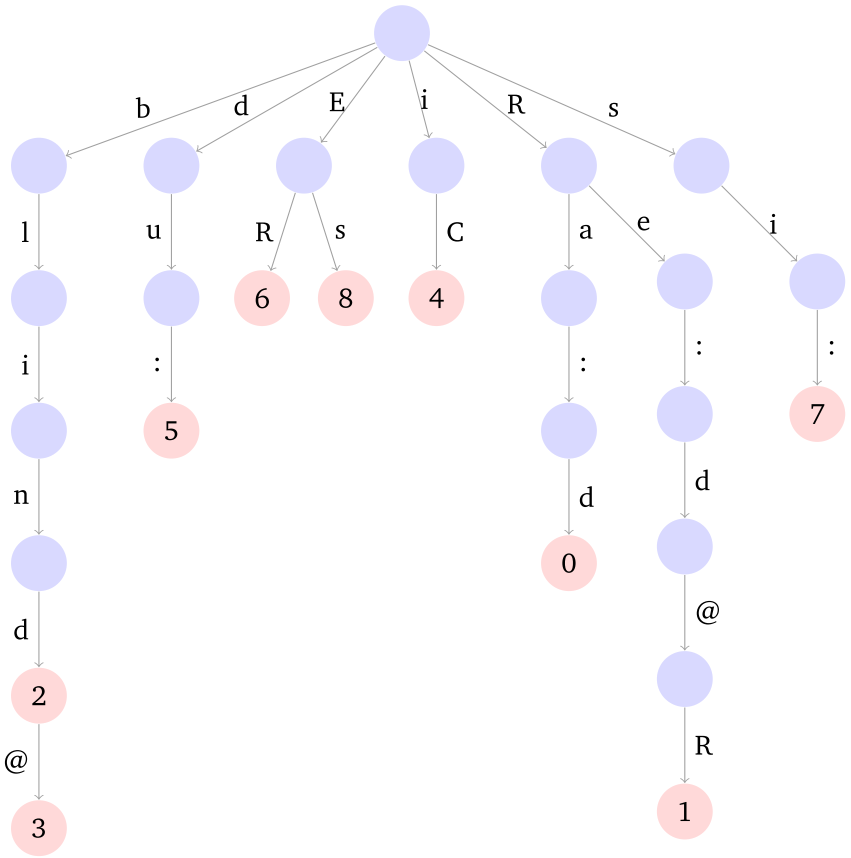
## 2.3 Prefix Trees

The storage requirements we just computed only take into account the output forms, not the keys and the (negligible) hash function, which are also an essential part of a hash table. With no further optimizations, the keys will double the memory usage. The keys we use, though, have the property that many of them are very similar. For instance, the keys *blind* and *blind@* are almost exactly the same. The latter is an extension of the former, or the other way round, *blind* is a *prefix* of *blind@*.

**Definition 2.1 (Prefix).** Given strings $u$ and $w$, $u$ is a *prefix* of $w$ iff there is some (possibly empty) string $v$ such that $w$ is the concatenation of $u$ and $v$.

Keep in mind that this usage of prefix has nothing to do with its linguistic meaning as an affix that precedes the stem of a word. Furthermore, since $v$ in the definition is allowed to be empty, i.e. a string that contains no symbols whatsoever, every string is its own prefix. For instance, the prefixes of *blinde* are *b*, *bl*, *bli*, *blin*, *blind*, and *blinde* itself.

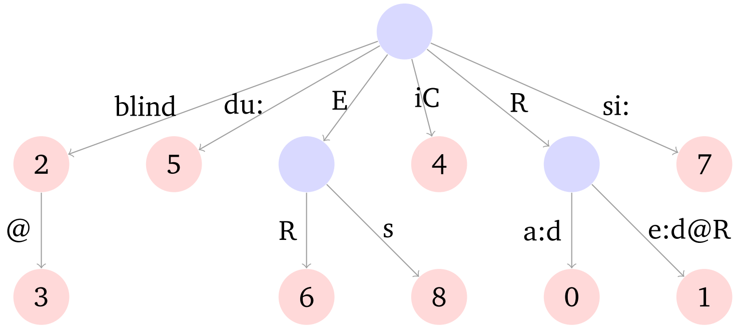
Considering that our phonology list consists of fully inflected output forms and that inflectional morphology in many languages involves a fair share of suffixation, it is very likely that the keys in the phonology list of a given language show a great amount of overlap. This allows us to represent them in a more memory efficient way via a *prefix tree* (also called *trie*) like the one below.



Nodes in red represent specific keys. For instance, the red node labeled 2 corresponds to the key *blind*, for if we follow the path from the root to this node, the branches spell out b-l-i-n-d. The node is labeled 2 because that is the index of the entry with key *blind*. If we move one branch further down we get the key *blind@*, which references the item with index 3. Notice that since the tree already encodes the key *blind@*, we already have all the nodes and branches that are required to encode *blind*. Thus the addition of this key only requires 2 more bits, which are used to link the corresponding node to the index 2 (remember, 2 is 10 in binary). If the keys were stored independently of each other, then *blind* would be one among $n$ elements, where $n$ is the number of items in the phonology list. Our toy example has 9 different elements, so we would need 4 bits to uniquely identify *blind* (9 is 1000 in binary). If there's 20,000 different keys, we would need 15 bits. This shows that representing one key as a prefix of another one can save us a lot of memory via structure sharing.

The prefix tree also renders the hash function obsolete as it already encodes the mapping from keys to indices. A prefix tree thus is a viable alternative to a hash table that saves quite a bit of memory as long as the majority of keys have common prefixes. Actually this is not the case in our toy example, where we find many unary branching nodes that do not correspond to list indices, and this has some unfortunate consequences for memory usage. For a single key like *du*, which shares no prefixes with any other keys, the prefix trees has to reserve memory for the characters *d*, *u*, and *:* which are found along the path to the corresponding index. In the standard ASCII encoding, that's 7 bits per character, and in UTF-16 it would be 16 bits per character. Remember that the hash function needs at most 4 bits for *du:* because it treats the key as one among 9 atomic symbols. The prefix tree decomposes the keys into sequences of characters, so now we have to reserve $7 \times 3 = 21$ or possibly even $16 \times 3 = 48$ bits for the same key. This is a general problem with prefix trees: they are only memory efficient if many keys share common suffixes, or equivalently, if most nodes in the tree are at least binary branching.

We can somewhat mitigate the problem by truncating sequences of unary branches into a single branch, yielding a *compacted prefix tree* (also known as *radix trie*).



Overall, it seems than an algorithmically sophisticated implementation of the phonology list model could attain some degree of psycholinguistic plausibility with respect to memory usage and retrieval speed. As we will see next, though, the phonology list model fails miserably at the computational level.

## 3 Computational Properties Missed by the List Model

If phonology is just a list of output forms, we predict several properties to hold of natural languages:

- **Free Typology**
  Since there are no restrictions on what items may occur in a list, any given list of output forms is a possible phonological system.

- **Defeat by the unknown**
  When presented with a new word, speakers cannot find this word in their phonology list and thus do not know how to pronounce it.

- **Isolationism**
  The pronunciation of a word is always the same and does not alter depending on the preceding and following words in a sentence.

- **Finiteness**
  If phonology is simply a list that keeps track of the pronunciation of words a speaker has encountered so far, then this list must be finite — nobody has heard an infinite number of words at any given point in their life. Consequently, a speaker's phonological system contains only a finite number of words.
- **Egalitarianism**
  All phonological forms are equally easy to learn because they simply involve memorization. If there are differences at all, they should be between short words and long words.

None of these properties hold of natural language phonology.

Contrary to Free Typology, there are obvious typological regularities that hold across languages. For instance, languages obey a principle of *vowel dispersion*: the size of the vowel inventory is a rough predictor of which vowels will not occur in this language. So if a language has only three vowels, odds are high that they will be close to the three cardinal vowels a, i, and u, and it is absolutely certain that they will not be i, y, and ʏ. There's also an infinite number of easily formulated principles that are not obeyed by a single language, such as "if the number of vowels in the word is a multiple of 3, they must all be a", or "a word contains more than four phones only if it is a palindrome", or "the number of bits used for the ASCII encoding of the word is greater than 17". Yet we can easily write lists that satisfy these conditions.

It is also obvious that native speakers do have intuitions about the pronunciation of nonce words, and this has also been verified in a myriad of experiments using *wug* tests. In a *wug* test, the test subject is presented with a made-up word and asked to determine its pronunciation. The experimenter may show the subject a drawing of a bird and tell them that this bird is called a [wʌg]. The experimenter then reveals a picture with two wugs and asks the test subject to complete the sentence "On this picture, there are two…". A native speaker of English will realize that the plural *wugs* is pronounced [wʌgz] rather than [wʌgs] because the voicing of the plural morpheme is dependent on the preceding sound. A similar experiment could be performed to test German speaker's knowledge of final devoicing: the subject is first told the plural [vagə], from which they should correctly infer that the singular is not [vag] but [vak].

Isolationism does not hold, either. For example, *phone bill* is often pronounced like *foam bill* in rapid speech, and *white board* may actually sound like *wipe board*. Here the pronunciation of the last sound of the first word is partially assimilated to the first sound of the following word: the alveolar nasal [n] is changed to the bilabial nasal [m] because [b] is bilabial, while the alveolar plosive [t] becomes the bilabial plosive [p] in this context. The only way this could be handled in the list model is by treating these words as compounds with their own entries. But *white board* is not necessarily a compound (we may just be talking about some kind of board that happens to be white), so that is hardly satisfying. Alternatively, we could add context information to every pronunciation, but that would increase the memory demands of the model quite a bit, and it also constitutes a step towards a more complicated model, so we will hold off on that for now.

The suggested fix of treating certain word combinations as compounds with their own entry is actually instructive in that it shows why natural languages do not have a finite vocabulary. Take a simple expression like *great grandfather*. Using various linguistic tests one can show that it is not a phrase, and since I) it is usually assumed that if a multi-word expression isn't a phrase, it must be a compound, and II) compounds are still words, the phonology list should include an entry for *great grandfather*. But then it must also contain an entry *great great grandfather*, *great great great grandfather*, and so on. In general, native speakers know exactly how to pronounce *great*$_n$ *grandfather* for any arbitrary choice of $n$, so if phonology is only a list of pronunciations, it must be an extremely large list that contains pronunciations for such high values of $n$ that nobody could ever reach the limit within their life span. But a list that is this large may just as well be considered to be infinite — it certainly isn't feasible to write such a list by hand, and it is unclear how a native speaker could have acquired such a list if phonology is just memorization of output forms.

Speaking of acquisition, it is also a fact that speakers have a harder time with certain pronunciations than with others. The simplest example are tongue twisters of course, although it is unclear whether the challenge stems from phonological idiosyncrasies or is simply articulatory in nature. A different example involves pronunciation shifts. For example, *biopic* (= biographic picture) used to be pronounced with primary stress on the *i*, similar to *bioinformatics* or *bio-degradable*. In recent years, however, more and more speakers have switched to a pronunciation with primary stress on the *o*, apparently in an analogy to *myopic*. If phonology is just a memorized list, it is unclear what the motivation for this change could be, any form is as good as the next.

Playing devil's advocate, one may propose that all these properties of natural language need not be explained by the list model of phonology since they could be due to the learning algorithm. We do not see free typological variation because the learning algorithm will only entertain specific ways of inferring a phonology list from the input data. The phonology list need not be able to determine new pronunciations because the learning algorithm can take care of that based on some probabilistic metrics. Finiteness is not an issue because the learning algorithm can dynamically add new words to the list whenever they are needed. And egalitarianism does not hold because the learning algorithm can be structured in a way such that certain pronunciations are more preferable to others, e.g. by reducing the entropy of the list ($\approx$ the amount of idiosyncratic information that cannot be derived from more basic principles).

But adopting this stance is no different from admitting defeat. Keep in mind that a learner that accounts for all these properties is much more complex than the simplest learner for the list phonology model, namely the one that simply adds new pronunciations to a list. The move towards a more complex learner is necessary because this maximally simple model fails miserably. We need a more complicated model, and for this purpose it does not matter whether we put the complexity in the grammar formalism or the learner, without a good understanding of whether one of the two is better suited for certain purposes it is irrelevant which component takes care of which job. For now, we want to know how these additional properties of natural language phonology can be characterized on the computational level, not how they are distributed over grammar and learning algorithm. The easiest way of doing this is to once again start out with the simplest model where all the work is done by one component, and since we already have a grammar component in place, we should first see if a more elaborate grammar model can solve these issues.

### Relevant literature for Unit 2

add more info