Computational Linguistics as Language Science

Spring 2019

Thomas Graf

About this book

What?

This textbook is a graduate-level introduction to proof-based (rather than modeling-based) computational linguistics for linguists. The focus is on how a computationally informed perspective can illuminate aspects of language that theoretical linguists and cognitive scientists care about. At concerted effort has been made to keep the material as approachable as possible without sacrificing exactness. To get the most out of the book, readers have to be willing to engage with mathematical notation, but each unit is designed in a modular fashion so that the less mathematically inclined can skip the parts they find too tedious.

Why?

Like most textbooks, this one was written because of a picky instructor who wasn't quite happy with the existing options. It grew out of my lecture notes for *Computational Linguistics 2* at Stony Brook University's Department of Linguistics. Within linguistics, computational linguistics tends to be geared towards model-building and simulations: MaxEnt learners, corpus-based techniques, and so on. These topics are covered in a different course at Stony Brook, though, with Computational Linguistics 2 exploring the proof-based side of the field: formal language theory, subregular complexity, logic, string and tree automata/transducers, learnability, parsing theory, algebra, plus some algorithms and data structures. There is a number of excellent textbooks that cover a subset of these topics, but they all fell short in some respect:

- Marcus Kracht's *The Mathematics of Language* is a treasure trove and still one of the most rewarding textbooks ever written, but it is too hard and theoretical for the average linguistics student.

- *Speech and Language Processing* by Jurafsky and Martin is an excellent reference book, but it is clearly aimed at computer science students. The focus is on engineering techniques rather than investigating linguistic questions from a computational perspective.

- András Kornai's *Mathematical Linguistics* presents a more appropriate compromise between linguistic inquiry and applications, but is still too far removed from linguistics and cognition for my taste.

- *Mathematical Methods in Linguistics* by Partee, ter Meulen & Wall is a classic, but as the title implies it mostly focuses on mathematics, in particular those areas that are needed for semantics (set theory, algebra, and predicate logic). Its final chapter on formal language theory is a commendable inclusion, but does not make a strong case for why linguists should care about this perspective.

- Ed Keenan and Larry Moss have produced an impressive and very comprehensive introduction to mathematical linguistics with *Mathematical Structures in Language* (and I'm not just saying that because of the many fond memories I have of TAing for the course that their book sprang from). But overall it is a mathematical methods book, not an introduction to proof-based computational linguistics as an means to study language.

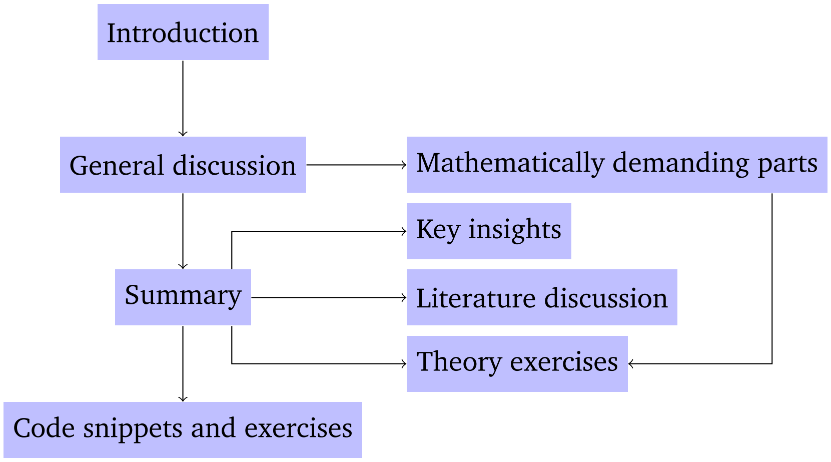
- Some subtopics have dedicated textbooks, e.g. Laura Kallmeyer's *Parsing Beyond Context-Free Grammars*, *Categorial Grammar* by Glyn Morrill, *The Logic of Categorial Grammars* by Richard Moot and Christian Retoré, or my colleague Jeff Heinz's book *Grammatical Inference for Computational Linguistics*, co-authored with Colin de La Higuera and Menno van Zaanen. I have happily used them in special topics courses, but they proved difficult to sample from for a broad introduction course like Computational Linguistics 2.

I am sure I have forgotten a few others, all of them deserving of a shout-out here. But the bottom-line is that none fit my vision of proof-based computational linguistics as a way of studying language. So here we are.

How?

Computational linguistics is a very interdisciplinary field, and this is also reflected in the students that have taken the course over the years: from advanced Ph.D. students in linguistics to philosophy and psychology grad students as well as computer scientists with no background in linguistics at all. It is impossible to serve all of them equally well, so I decided to keep the focus on what would benefit and interest linguists the most. But I did make minor additions to include other groups — that is why the linguistically trained reader will sometimes encounter brief sections on very basic material they obviously know about already. I hope they won't feel patronized.

Even the group of linguistics students is far from homogeneous, though, in particular at Stony Brook. Depending on which classes they have taken before, their mathematics and programming background may range from non-existant to far above average. For this reason the textbook has a partly non-linear design:



Every reader should be able to complete the main path from the introduction to the summary. While some math is unavoidable even in those general sections, the more demanding parts are put in a separate section. All the math that is required to follow the main discussion is explained in separate background boxes:

Background

These boxes occur throughout the text and introduce basic concepts of mathematics and computer science.

Exercises are included without solution so that they can be reused by instructors in their course. Each exercise is assigned one of four difficulty levels, indicated by asterisks:

[Exercise 0.1 The lowest difficulty. Every student should be able to solve this by simply applying a key technique of the unit in a mechanical fashion.]

[Exercise 0.2* Requires some effort and creativity. The student cannot simply follow a recipe from the unit but has to build on their understanding of the material to come up with a solution of their own.]

[Exercise 0.3** Very difficult for students, and instructors might have to sit down for a minute or two to figure out the general strategy. Writing up the full solution might take quite a while, and/or the solution involves a lot of lateral, out-of-the-box thinking.]

[Exercise 0.4*** Open research problem. If somebody figures out the answer, they should write a paper about it.]

The book also includes optional discussions of programming issues. Make not mistake, though, this is not a programming textbook. Nonetheless it is often instructive to analyze how exactly formal ideas can be translated into concrete code. I have opted not to put these code snippets directly in the text as they may distract those readers who want to focus exclusively on the interplay of linguistics and abstract computation. Instead, the relevant code snippets are discussed at the very end of the respective unit, after the summary.

A few more remarks on the coding sections are in order. I decided to go with Python instead of pseudo code. Pseudo code is difficult to follow for the uninitiated. Python, on the other hand, enjoys rapidly growing popularity in linguistics and the sciences. Even a reader with a general programming background but no prior experience with Python in particular should be able to understand the code snippets.

To further increase the approachability of the code sections, I have adopted a beginner-friendly coding style. I err very much on the verbose side with line comments and docstrings (which document the purpose and overall design of each function). I also make a deliberate effort to avoid advanced techniques such as decorators and generators. Error handling, which is very important for production-level software, is completely omitted. Similarly, efficiency is considered only to the extent that it illustrates a key pedagogical point. Quite simply, the code sections are of little interest on their own and are designed only to support the concepts covered in each unit.

Depending on which parts of a unit one skips, the book can range from a lightweight introduction that conveys the key intuitions to a very formal *tour de force*.

Contents

Contents (detailed)

Unit 1

Computation(al) linguistics

Perhaps the most unfortunate fact about the field of computational linguistics is that its name is *computational linguistics*. The term inherits a dichotomy that is hard to tease apart for the uninitiated: the distinction between computers and computing.

When the layperson hears the term *computational*, they immediately think of doing things with computers. The actual hardware might be a laptop, a phone, or the NSA's giant server farm in Utah, but in the end it always boils down to some kind of electronic device that was deliberately designed and engineered by humans. In the case of language, there certainly is no shortage of tasks we want these devices to handle: word completion for text messages, speech recognition and speech synthesis for our GPS, detecting spam mails, translating websites on the fly, and much more. Thanks to decades of research in computational linguistics, computers now do surprisingly well at these tasks. Sure, the existing solutions are far from perfect and occasionally make bewildering mistakes unlike anything even a highly inebriated human would produce. The more linguistically complex the task, the worse computers tend to fare. But the technology has proven good enough to become an essential part of our daily lives, and it keeps improving at a rapid rate. Computational linguistics as the study and design of language technology has been a resounding success.

But language technology isn't all there is to computational linguistics. Limiting the notion of computation to "doing things with computers" is doing it a great disservice. Similarly, the scope of computational linguistics goes far beyond computers. It encompasses any object that is capable of computing, be it computers, the human brain, or abstract computing devices like the Turing machine, which isn't tied to a specific physical medium. With this general notion of computing, the goal is no longer to solve language-related tasks. No, language **is** the task: what does language look like from a computational perspective?

The book you are holding in your hands (and, presumably, reading) is all about this broader notion of computational linguistics. For lack of a better term, I call this *computation linguistics*. Its focus on understanding language makes computation linguistics a subfield of linguistics, even if its methods borrow heavily from theoretical computer science and mathematics (formal language theory, learnability, algebra, lattices, parsing theory, and so on). The remainder of this chapter tries to sharpen the profile of computation linguistics and how it differs from other varieties of computational linguistics. I will also discuss why this approach is worth pursuing. The specific merits of computation linguistics depends a lot on whether you are a natural language engineer, a theoretical linguist, or a cognitive scientist. But rest assured, each group stands to gain something.

One more remark: given its subject matter, this chapter is necessarily very meta-theoretic. If you would rather get right into the thicket of things, feel free to jump ahead to the next chapter. After all, the pudding is in the eating.

1 Computers vs computation

While computers are the most common tool for carrying out computations nowadays, they are not what computation is about. Computation, in its barest form, is the principled manipulation of information, of transforming some input into some output in a precise, step-wise fashion. When a computer verifies $1 + 1 = 2$, this act of computation is instantiated via a series of electrical impulses that affect some of the millions of transistors that make up its hardware. But the computation is not tied to that specific electrical process, it can take many physical instantiations in this world.

The movie buffs among you will remember the 1990s masterpiece *MacGyver: Lost Treasure of Atlantis*. In this spiritual successor to the *Indiana Jones* movies, MacGyver discovers the secret of Atlantis: a giant, steam-powered computer that operates without electricity. MacGyver is incredulous — how could they have a computer without electricity? But the idea of a steam-powered computer is actually far from outlandish. Any device that can assume multiple different states and switch between those states in a controlled fashion is capable of computation.

The idea that the act of computation is about transitioning from one internal configuration to another in a principled fashion is the central insight behind the *Turing machine*. It was first defined by Alan Turing in 1936 in his seminal paper *On Computable Numbers, with an Application to the Entscheidungsproblem*.

Alan Turing (1912–1954)

Alan Turing was an English mathematician and polymath. His gamut of accomplishments have made him one of the most influential researchers of the 20th century. He laid large parts of the foundation of modern computer science, in particular artificial intelligence and the theory of computation. During World War 2, he took a leading role in cracking the Enigma, an encryption device used by the Nazis. It has been estimated that this breakthrough saved millions of lives. Turing was also an excellent long-distance runner, and was even a contender for a spot on the British Olympic team in 1948.

Turing's numerous accomplishments were not fully acknowledged during his lifetime, and as a homosexual he even faced criminal prosecution. In 1952, he was sentenced to undergo chemical castration treatment due to acts of "gross indecency" (i.e. homosexual relations). Two years later he died of cyanide poisoning, presumably a suicide.

It is astounding how much a Turing machine can accomplish even though it consists of only three components:

1. a rewritable and extendable tape that acts as a general data storage, and

2. a read/write head that can modify the tape, and

3. a state register that controls the behavior of the machine.

The read/write head can move to any position on the tape, read the symbol that is there, and possibly overwrite it with a new symbol. The state register is like a knob that can be in one of finitely many positions, e.g. 7 out of 10 on a volume dial. The purpose of the state register is to control the behavior of the Turing machine. Based on the symbol that is currently under the read/write head and the state of the register, the Turing machine consecutively executes three specific operations:

1. a *write action* (overwrite or do nothing), and
2. a *move action* (move left, move right, stay in place), and
3. a *state register change* (keep state, switch to different state).

A simple instruction of a Turing machine may read "if the symbol under the head is 1 and the state is A, overwrite 1 with 0, move left, and switch to state B". A finite collection of such instructions is a program that can be run on a Turing machine to carry out specific computations.

Example 1.1 Copying with a Turing machine

The table below describes a small program for a Turing machine.

current state	*tape symbol*	*write action*	*move action*	*new state*
A	0	none	none	F
A	1	write(0)	⇐	B
B	0	none	⇐	C
B	1	none	⇐	B
C	0	write(1)	⇒	D
C	1	none	⇐	C
D	0	none	⇒	E
D	1	none	⇒	D
E	0	write(1)	⇐	A
E	1	none	⇒	E

This looks fairly cryptic, so let us tease apart what is going on here.

Each individual instruction is easy enough to decypher. The machine has 6 different states: A, B, C, D, E, and F. Only two kinds of symbols are used on the tape, 0 and 1. A command like write(1) means that the machine writes a 1 onto the tape, whereas the arrows ⇐ and ⇒ specify that the machine moves one symbol to the left or one symbol to the right after the write action is finished. So line 5, for example, tells us that if the machine is in state C and has a 0 under its read/write head, it writes a 1, moves to the next symbol to the right, and switches into state D.

That is all nice and dandy, but what is it the program does? So far this feels like a badly written instruction manual where each individual sentence makes sense but you can't figure out how they fit together (very much **un**like this book, I hope). Let us boost our understanding of the program by working through a concrete example.

Suppose that the machine starts in the following configuration: The tape consists mostly of 0s, except for two adjacent 1s, and the read-write head is positioned on the right 1, with the state register in state A. This is visualized below.

$$0\quad 0\quad 0\quad 0\quad 1\ \boxed{\begin{array}{c}1\\ \mathrm{A}\end{array}}\ 0\quad 0$$

This configuration is matched by the second line of the instruction table. Hence the machine overwrites the current symbol with a 0, moves to the left and switches into state B. Here is the resulting configuration, with the changed symbol highlighted in **boldface**.

$$0\quad 0\quad 0\quad 0\ \boxed{\begin{array}{c}1\\ \mathrm{B}\end{array}}\ \mathbf{0}\quad 0\quad 0$$

Since the read/write head is now over a 1 while the machine is in state B, the instruction on line 4 is triggered: the machine keeps the current symbol as is, moves to the left, and keeps the register in state B.

$$0\quad 0\quad 0\ \boxed{\begin{array}{c}0\\ \mathrm{B}\end{array}}\ 1\quad 0\quad 0\quad 0$$

This in turn triggers the third instruction, which tells the machine not to perform any write action, move one symbol to the left, and switch the register to state C.

$$0\quad 0\ \boxed{\begin{array}{c}0\\ \mathrm{C}\end{array}}\ 0\quad 1\quad 0\quad 0\quad 0$$

The rest of the computation then proceeds as depicted below:

$$0\quad 0\quad \mathbf{1}\ \boxed{\begin{array}{c}0\\ \mathrm{D}\end{array}}\ 1\quad 0\quad 0\quad 0$$

$$0\quad 0\quad 1\quad 0\ \boxed{\begin{array}{c}1\\ \mathrm{E}\end{array}}\ 0\quad 0\quad 0$$

$$0\quad 0\quad 1\quad 0\quad 1\ \boxed{\begin{array}{c}0\\ \mathrm{E}\end{array}}\ 0\quad 0$$

$$0\quad 0\quad 1\quad 0\ \boxed{\begin{array}{c}1\\ \mathrm{A}\end{array}}\ \mathbf{1}\quad 0\quad 0$$

$$0\quad 0\quad 1\ \boxed{\begin{array}{c}0\\ \mathrm{B}\end{array}}\ \mathbf{0}\quad 1\quad 0\quad 0$$

$$0\quad 0\ \boxed{\begin{array}{c}1\\ \mathrm{C}\end{array}}\ 0\quad 0\quad 1\quad 0\quad 0$$

$$0\ \boxed{\begin{array}{c}0\\ \mathrm{C}\end{array}}\ 1\quad 0\quad 0\quad 1\quad 0\quad 0$$

0 **1** $\underset{\text{D}}{\boxed{1}}$ 0 0 1 0 0

0 1 1 $\underset{\text{D}}{\boxed{0}}$ 0 1 0 0

0 1 1 0 $\underset{\text{E}}{\boxed{0}}$ 1 0 0

0 1 1 $\underset{\text{A}}{\boxed{0}}$ **1** 1 0 0

0 1 1 $\underset{\text{F}}{\boxed{0}}$ 1 1 0 0

The F state is special because it does not trigger any new instructions, so the machine halts once it reaches this state. Looking at the result of the individual steps, we can now make sense of the instructions at the beginning of this example. Put together, they program the Turing machine so that it copies sequences of 1s. If the tape had contained 11111 instead of 11, the final output would have contained two instances of 11111. That's longer than the tape in our example, but remember that the tape of a Turing machine can always be extended as necessary.

The example above shows that a Turing machine can create copies of an input. As we will see in later chapters, copying is actually a very complex task that plays an important role in natural languages. Despite the complexity of copying, it can be understood in the very general terms of a Turing machine as simply a sequence of configuration changes: what does the tape look like, where are we on the tape, and what state is the machine in?

The generality of the Turing machine is what enables a broader understanding of computation that does not care about the actual hardware. A Turing machine is not a concrete object, it's not a tiny box with some tape and a state dial that you can order from Amazon. Instead, it is an abstract model of what it means to carry out a computation, and there are many different ways a Turing machine can be instantiated in the real world. This is particularly noteworthy because Turing machines act as a kind of standard model for computation. If there is no limit on how much tape is available, any problem that can be solved computationally can be solved by a Turing machine. So all kinds of computation can be regarded as a specific program that runs on a Turing machine. But this also means that any machine, system, or construct that provides the equivalent of a tape and a controllable read-write head is a computing device.

For example, a Turing machine can even take the form of a very selfish drinking game: Gather 4 friends of yours and 6 shot glasses — I am boldly assuming that you have enough of both. Put the shot glasses in a line and fill the rightmost two with a beverage of your choice. Then give each one of your friends a card with instructions they have to follow. For the sake of exposition, let's assume that your friends are called Bill, Cathy, Damian, and Edith. Their respective cards read as follows:

- **Bill**
  If the shotglass in front of you is empty, get out of line and put Cathy in front of the glass to the left. Otherwise, leave the glass alone (sorry!), and move to the glass to the left.

- **Cathy**
  If the shotglass in front of you is empty, fill it up, get out of line, and put Damian in front of the glass to the right. Otherwise, leave the glass alone (sorry!), and move to the glass to the left.

- **Damian**
  If the shotglass in front of you is empty, get out of line and put Edith in front of the glass to the right. Otherwise, move to the glass to the right.

- **Edith**
  If the shotglass in front of you is empty, fill it up, get out of line, and put me in front of the glass to the left. Otherwise, move to the glass to the right.

The instructions for yourself are slightly more fun. If the glass in front of you is full, drink it all, get out of line, and put Bill in front of the glass to the left. If the glass is empty, the game is over.

I suppose you can already tell what is going on here. When you play this game, it will proceed exactly like the Turing machine from our copying example (it is a selfish drinking game because you are the only one who gets to drink, i.e. rewrite a 1 as a 0). Even though you and your friends are separate individuals, the combination of you, your friends, and the shotglasses constitutes a Turing machine. The instructions you give each person are parts of the program that runs on this Turing machine.

We can make all kinds of changes to this setup without losing the connection to Turing machines. For example, we may use bowls instead of glasses, and fill them with M&Ms instead of some beverage. And maybe we do not actually put the bowls in a line but instead assume that they all differ in size, like a set of baking bowls that is randomly distributed around your ktichen. We then reinterpret “left” so that it means “the largest bowl that is smaller than the current bowl”. And “right” now means “the smallest bowl that is bigger than the current bowl”. Even though we no longer have the bowls in a line, we can still move “left” and “right” based on the relative size of the bowls. Perhaps we could even replace your friends with a very well-trained dog. Clearly a dog and a human are two very different things, but it changes nothing about the computation that is being carried out. No matter how we set things up, the same input will always be transformed into the same output. Shot glasses, bowls, M&Ms, humans, dogs, it does not matter, we always end up copying the input.

Silly as these examples may be, the central point stands: changes to the physical make-up of the device that carries out the computation does not entail changes to the computation itself. The notion of computation operates at a higher level of abstraction, and that is what gives it such a unifying power. We can take computational concepts and apply them to systems that do not at all look like the computers we are familiar with: the human brain, the biological mechanisms of gene expression, even the universe itself. Despite the differences in physical substrates, structural changes, and sheer computing power, they are all equally valid examples of computing devices and we can discover interesting things about them by adopting this perspective.

Hopefully you can now appreciate why it is unfortunate that the term *computational linguistics* does not clearly disambiguate between computation and computers. While the latter emphasizes engineering concerns, the former strives for a more abstract perspective that applies to computers as well as humans. Remember, humans are the only known computing device with perfect command of natural language. In the spirit of learning from nature, we would do well to study language at a level that is compatible with these devices and learn from them as much as possible.

To clearly differentiate the two notions of computational linguistics, I will use *natural language processing* (NLP) in this book to refer to those aspects of computational linguistics that are solely concerned with computers. NLP is about solving language-related tasks with computers, e.g. speech synthesis, machine translation, or even the basic search function in your text editor. Computation linguistics is about studying language as an instance of computation. Both NLP and computation linguistics belong to the larger field of computational linguistics, but they differ largely in their goals and also somewhat in their methods.

While the terminology might be less vague now, the underlying concepts are still very abstract and intangible. To some extent things will only get clearer once we move on from the high-level perspective in this chapter and start looking at concrete issues. Still, if the road ahead is shrouded in mystery, it would at least be nice to why it is a road worth travelling. So let us next consider why one would want to study language from a computational perspective.

2 Why computation linguistics?

The focus on computation linguistics might seem peculiar to you. NLP has an easy sales pitch: “Make the world a better place while earning a six digit salary.” Computation linguistics, on the other hand, has less tangible goals. Even if you are a heavily theory-minded researcher, it might not be immediately clear what computation linguistics has to offer. But there are good arguments for computation linguistics, and they range from real-world applications to scientific insights.

2.1 Practical arguments

The standard argument, and its standard counterargument

Let us first look at an argument that seems plausible, but ends up running into several issues. The argument starts with the plausible assumption that a world in which computers can successfully handle all kinds of language-related tasks is preferable to one where they cannot. This would create a second industrial revolution that boldly pushes automation into language-heavy domains: customer service and speech-driven user interfaces, language and writing instruction, knowledge aggregation, and much more. Admittedly there is also the risk of mass surveillance, mass unemployment, and the social upheavals that tend to follow both, but there is reason to believe that those would just be short-term growing pains on the way to a more prosperous future. If this is correct, then it is imperative that we do whatever we can to get computers to this level of aptitude. And just like some understanding of physics had to be in place before engineers could bless mankind with the radio or the combustion engine, we cannot have successful NLP applications without a minimum understanding of language and the computational challenges it poses. Computation linguistics thus is a prerequisite for NLP.

This argument is intuitively pleasing and, in my humble estimate, ultimately correct. In the form presented above, though, it is too simplistic and easy prey to somebody playing devil's advocate. Let's take a careful look at the counterargument such a person might put forward:

One of the most shocking aspects of the applied sciences and engineering is how little genuine understanding one needs to construct a useful tool. To give but a few examples: relativity theory is not an integral part of calculating artillery ballistics, the smallpox vaccine did not need a theory of germs, and you don't need to understand convection to build a good chimney. In many areas of life, the permitted margin of error is large enough that shortcuts, hacks, and brute force methods will get the job done just fine. For practical purposes it is also perfectly fine to make stipulations that fly in the face of scientific consensus but improve the final results. In the words of Noam Chomsky, the founding father of modern linguistics (Chomsky 1990: 147):

> Throughout history, those who built bridges or designed airplanes often had to make explicit assumptions that went beyond the understanding of the basic sciences.

Similar things can be observed in NLP. Many of its tools and techniques ignore linguistic ideas for the sake of simplicity and efficiency. These tools do surprisingly well and often outperform competing models that draw from what linguists have learned about language. The state of affairs is summarized very succinctly by a hyperbolic quote that is commonly attributed to the computational linguist Frederick Jelinek:

> Every time I fire a linguist, the performance of the speech recognizer goes up.

Frederick Jelinek (1932–2010)

Frederick Jelinek played a key role in bringing information theory and probabilistic methods to computational linguistics, or rather, bringing them back from the dead.

Following the success of Chomsky's *Transformational grammar* in the 50s and 60s, computational linguists put their hope in rule-based approaches and largely stayed away from statistics and probabilities. Jelinek bucked this trend. After he joined IBM in the 70s, he worked tirelessly on designing speech recognition systems that were sufficiently robust for real-world application. The more theoretically minded, rule-based approaches had nothing comparable to offer. By the 1990s, the probabilistic models Jelinek pioneered had become a corner stone of NLP, and they remain important to this day. But perhaps the pendulum has swung a bit too far — the rule-based side may still come back with a revolution of its own.

There are numerous examples of simple (or downright simplistic) models matching or even outperforming more sophisticated models that are deeply steeped in theory. Consider the case of a 2013 study that evaluated various text-based models for predicting the success of novels. Naively, one would expect that a novel's success is largely dependent on its writing style, narrative structure, and subject matter. Yet one of the best models completely ignored those aspects and focused exclusively on word frequencies. As it turns out, successful novels have an unusually high frequency of thought-oriented verbs like *recognize* and *remember*. A model that pays attention to such word frequency effects can even perform better than one that also analyzes the minute structural intricacies that linguists care about.

It seems, then, that there is a shockingly large gap between "language as a computational problem" and "computers solving natural language tasks". The latter is doing just fine without the former, and in some cases theory may even make things worse for practical applications.

With the rapid rise of machine learning methods in recent years, one might even expect the gap to widen over the next few decades. General purpose machine learning strategies have made major inroads in recent years, in particular in the form of neural networks. The approaches deliberately forego all linguistic knowledge and instead opt for treating language as an arbitrary data-crunching problem like any other. And once again they turned out to be surprisingly performant. If this trend continues to its logical extreme, NLP may be completely absorbed into the field of machine learning in the near future. In this case, any language-specific insights would be even harder to incorporate given the domain-agnostic nature of the machine learning techniques. If "computers solving natural language tasks" simply turns into "computers solving tasks", then "language as a computational problem" might be too specific an enterprise to make many worthwhile contributions. In sum, then, computation linguistics might have few actual contributions to make to NLP.

And thus we have arrived at the conclusion of the devil's advocate: while in principle computation linguists might be able to help with practical applications, recent history tells us otherwise. Other fields like physics and chemistry might show a trickle-down effect from theory to applications, but a look at the field right now reveals no comparable pipeline from computation linguistics to NLP.

A counterargument to the counterargument

There is more than a grain of truth to the devil's advocate's reasoning, but at the same it is too narrow and overly reductionist. Things simply aren't that black-and-white, and a more nuanced analysis reveals interesting shades of gray.

Admittedly some areas of application do not stand to profit much from linguistic insight. For example, a government may want to improve the mental well-being of its citizens by screening tweets for signs of mental illness and offering preventive care to users that seem to be at risk. The nature of this task is largely independent of the complexities of natural language and its computational intricacies. For one thing, tweets are often short enough that their intended meaning can be inferred just from a few keywords and hashtags. Second, the correlation between mental health and language use is fairly loose and not particularly well-understood. In the absence of a robust theory, the best approach is trial-and-error: identify a few reasonable indicators of mental illness, build several probabilistic models that pay attention to these indicators, and go with the model that performs best. This route is guaranteed to produce some kind of model within a short amount of time, and the model is very likely to perform better than any model that builds on the most recent scientific research, simply because the problem is too limited for the theoretical underpinning to pay off in a significant manner. The more specific and restricted the problem, the less need there is for deep understanding.

But not all problems are simple or very restricted, and this leaves room for computation linguistics to make worthwhile contributions. Language technology is still very much about going for the low-hanging fruit. Many interesting problems remain untackled. No computer currently understands instructions like the following: “Go to my pictures folder and rename the files. I want each filename to start with the date and then list all the names of the people in the picture.” A simple request, it seems, but it is actually bursting with linguistic complexity:

1. “my pictures folder”: Which folder is that? Who is speaking here? What if there’s multiple folders with pictures? Can we make a reasonable guess about which folder the user has in mind?

2. “rename the files”: What files? Presumably the user only wants to rename the files in the pictures folder, not all the files on the hard drive. But this piece of information needs to be inferred, it is not given explicitly.

3. “I want”: So the computer should make sure that the files end up looking like what the user has in mind, even though the user isn’t explicitly issuing a command.

4. “each filename”: Actually the user wants only image files like JPGs or PNGs to be renamed, whereas the doc-file that they randomly dropped in there should be left alone.

5. “start with the date”: Dates can be formatted in various ways. Can we make a reasonable guess about what format the user wants? If not, how can we ask for clarification?

6. “then list”: Is this still part of the description of the filename or a new command to the computer? That is to say, should the filename include the names of people, or should the computer list their names for the user after each successful file renaming?

7. “list all the names of the people in the picture”: What is the intended interpretation here? People that appear in the picture should have their name listed in the filename? Each person should have their name listed in the picture? List every name that belongs to a group of people and appears in the picture?

8. “the picture”: Which picture? Probably the one that is being renamed, but again this restriction is left implicit.

In our amazement that computers seem to accomplish super-human feats such as translating a website in a split-second, we tend to forget that most of the language-related tasks that even 5-year olds solve with ease still pose insurmountable challenges for computers. Once one broadens the horizon from what NLP is currently working on to what NLP could be working on, the importance of computation linguistics becomes much more apparent.

In a certain sense this holds even if one considers only those areas where NLP has been successful. Machine translation, for example, has made tremendous leaps in the last 10 years and can produce remarkable results now — if one picks the right source and target language. Translating a text from English to Spanish will work much better than translating the very same text from Swahili to Inuktitut. This is because modern NLP techniques are tremendously data-hungry. In order for a probabilistic model to perform well, it has to be fed millions of data points. Only a few languages have enough digitized text to meet the data hunger of these models, all other languages are considered *resource poor*. Such resource poor languages are second-class citizens in the realm of NLP. Note that even languages like Swedish and Afrikaans, which are spoken in very affluent countries, are resource-poor for NLP purposes. And the problem isn't limited to languages, every dialect that deviates to a significant extent from the standard is resource poor. So Scottish English, Bavarian German, and African-American Vernacular English are all ill-served by current NLP solutions compared to their standard counterparts. As language technology becomes increasingly important, there is a real danger that current NLP techniques end up indirectly discriminating against (linguistic) minorities.

That this does not need to be the case is proved thousands of times each day all over the world. Each day, children in linguistic communities of all shapes and sizes effortlessly acquire their native language from very limited input. In fact, children are amazing language learners and frequently generalize in ways that go far beyond what the data provides. This is what linguists call the *poverty of stimulus*: even though the linguistic input children get is very limited, they infer a lot more from it than would be possible just via raw statistics. How exactly children manage to do this is still very much an open issue, in particular on the computational level. But the more our knowledge advances on this front, the less data future models will need, thereby shrinking the gap between resource-poor and resource-rich languages.

It is also worth noting that the performance of NLP models should not be taken at face value because it is based on a particular notion of performance. The NLP community has devised numerous quantitative metrics for benchmarking its models. But these benchmarks have several blind spots in their design. For one thing, they only measure how many errors a model makes, rather than how grave the errors are. Suppose you have to evaluate how well two children — let's call them Shorty and Cat — have mastered basic addition. The table below shows your questions and their respective answers.

Question	**Shorty**	**Cat**
What is 3+4?	6	7
What is 6+3?	9	9
What is 7+3?	10	10
What is 3+4+3?	9	343

Looking purely at the number of correct answers, Luna does better than Shorty. But every teacher will tell you that Shorty has a better grasp of addition than Luna. Yes, Shorty is short by 1 when adding up 3 and 4. But once you take that mistake into account, all the other answers make sense. In particular, under Shorty's mistaken assumption that $3+4=6$, it is necessarily the case that $3+4+3=6+3=9$. Cat, on the other hand, missed this important generalization and gives a very puzzling answer to the last question. Instead of adding, Cat suddenly concatenates the numbers. That is extremely bewildering, and a teacher who only evaluates students based on how many answers they get wrong would miss this.

You might object that the problem here isn't with counting the number of mistakes, but rather with the small number of questions we asked. If we had asked more questions of the form $3+4+3$, Cat would have done worse than Shorty. You are correct, and this is also the answer you will hear in NLP. NLP models are tested on millions of sentences, which the example above obviously does not do justice to. But the size of a test sample is no solid indicator of its quality. What if Cat's problem only occurs with sequences of addition of the form $n+(n-1)+n$? Even in a test set with millions of questions, this pattern might be fairly rare, whereas Shorty's problem with adding 3 and 4 might crop over and over again. With addition it is fairly easy to design a balanced test set, but with language NLP engineers rely largely on the data that is freely available in the wild. This includes tweets, posts on message boards, news paper articles, and so on. Linguists have argued for a long time that these samples are not representative of the richness of language. The most intricate aspects of language tend to reveal themselves only rarely, but when they do they are of great importance.

As a result, NLP performance metrics should be taken with a grain of salt. Yes, model A might outperform model B by 10 percentage points in your benchmark. But model B might still make a better impression on users because its errors aren't nearly as serious or puzzling as those made by model A. When interacting with humans, slightly wrong beats really wrong, and wrong beats weird.

Such qualitative performance metrics are still sorely missing in NLP. A good theoretical grounding — where "theory" subsumes both theoretical linguistics and the theory of computation — is a guide through those areas of language where data is insufficient. The more language-like a model, the less data it needs to display language-like behavior.

In sum, then, NLP approaches still have their fair share of shortcomings that insights from computation linguistics can help address. As NLP moves into increasingly complex domains of language, with increasingly impoverished data, the use of a strong theoretical foundation will become increasingly apparent.

An example from the history of computational linguistics

Overall, then, there is definite potential for improving NLP with rich computational theories of natural language. A brief look at the history of computational linguistics furnishes multiple examples of that.

Consider the problem of word structure, which linguists call *morphology*. A native speaker of English knows that *liked* is the combination of the verb *like* and a past tense marker, and in the other direction he or she also knows that the addition of a past tense marker turns *walk* into *walked*, *run* into *ran*, and *go* into *went*. Linguists have collected a detailed inventory of morphological processes, and computational linguists have developed the framework of finite-state morphology to model these processes. These two steps did not take place independently of each other. The crucial contribution of linguists was to point out that the range of possible morphological processes is very limited. Once computational linguists saw what is (im)possible in morphology, they realized that almost all the processes are very simple and can be captured with very restricted tools that are still very flexible and scalable, which is exactly what one wants for practical purposes. Nowadays, finite-state machinery is still used for modeling specific aspects of morphology, e.g. number systems. The ideas have undergone a voyage from theoretical linguistics to computation linguists to NLP, where they have proven tremendously useful for over 30 years now. But insights have also traveled in the other direction. A few processes in morphology that involve copying pose serious challenges to finite-state morphology, highlighting the need for a better understanding of these phenomena. This way theoretical and computational linguists feed each other's research, with NLP as the happy consumer of the final results.

Sometimes the transmission of insights is almost imperceptibly indirect, and is easily lost in fogs of history. Formal language theory — which is still an integral part of computational linguistics but also indispensable for the design of programming languages and file standards like XML — grew out of Noam Chomsky's linguistic theories of natural language. Chomsky's Transformational grammar also served as an important inspiration for William Rounds's work on tree transducers, and those are now used in machine translation (outside of computational linguistics, they are also heavily used in compiler design, which one may regard as machine translation for programming languages). The computational linguist Aravind Joshi relied on key insights about the nature of language when he developed Tree Adjoining Grammars (TAG), which are still highly influential and have even been used to model biological structures such as messenger RNA. The same can be said about the formalism of *Combinatory Categorial Grammar*, which is equally driven by engineering concerns and pure linguistic theory. As you can see, there is no telling how fields will cross-fertilize each other as their ideas and insights slowly transgress disciplinary boundaries. But one can certainly tell that his has happened numerous times between computation linguistics and NLP. The claim that NLP is doing perfectly well on its own and could not possibly benefit from computation linguistics does not hold up to scrutiny.

Interim Summary

In sum, NLP is no different from other fields in that it, too, can profit from more theoretically minded endeavors. This holds whether the theory is grounded in mathematics, computer science, or linguistics, each one of them has already made valuable contributions and will continue to do so. The value of theory is not constant across all applications. Hard problems tend to benefit more than easy ones. It also depends on the quantity and quality of available data, and the need for strong safeties — minor mistranslations are acceptable for a blog post, but not in a business contract. But it is also important that the problem is well-defined — if it isn't even clear what exactly the problem is, theory will be of very limited use. This is why Twitter-based mental health assessment requires a healthy dose of opportunistic trial-and-error with all available tools, whereas computational morphology, for instance, benefits a lot from a good theoretical foundation. So no, computation linguistics isn't the magic bullet for solving NLP, but it has valuable contributions to make in key areas. These key areas may well become the central focus of attention soon. If the current staples of NLP end up being subsumed by general purpose machine learning techniques, we may see the field move towards harder problems that are ripe with language-specific issues.

2.2 Scientific arguments

The previous section has defended computation linguistics in terms of its utility for NLP. These utilitarian arguments are needed to convince politicians, taxpayers, and engineers. But they hold no sway in the court of science. A linguist might shrug at the arguments above on a good day and publicly denounce us as nimrods on a bad one. If computation linguistics can somehow get NLP to incorporate more linguistic insights, that's nice and dandy. It probably won't tell us much about language, though. So it won't improve linguistics, psychology or cognitive science. Fortunately, the scientific merit of computation linguistics is a lot more clear-cut than the utilitarian argument: since language is intrinsically a computational problem, studying language from a computationally informed perspective is only natural.

Probably the most important development in 20$^{\text{th}}$ century linguistics is the *cognitive turn*, the shift from viewing language as a disembodied system of rules and words to its reinterpretation as one of humans' many cognitive abilities. Language is not an abstract platonic object that exists independent of reality. It is done by humans, it is an algorithm that runs on the hardware of the human brain. Since the brain is pretty moist and mushy, its hardware is often called the *wetware* to emphasize the contrast to man-made machines. Specific languages are just specific externalizations of this abstract program "language" that runs on the wetware. Languages are but a window into a specific kind of computation that the human brain carries out with little effort. Since the cognitive turn, theoretical linguistics is no longer about languages, it is about language.

But if language is a computation carried out by wetware, this immediately raises the question how exactly language is computed. Linguists actually split this up into two subquestions:

Competence What is the specification of the computations?

Performance How is this specification implemented and used?

Competence questions are concerned with the rules of grammar and how they are encoded. A native speaker of English, for instance, recognizes that in the two sentences below, only the first one is ambiguous.

(1) a. Who do you want to leave?
    b. Who do you wanna leave?

The first question has two meanings, paraphrased as "Who is such that you want them to leave?" and "Who is such that you want to leave them?", respectively. The second question only allows the latter interpretation. Competence describes the implicit knowledge of a speaker that is needed for him or her to recognize such a contrast. But the actual process of bringing competence to bear on these questions is a much more involved matter, and that is what performance is about.

Somewhat sloppily, one could describe competence as "language modulo the resource restrictions on working memory and attention span". Of course nothing of this sort can be observed in nature — competence is an artificial abstraction of performance, which is about how the specification behaves when it is run on an actual machine, i.e. the wetware. The distinction between competence and performance also exists in computer science to some extent. For example, complexity theory studies the difficulty of problems of unbounded size, even though in practice problems are usually bounded, for instance because a computer can only hold so much information before it runs out of memory. Nonetheless complexity theory has produced results that are also relevant in practice. Competence questions in linguistics have similarly shed light on language despite their high level of abstraction.

Since competence cannot be directly observed, research into how language is computed usually operates in the realm of performance. Linguists approach this questions through the lens of neuroscience and psychology: how does the wetware behave when carrying out specific linguistic tasks, and can we design a procedure that mimics humans' behavior? For example, native speakers of English usually have no problem understanding English sentences, it is an incredibly fast and effortless process. But it does exhibit surprising quirks. When asked whether the sentence (2) is grammatically correct, native speakers usually say no.

(2) The player tossed a frisbee smiled.

However, the sentence is actually well-formed, it has the same structure as the minimally different (3).

(3) The player thrown a frisbee smiled.

For some reason the algorithm native speakers of English use has no problem with (3) but is completely thrown off by the exchange of a single word that serves exactly the same grammatical function. This is almost like a computer that can compute $1 + 2$ but not $2 + 1$. There is no obvious reason for this behavior, and it doesn't exactly look like good engineering (so much for Intelligent Design). Linguists have come up with various elaborate explanations of such phenomena over the years — some more successful than others — but they all share a variety of properties that necessarily limit their scope and thus the questions they can address. Where the reach of these approaches ends, the realm of computation linguistics begins.

Neuroscience

Neuroscience has made tremendous progress in the last few decades, and as a result there has also been increased interest in neurolinguistics. As in many other areas of neuroscience, the hope is that the "code" of the human brain will reveal itself through its wetware. Know the hardware, and you know the software; unfortunately, things are not that straight-forward.

It is certainly interesting to see how the human brain works on the hardware level. But it is far from obvious that this tells us much about the actual computations. Nothing we have learned about computation since Turing's pioneering work supports this notion, intuitive as it might be. Studying a computer's hardware tells us little about what it is computing. Suppose your computer still has an old rotary hard drive, rather than an SSD. When we hear your hard drive spin up, we can reasonably assume that some file is being accessed, but that is about all we can deduce with our own senses. Inquisitive minds that we are, we may then decide to connect a set of thermal diodes to various points of the hardware in order to detect whether this or that piece of hardware starts heating up. Increased heat output would suggest an increase of computational activity in that specific area. A hot graphics chip, for instance, would be indicative of some high graphical load. But that is still very inconclusive, for various reasons.

First of all, a higher load on the graphics chip might indicate that some kind of 3D graphics is being rendered, e.g. for a video game. Actually, though, a lot of non-graphical tasks are outsourced from the processor to the graphics chip nowadays. This is why the acronym GPU, which stands for *Graphics Processing Unit*, has been extended to GPGPU, which is short for *General Purpose Computing on GPUs*. Among other things, GPUs are now commonly used in the training of neural networks, which at an intuitive level has very little to do with graphics rendering. Mathematically, the differences are less pronounced. This is exactly why it is difficult to draw conclusions from hardware to software — the former is capable of a lot more than one would naively assume.

A curious example of this principle is the first-person shooter *Doom*, well-known for its high-adrenaline gameplay and impressive music. When this classic video game was ported from PC to Atari's *Jaguar* console, buyers and reviewers alike lamented the lack of any in-game music. The reason for that is surprising and insightful at the same time. At the time, Doom was a very demanding game, too demanding for the Atari Jaguar. Its CPU simply wasn't beefy enough to handle the game. The developers, however, realized that some of the game functions could be off-loaded to another chip, freeing up more CPU cycles for the rest of the game. But this meant that this other chip could no longer serve its intended purpose as it was busy with other computations. Yes, you guessed it, that chip's standard role was to handle the music. There was no other chip with free computing cycles for the music, and thus the music had to be completely cut from the port. While ultimately unsatisfying for Atari Jaguar owners, the developers' decision to co-opt a music chip for general purpose computation was genius.

Admittedly the human brain is not as content-agnostic as a modern computer, specific tasks seem to be tied to specific areas of the brain. But it is too early to say how tight this link actually is from a computational perspective. Suppose task X is localized in brain area A. Then a computationally similar task Y may or may not be localized to area A. In the other direction, not every task Z that is localized to A is necessarily computationally comparable to X. There is no strict implication in either direction. The locus of computation seems to depend more on content than the type of computation.

There is also a severe granularity mismatch between hardware and computation that makes it very hard to make meaningful inferences from the former to the latter. Even a simple task like searching through a list can be accomplished in various ways, each with their own advantages and disadvantages. The two most fundamental strategies are linear search and binary search.

Example 1.2 Linear search and binary search

Suppose you ask somebody to memorize the list A, C, D, F, G, X. Then you ask them if the list contained the symbol G. How could they answer your question? I honestly don't know how a human does it, nor does anybody else. But for a computer, there are at least two strategies: *linear search* and *binary search*.

In linear search, we simply scan the list from left to right and check one symbol after the other. If we find the symbol we are looking for, we reply that it is in the list. If we make it through the whole list without ever seeing the requested symbol, we reply No. The table below shows what happens at each step of the computation.

Step	**Symbol**	**Remaining list**
0		A, C, D, F, G, X
1	A	C, D, F, G, X
2	C	D, F, G, X
3	D	F, G, X
4	F	G, X
5	G	done

As you can see, linear search finds the requested item in 5 steps.

Binary search is a more efficient search strategy that exploits a specific property of the list. While in principle the symbols in a list can appear in any random order, our list A, C, D, F, G, X lists the symbols in alphabetical order. This fact can be used to avoid searching through irrelevant parts of the list.

Instead of starting at the beginning of the list, binary search goes straight to the middle. If the middle point is the symbol we are looking for, we stop there. If the symbol there is alphabetically **before** the searched symbol, then we throw away the **left** half of the list. After all, if the middle symbol alphabetically precedes the symbol we are looking for, then so do all the symbols to the left of the middle symbol. Hence it makes no sense to search there. After the left half has been removed, we perform binary search on the remainder, i.e. the right half. On the other hand, if the symbol at the middle point is alphabetically **after** the search symbol, then we throw away the **right** half of the list and perform binary search on the remainder, i.e. the left half. We continue this way, throwing away one half in each step and looking at the middle point of the remaining half. If the list contains the item we are looking for, we will find it eventually. If at some point we have thrown away the whole list, the item wasn't in the list.

Step	Symbol	Remaining list
0		A, C, D, F, G, X
1	D	F, G, X
2	G	done

Binary search can be much faster than linear search. The further to the right in a list an item occurs, the longer linear search will take compared to binary search. That's because binary search can skip large chunks of the list.

On average, binary search is much faster than linear search for sorted lists. So if we had two computers with exactly the same hardware, one using linear search and the other binary search, the latter would show vastly superior performance in general. Yet the behavior cannot be explained in terms of hardware, because we know for a fact that the two machines are exactly the same. This is exactly why good algorithms and data structures are so important in computer science — progress on the hardware side cannot match the massive speed-ups of smart software. The ingenuity of the human brain probably does not lie in its hardware, at least not exclusively. It stands to reason that millions of years of evolution have resulted in algorithms that have been tweaked, tuned, and optimized to be as efficient as possible for their respective domain. Hardware alone is not enough.

Keep in mind that in the case of computers, we have the additional advantage that we already know how their architecture works because we designed it. So if that still is not enough to deduce from the hardware what kind of computations a computer is carrying out, it seems rather unlikely that a similar process could derive from wetware the functioning of the human brain. Granted, we can uncover limiting factors and some basic facts, just like a close analysis of a processor can reveal a maximum limit on its memory and that all information is encoded in a categorical fashion via series of on and off states — the proverbial 0s and 1s. But all the probing, all the high-tech machinery tells us very little about the actual computations.

Thus it isn't exactly surprising that no neuroscientist on this planet knows whether at least some computations carried out by the wetware of the human brain involve anything resembling linear search or binary search. But this is one of the simplest distinctions that can be made when it comes to search. Many of the concepts discussed in this book are much more abstract, which makes it even harder to bridge the gap between the hardware/wetware-level and computation. It might be possible to do so if one approaches the issue from both directions. But a unidirectional process that seeks to build a computational theory of the human mind, or even just language, purely in terms of how the wetware operates is very unlikely to succeed due to the enormous mismatch in granularity and the concomitant problem of underspecification. In the domain of language, we have the potential for a bidirectional approach that proceeds both top-down (from computation to hardware) and bottom-up (from hardware to computation). Computation linguistics can contribute towards such a linking theory by identifying computational core mechanisms of language that are sufficiently general that they may be detectable in the wetware with current day techniques.

Psychology

In contrast to neuroscientists, psychologists don't investigate the physical instantiation of cognition. They are perfectly happy to treat the brain's wetware as a black box that produces a certain output (= reaction) given a certain input (= stimulus). Their goal is to develop models that replicate these input-output mappings. But again there are certain limitations that computation linguistics can help address.

One central issue is again a granularity mismatch between the explanandum — language as a cognitive ability — and the means of explanation. Psycholinguistic models can be highly specific in what cognitive parameters they presuppose.

Example 1.3

A psycholinguistic account of priming A common assumption in the literature is that humans use *content addressable memory* (CAM). CAM operates very differently from a computer's *random accessible memory* (RAM). RAM is similar to a very long street where one can instantaneously travel to any house as long as one knows its house number. So if a computer has stored some item in memory at a given address, all one needs is this address to immediately retrieve the item from memory. Information stored in CAM, on the other hand, is not retrieved via an address that specifies its precise location in memory. Instead, the individual pieces of the information themselves act as a way of specifying the path to its location in memory. Imagine traversing a large, maze-like network of intersecting roads, where every road has a descriptive title such as *doctor*, *crab*, and *cartoon*. At the point where those three roads intersect, you will find the house of Doctor John A. Zoidberg, the alien crab doctor from the cartoon show *Futurama*. There is no longer a need for arbitrary addresses because the properties of any given thing carve out the path for finding this thing. Psycholinguists couple CAM with certain models of memory activation and retrieval to explain specific aspects of human cognition, e.g. that memory retrieval for item X takes less time if a content-related item Y already had to be retrieved immediately before (this is known as *priming*).

A very extreme case of this is the ACT-R framework, an elaborate computer model of human cognition. The specificity of ACT-R allows researcher to run detailed simulations and tweak parameters to find the best fit for experimental data. ACT-R is an impressive achievement, but it is a machine with many turning cogs and gears. This makes it difficult to ascertain what each component contributes to the final result. This has to downsides. First, the findings aren't as insightful as one might hope. Simulations may tell us that configuration X accounts for the facts, but what about X' or X''? Do we need all the components interacting in this specific manner, or are most assumptions dispensable except for maybe one or two?

You may think that we can answer this by just running additional simulations with other parameter values. But this is often not practically feasible due to combinatorial explosion. If your model has 10 binary parameters, then there are $2^10 = 1024$ different instantiations of the model. Even if your simulation takes only half an hour to complete, you're already looking at 512 hours, which is over three weeks. In practice, your model will take longer to run and have many more parameters to tweak, so exhaustive simulation of all options isn't feasible. Computation linguistics, on the other hand, has the advantage that its concepts, while abstract, are sufficiently simple to be explored from a mathematical perspective using little more than pen and paper. While this certainly poses its own challenges, it often takes much less time and effort than designing models and running simulations.

Putting aside the time investment factor, though, we are still left with the problem that one cannot generalize from modeling results in a monotonic fashion. That model M captures phenomenon P does not entail that it also accounts for phenomenon P'. Once we make tweaks to account for P', we have to rerun our simulations for P to ensure that everything still works as desired. The mathematical approach favored by computation linguistics, on the other hand, provides crucial clues under which conditions key properties are preserved, and this makes it easier to make inferences from P to P'.

This is not to say that the modeling approach is inferior. It has the major advantage that it is close to the empirical realities. The models are complex, but their behavior and predictions can be directly tested against the observed data. The highly abstracted and idealized concepts of computation linguistics require more effort to bring to bear on the data. Some problems are so complex that it is far from obvious how they should be idealized in order to make them amenable to mathematical investigation. Just as with neuroscience, this isn't a case of one approach supplanting the other, but rather of natural complementation.

Some psycholinguists may object, though, that the previous paragraphs only engage with a particular subpart of their field, namely the cognitive modeling approach. Even if one grants that all the postulated shortcomings are on the mark, that would only apply to a small part of psycholinguistics. The overwhelming majority of psycholinguists operates with less specific models and theories. These models may be too coarse for computational simulations, but they work just fine as a generator of new research questions that can be explored through experiments. While correct, these models face the opposite problem of lacking the fine-grained details that would make them sensitive to issues of linguistic competence and theoretical linguistics. They cannot distinguish between competing analyses or illuminate whether language is rule-based or constraint-based. Computation linguistics provides the scaffolding to explore these issues and identity properties that might be detectable in psycholinguistic experiments.

Interim summary: The promise of computation linguistics

The limitations of neuroscience and psychology pointed out above can be traced back to a comparatively low degree of abstraction. Both fields operate at levels that specify a lot of information — like memory addressing and neural connections — whose relevance to language isn't apparent; in particular if one cares mostly about competence questions, as most theoretical linguists do. The great promise of computation linguistics, the one advantage that sets it apart from neuroscience and psychology, is one of *productive abstraction*. Computation linguistics can completely abstract away from all extraneous detail and performance aspects without losing the connection to cognition. The methods used by computation linguists allow us to connect language and computation (and hence cognition) at the competence level. We can tackle questions such as

- What is the weakest memory architecture that is sufficiently powerful to support a specific model of competence?
- What is the weakest competence model that is sufficiently powerful for a given empirical domain, e.g. morphology.
- Do alternative competence models describe the same class of computations?
- Is there an alternative representation of a given model that lowers memory requirements?
- How can we carve up complex models into simple subparts?
- What kind of computational universals hold of language? That is to say, what kind of computational properties hold of every natural language?

3 The virtue of abstractness

3.1 Marr's three levels of analysis

The preceding observations regarding abstractness are far from new. In 1976, David Marr and Tomaso Poggio formulated the *Tri-Level Hypothesis* in their paper *From Understanding Computation to Understanding Neural Circuitry*. They proposed that computational processes, including cognition, should be described on three levels of increasing abstraction:

physical the wetware or hardware instantiation; e.g. neural circuitry for vision or the machine code running on your computer

algorithmic what kind of computational steps does the system carry out and in which order, what are its data structures and how are they manipulated; many aspects of programming deal with issues at this level

computational what problem space does the system operate on, how are the solutions specified (but not necessarily carried out)

Example 1.4 Set intersection on three levels

Suppose you have two sets of objects, A and B, and you want to write a computer program that tells you which objects belong to both sets.

- On a computational level, that program is easily specified: it takes two sets as input and returns their intersection ($A \cap B$ in mathematical notation).
- On the algorithmic level, things get trickier. For instance, do you want to store the sets as lists or something else, and just how does one actually construct an object that is the intersection of two sets?
- On the physical level, finally, things are so complicated that it is nigh impossible to tell what exactly is being computed by the machine. Voltages increase or decrease in various transistors spread over the CPU, memory and mainboard, and that's about all you can make out. In the human brain, neurons are firing in intricate patterns that somehow give rise to the desired output. Unless you already have a good idea of the higher levels and the computational process being carried out, it is probably hopeless to reverse engineer the program from the physical evidence.

David Marr (1945-1980)

David Marr was a British neuroscientist who did revolutionary work on visual processing. Marr took a very pragmatic approach to science. He maintained that working on concrete empirical problems is more fertile than sterile debates about methodology and theoretical primitives (so he probably wouldn't have liked this chapter). History presumably proved him right, as it is very doubtful that his Tri-Level Hypothesis would have caught on if it weren't for the strengths of his model of vision, which combines the physical, algorithmic, and computational level for maximum effect.

The tri-level hypothesis highlights that one and the same object can be described in very different ways, and all three levels are worth studying. A computational specification can be implemented in various distinct ways on the algorithmic level, and algorithms can be realized in a myriad of physical ways — for instance, your laptop and your phone use very different processor architectures (x86 and ARM, respectively), but a Python program will run just fine on either platform despite these differences. And of course this hierarchy is continuous: Assembly code is closer to the physical level than the code of the programming language C, which in turn is closer to the hardware than Python. However, the more you are interested in stating succinct generalizations, the more you will be drawn towards abstractness and hence the computational level at the top of the continuum. And this is exactly the level computation linguistics is aiming for.

3.2 Abstraction necessitates formal rigor

The problem with abstraction is that one can no longer reason on a purely intuitive level. Since the objects are characterized by a few basic properties, it is important that these properties are described as precisely as possible. A minor misunderstanding may be enough to lead to completely contradictory conclusions. In the worst case, we may end up with an *inconsistent* theory, which means that there is at least one property that is both true and false at the same time. This may be perfectly fine in a post-modern analysis of transphobic slurs in humoristic epitaphs, but it has no place in a scientific theory. So abstraction necessarily requires a certain degree of care and rigor.

This shouldn't come as a big shock to you. Computer science can be very rigorous, in particular its theoretical subfields like complexity theory and formal language theory. The same is also true of linguistics: Generative syntax and phonology are abstract and involve a lot of technical machinery that seems arcane and intimidating to outsiders. The technical machinery is indispensable for each field's areas of scientific inquiry, and you all got the hang of it eventually after a few initial struggles.

The same is true of the machinery we will use in this course. It is more technical than linguistics, but only because we cannot make do with less. It does involve some math, but nothing that one couldn't pick up in a week. It is harder to read at the beginning, but ultimately notation will make reading easier and faster. You will get stuck sometimes, but that just means you have to think about the problem a couple more times until you get it. Nothing we do in here is truly difficult, but it takes patience and dedication. If some of the material covered in this book seems overwhelming to you and makes you doubt your own intellect, remember that the most important ingredient for a scientist is the ability to enjoy feeling stupid:

> “The more comfortable we become with being stupid, the deeper we will wade into the unknown and the more likely we are to make big discoveries. (Schwartz 2008)”

4 Summary

1. **Computation** and computers are not the same. Computation is an abstract concept that can be instantiated by various objects, including non-existing ones. Computers are one such object, but so is the human brain, a cell, or a line of water buckets when manipulated appropriately.

2. **Turing machines** show how little is needed to carry out very complex computations. A movable read-write head, a state register, and an unlimited tape are sufficient to solve even the most complex computations. But figuring out the correct set of instructions may be hard.

3. **Natural language processing** (NLP) is focused on solving language-related tasks with computers. A different aspect of computational linguistics (which I call **computation linguistics** for lack of a better term) instead seeks to study language as a computational problem. This is the approach presented in this textbook.

4. A cognitive system like language can be described on **Marr's three levels of description**. These are, in increasing order of detail: *computational*, *algorithmic*, and *physical*. The computational level allows us to abstract away from many details that are irrelevant for the issues linguists care about.

5. Abstraction is a virtue. It allows for profound insights and succinct generalizations that could not be stated at more fine-grained levels.

6. Neurolinguistics and psycholinguistics operate at less abstracted levels, as do approaches based on computational modeling. Each one has valuable contributions to make, but integrating insights from the different levels remains one of the hardest problems in cognitive science.

Relevant literature for Unit 1

Computation linguistics and related fields One of the central themes of this chapter is the difference between NLP on the one hand and computation linguistics on the other. This contrast is very salient when comparing Wilks's (2006) history of NLP to Penn's (2006) overview of symbolic computational linguistics (which is more closely related to the computation linguistics presented here). It is particularly striking how differently the two papers view the historical role and contribution of linguistics. The papers might be a difficult read at this point, but even if the jargon and concepts are still opaque, the spiritual difference between the two approaches emerges clearly.

Pullum & Kornai (2003) and chapters 1 and 10 of Kornai (2007) present an overview of *mathematical linguistics*. As the name suggests, mathematical linguistics studies language with the help of mathematics. This may sound very similar to computation linguistics, but the two are not the same. Mathematical linguistics is defined by its mathematical methodology and can be brought to bear on any question. Computation linguistics, on the other hand, is a subfield of linguistics that explores the computational aspects of language. In practice, though, the two have a very large intersection of researchers.

Krahmer (2010) discusses the interplay of computational linguistics and psycholinguistics. Crocker (2010) is a more extensive survey of the subfield of computational psycholinguistics. Hale (2014) carefully develops a powerful framework for studying sentence processing from a computational perspective. There are numerous publications on ACT-R, but a computationally minded reader is best served by Whitehill (2013).

Alan Turing The Turing Machine was first discussed in Turing (1936, 1938). Hodges (1983) is a gripping biography of Alan Turing; it had been out of print for a long time, but was reprinted after the release of Hollywood's biopic *The Imitation Game*, with Benedict Cumberbatch as Alan Turing. Turing has also been tremendously influential in the domain of artificial intelligence. For a technically light-weight but sprawling pop-science introduction to this topic, see Hofstader (1979).

Linguistics beyond language Section 2.1 lists several examples where approaches from theoretical and/or computational linguistics made their ways into seemingly unrelated applications. All these approaches will be discussed more extensively in later chapters, but the curious reader may already take a gander at the following references:

Formal language theory had its beginnings in Chomsky (1957, 1959) and Chomsky & Schützenberger (1963). One of the first papers on tree transducers is Rounds (1970), who explicitly names Chomsky's Transformational grammar as a major motivation for his work. Tree Adjoining Grammar (Joshi 1985) is a different, computationally minded grammar formalism. For applications of TAG in molecular biology see Uemura et al. (1999) and Matsui, Sato & Sakakibara (2005).

Other references The study on predicting the success of novels is Ashok, Feng & Choi (2013). The tri-level hypothesis is formulated in Marr & Poggio (1976) and Marr (1982).

Ashok, Vikas Ganjigunte, Song Feng & Yejin Choi. 2013. Success with style: using writing style to predict the success of novels. In *Emnlp 2013*, 1753–1764.

Chomsky, Noam. 1957. *Syntactic structures*. The Hague: Mouton.

Chomsky, Noam. 1959. On certain formal properties of grammars. *Information and Control* 2. 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6. https://doi.org/10.1016/S0019-9958(59)90362-6.

Chomsky, Noam. 1990. On formalization and formal linguistics. *Natural Language and Linguistic Theory* 8. 143–147.

Chomsky, Noam & M. P. Schützenberger. 1963. The algebraic theory of context-free languages. In P. Braffort & D. Hirschberg (eds.), *Computer programming and formal systems* (Studies in Logic and the Foundations of Mathematics), 118–161. Amsterdam: North-Holland.

Crocker, Matthew. 2010. Computational psycholinguistics. In Alex Clark, Chris Fox & Shalom Lappin (eds.), *Handbook of computational linguistics and natural language processing*, 482–514. London: Blackwell.

Hale, John. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI.

Hodges, Andrew. 1983. *Alan turing: the enigma*. Simon & Schuster.

Hofstader, Douglas. 1979. *Gödel, Escher, Bach: an eteneral golden braid*. 1999 Anniversay Edition. Basic Books.

Joshi, Aravind. 1985. Tree-adjoining grammars: How much context sensitivity is required to provide reasonable structural descriptions? In David Dowty, Lauri Karttunen & Arnold Zwicky (eds.), *Natural language parsing*, 206–250. Cambridge: Cambridge University Press.

Kornai, Andras. 2007. *Mathematical linguistics*. New York: Springer.

Krahmer, Emiel. 2010. What computational linguists can learn from psychologists (and vice versa). *Computational Linguistics* 36. 285–294. https://doi.org/10.1162/coli.2010.36.2.36201.

Marr, David. 1982. *Vision: a computational investigation into the human representation and processing of visual information*. Reprinted in 2010 by MIT Press. New York: Freeman. https://doi.org/10.7551/mitpress/9780262514620.001.0001.

Marr, David & Tomaso Poggio. 1976. *From Understanding Computation to Understanding Neural Circuitry*. Tech. rep. Artifical Intelligence Laboratory, MIT, AIM-357.

Matsui, Hiroshi, Kengo Sato & Yasubumi Sakakibara. 2005. Pair stochastic tree adjoining grammars for aligning and predicting pseudoknot RNA structures. *Bioinformatics* 21. 2611–2617.

Penn, Gerald. 2006. Symbolic computational linguistics: overview. In Keith Brown (ed.), *Encyclopedia of language & linguistics*, 338–352. Amsterdam: Elsevier. https://doi.org/10.1016/B0-08-044854-2/00928-7.

Pullum, Geoffrey K. & András Kornai. 2003. Mathematical linguistics. In *Oxford international encyclopedia of linguistics*, 2nd edn., 17–20. Oxford: Oxford University Press.

Rounds, William C. 1970. Mappings on grammars and trees. *Mathematical Systems Theory* 4. 257–287.

Schwartz, Martin. 2008. The importance of stupidity in scientific research. *Journal of Cell Science* 121. 1771–1771. https://doi.org/0.1242/jcs.033340.

Turing, Alan M. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42. 230–265. https://doi.org/10.1112/plms/s2-42.1.230.

Turing, Alan M. 1938. On computable numbers, with an application to the Entscheidungsproblem: a correction. *Proceedings of the London Mathematical Society* 43. 544–546. https://doi.org/10.1112/plms/s2-43.6.544.

Uemura, Yasuo et al. 1999. Tree adjoining grammars for RNA structure prediction. *Theoretical Computer Science* 210. 277–303.

Whitehill, Jacob. 2013. Understanding ACT-R — an outsider's perspective. Ms. UCSD. http://arxiv.org/abs/1306.0125.

Wilks, Yorick. 2006. Computational linguistics: history. In Keith Brown (ed.), *Encyclopedia of language & linguistics*, 761–769. Amsterdam: Elsevier. https://doi.org/10.1016/B0-08-044854-2/00928-7.

Exercises

[Exercise 1.1 Consider once more the Turing machine from example 1.1. Show how this Turing machine operates on an input of the form 011110, with the read-write head starting on the rightmost 1 in state A. Since the tape of a Turing machine can be expanded without limits, you may pad it out with 0s as necessary. This does not require any special instruction for the machine.]

[Exercise 1.2 Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: 0, 1, and $\diamond$ as a special symbol for cells that do not contain one of the two digits.

2. The Turing machine takes as its input a single string of 0s and 1s (e.g. 001101 or 1010110).

3. All other cells of the tape are filled with $\diamond$.

4. The machine starts on the leftmost symbol of the input.

5. The machine outputs a string that is almost exactly the same as the input, except that the first symbol is replaced by $1 - n$, where n is the value of the last symbol. For instance, 11101 becomes 01101, whereas 01101 and 11100 stay the same.

]

[Exercise 1.3 Continuing the previous exercise, modify your machine so that it works even if the read-write head can start on any random symbol of the input string.]

[Exercise 1.4* Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and ⋄ as a special symbol for cells that do not contain a digit.
2. The Turing machine takes as its input a single number n between (and including) 0 and 999.
3. The machine starts on the leftmost digit of the input.
4. The machine outputs $1000 - n$ (leading 0s are allowed, e.g. 0001 instead of 1).

To save yourself some writing, you may conflate multiple lines by using variables. So if 7 is rewritten as 3 and 8 as 2, you can just have a line rewriting n as $10 - n$ and adding "n is 7 or 8". As long as there are only finitely many choices for n, this does not change anything about the machine, it is just a way to represent multiple instructions with a single line.]

[Exercise 1.5 Continuing the previous exercise, look at the specification of your Turing machine. Is there any individual part that captures the "essence" of substraction? Or does the whole come about only through the interaction of the individual parts?]

[Exercise 1.6 The practical argument for computation linguistics presupposes that man-made engineering solutions can be improved by learning from nature — in the case of NLP, human language use. But at the same time, airplanes aren't built like bumblebees, and submarines aren't fish. Doesn't this suggest that technology carves out its own way, with its own solutions? Formulate an extended argument for this position, and one against it. Which one do you find more convincing?]

[Exercise 1.7 Consider once more the sorted list A, C, D, F, G, X. Using the tabular format from example 1.2, show how linear search and binary search move through the list when looking for item C.]

[Exercise 1.8* Binary search is only guaranteed to work correctly with ordered lists. But this does not imply that binary search necessarily fails on unordered lists. Write down an unordered list with 5 elements A, B, C, D, E such that binary search will find A but not E.]

5 Code

5.1 Linear search and binary search

Linear search is one of the simplest search algorithms and can be implemented in 4 lines. The code here implements it as a function that returns either `False` or an integer. `False` is returned iff the search item is not in the list. Hence when an integer is returned, it indicates the leftmost position of the search item.

Note that the docstring specifies the intended type of each argument. The type of the search item is set to `any`, which means that there are no restrictions on its type.

```
def linear_search(search_list, item):
    """Left-to-right search for position of item in search_list.

    Parameters
    ----------
    search_list : list
        list to be searched
    item : any
        item we are looking for

    Returns
    -------
    int or False

    Examples
    --------
    >>> linear_search([0,1,7,9], 7)
    2

    >>> linear_search([0,1,7,9], 8)
    False
    """
    # iterate over all positions in the list
    for i in range(len(search_list)):
        if item == search_list[i]:
            return i
    # if we made it this far,
    # then we haven't found anything
    return False
```

```
    >>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
    >>> linear_search(test_list, 'e')
    3
    >>> linear_search(test_list, 'g')
    False
```

[**Exercise 1.9** What is the output of `linear_search([1,1,1,1], 1)`?]

[**Exercise 1.10** In Python, `0` is sometimes treated the same as `False`. This can result in some unexpected bugs, for instance with the piece of code below. What exactly is the problem? How would you fix it? Is this an issue that a high-level definition of linear search should address?

```
if linear_search([1,2,3,4], 1):
    print("Item in list!")
else:
    print("Item not in list!")
```

]

Now let's contrast linear search against binary search. As you can see, the code is slightly different from the intuitive description in example 1.2. There we said that binary search picks the middle item of the list. If the middle it is not the search item, one discards one half of the list and runs binary search on the remainder. This is a recursive definition: an instance of binary search can spawn another instance of binary search. While recursion definition is more elegant and mathematically pleasing, it makes for a bad Python implementation because Python is not well-optimized for recursion. The implementation below uses a while-loop instead, which is much more performant for Python. With another programming language, e.g. Haskell, the recursive definition might be the most natural and efficient implementation.

```python
def binary_search(search_list, item):
    """Use binary search to find position of item in search_list.

    This algorithm is more efficient than linear search,
    but only works for sorted lists!

    Parameters
    ----------
    search_list : list
        list to be searched
    item : any
        item we are looking for

    Returns
    -------
    int or False

    Examples
    --------
    >>> binary_search([0,1,7,9], 7)
    2

    >>> binary_search([0,1,7,9], 8)
    False
    """
    start = 0
    end = len(search_list) - 1

    while start <= end:
        # pick middle element of list;
        # we use int() for rounding down
        middle = int(start + (end - start)/2)

        # Case 1: our item is to the left of the item at the midpoint,
        #         limit next search to left half
        if item < search_list[middle]:
            end = middle - 1
        # Case 2: our item is to the right of the item at the midpoint,
        #         limit next search to right half
        elif item > search_list[middle]:
            start = middle + 1
        # Case 3: we found our item, return its index
        elif item == search_list[middle]:
            return middle
    # Case 4: item not in list, while-loop aborted
    return False
```

```
>>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
>>> binary_search(sorted(test_list), 'e')
4
>>> binary_search(sorted(test_list), 'g')
False
>>> binary_search(test_list, 'c')
1
>>> binary_search(test_list, 'e')
False
```

This shows very clearly that one and the same idea can be implemented in many different ways. Which way is the best often depends on very subtle and arcane details. These details often end up obscuring what really matters, and this is why we should always strive for a level of description that abstracts away from implementation-dependent differences.

[Exercise 1.11 Write a recursive implementation of binary search.]

[Exercise 1.12 What would a recursive implementation of linear search look like? Compared to binary search, do you find the recursive implementation of linear search simpler, more convoluted, or about the same in complexity?]

5.2 Set intersection

Example 1.4 briefly raised the question how exactly set intersection could be implemented at the algorithmic level. Python actually provides an out-of-the-box solution in the form of a set data type that provides the method `intersection`.

```
>>> A = {1, 2, 3, 4}
>>> B = {2, 4}
>>> A.intersection(B)
{2, 4}
```

While useful and very fast, this solution hides all the interesting parts under the hood. Let's first look at a piece of code that only uses the most basic programming concepts and is also very intuitive. Unfortunately, it is also very inefficient.

```
def list_intersection(listA, listB):
    """Build a list that only contains elements of both lists.

    This algorithm is highly ineffecient
    because it loops over the second list multiple times!

    Parameters
    ----------
    listA : list
        first list of elements
    listB : list
        second list of elements

    Returns
    -------
    list

    Examples
    --------
    >>> list_intersection([3, 1, 2], [4, 7, 2, 1])
    [1, 2]

    >>> list_intersection([3, 1, 2], [])
    []

    >>> list_intersection([3, 1, 2], [1, 2, 3])
    [3, 1, 2]
    """
    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is any b of listB the same as a?
        for b in listB:
            if a == b:
                # found a, add it to intersection
                intersection.append(a)

    # all loops done, return intersection
    return intersection
```

Here we create an empty list and only add an item to it if said item occurs in both lists. Doing this requires two `for`-loops. Notice how we do a full iteration over the second list for each element in the first list. If the first list contains m items and the second one n, we perform a total of $m + (m * n)$ lookups. So the number of lookups grows much faster than the combined number of elements in the lists.

The obvious problem is that we look at the second list over and over again, rather than just processing it once and memorizing what elements occur in it. In Python, such memorizing takes the form of converting a list to a dictionary (also known as a *hash map* or *hash table* in other programming languages). A dictionary uses keys as an addressing system for quickly looking up items.

```
>>> A = {'some_key': 'the_value', 'key_2': [5, 3],
         10: 'integers can be keys, too'}
>>> A.get('some_key')
'the_value'
>>> A.get('key_2')
[5, 3]
>>> A.get(10)
'integers can be keys, too'
```

Instead of searching through the second list over and over again, we first convert it to a dictionary and then use that for quick lookup.

```python
def list_intersection(listA, listB):
    """Build a list that only contains elements of all lists.

    This algorithm uses dictionaries for speed,
    but requires more memory.
    """
    # convert second list to dictionary
    dictB = {item: item for item in listB}

    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is a in listB?
        if dictB.get(a):
            # found a, add it to intersection
            intersection.append(a)

    # return intersection
    return intersection
```

With larger lists, this implementation will be much faster. And it is essentially what the very first solution with sets is doing, because Python's `set` data type is just a special case of Python dictionaries. For real-world applications, seemingly minor differences like this can have major repercussions. But if the lists are always very small, memorizing them with dictionaries might not be worth it. There are no simple answers here, it all depends on the specific use case.

You already saw in the case of linear search versus binary search how implementation details can obscure the bigger picture. But here the issue wasn't even choosing between largely different algorithms, but just how exactly one wants to implement a (very simple) operation. The two pieces of code we came up with are almost exactly the same, yet they differ greatly in their runtime behavior.

[Exercise 1.13 Consider the first version of the list intersection function, which does not use dictionaries. Instead of the second `for`-loop, we could have used Python's built-in membership test for lists:

```python
if a in listB:
    intersection.append(a)
```

Would this have solved the speed issues? Justify your answer. In order to do so, you might have to do some research on how exactly Python's `in`-operator works.]

[Exercise 1.14* The problem with multiple `for`-loops gets even worse when computing the intersection of multiple lists. For these cases, then, it's even more important to have an efficient solution.

Generalize the dictionary-based intersection function so that it can take an arbitrary number of lists as arguments. All lists except the first one should be converted to a dictionary.]

Part I

Phonology

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 ($/\mathrm{b}/$, $/\mathrm{d}/$, $/\mathrm{g}/$, $/\mathrm{z}/$, $\dots$) 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 **??**, 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 8×3 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.

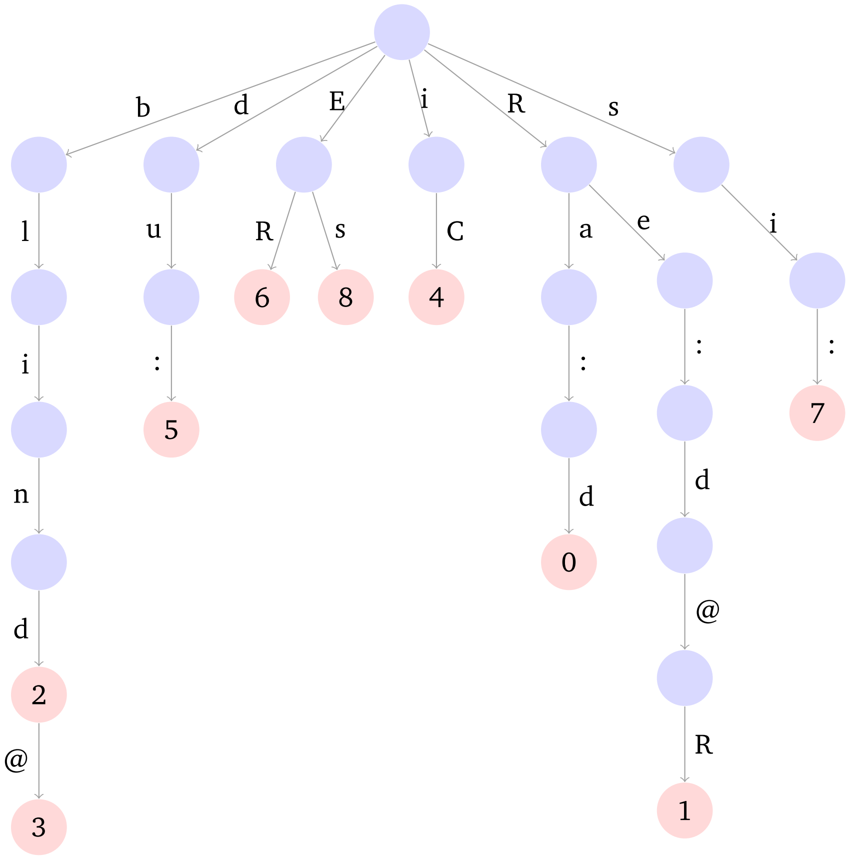
2.3 Prefix Trees

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.

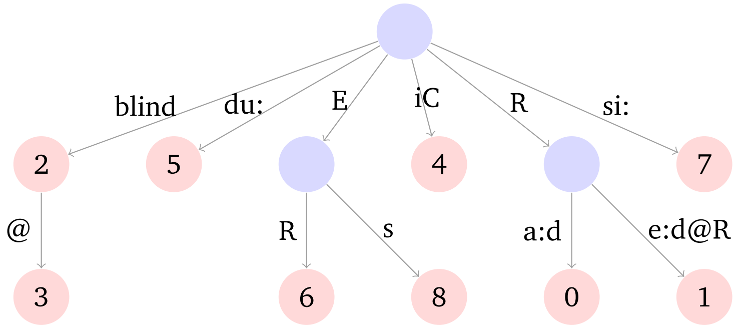
Considering that our phonology list consists of fully inflected output forms and that inflectional morphology in many languages involves a fair share of suffixation, it is very likely that the keys in the phonology list of a given language show a great amount of overlap. This allows us to represent them 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 (≈ 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

Dasgupta, Sajoy, Christos Papadimitriou & Umesh Vazirani. 2006. *Algorithms*. New York, NY: McGraw Hill.

Unit 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.

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.

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

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

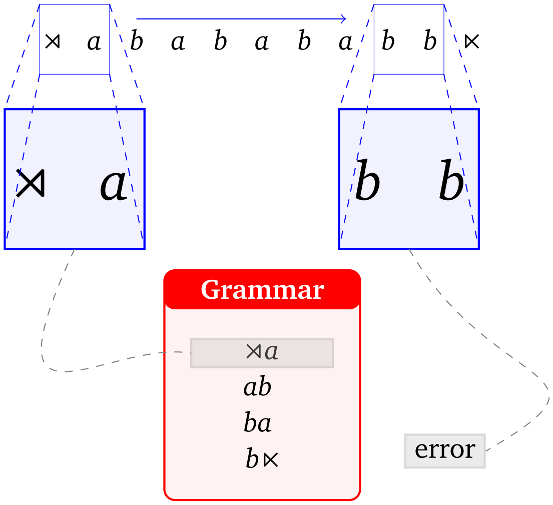


Figure 3.1: Scanner for a bigram grammar

point. An alternative is to first move the scan window through the entire string while 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.

```python
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
```

```python
>>> 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 & 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, $\text{2-grams}(w) := \{ab \mid \exists u, v \in \Sigma^* \text{ s.t. } u \cdot ab \cdot v = \hat{w}\}$ 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 $L(G) := \{w \mid \text{2-grams}(w) \subseteq G\}$. If G is a *negative bigram grammar* (denoted ${}^{-}G$), then it generates the language $L(G) := \{w \mid \text{2-grams}(w) \cap G = \emptyset\}$.

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.

$$\begin{tikzcd}
w \in L({}^{+}G) \arrow[d, leftrightarrow] & w \in L({}^{-}\overline{G}) \arrow[d, leftrightarrow] \\
\text{2-grams}(w) \subseteq G \arrow[r, leftrightarrow] & \text{2-grams}(w) \cap \overline{G} = \emptyset
\end{tikzcd}$$

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. $\lrcorner$

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 ⋊foʊn⋉⋊bɪl⋉. 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.

Relevant literature for Unit 3

add more info

Keenan, Edward & Larry Moss. 2012. Mathematical structures in language. Ms., UCLA and Indiana University.

Unit 4

Extending Bigram Grammars

Bigram grammars are a step in the right direction from the initial list phonology model. Every grammar is still just a list of items, but now we use this list to determine the well-formedness of a word in a compositional manner. The problem of bigram grammars is that they are not powerful enough to capture certain phonological processes like intervocalic voicing — in linguistic parlance, the size of the grammars' locality domain is limited to two adjacent symbols, so bigger contexts cannot be represented correctly. The obvious solution is to extend the size of the scanner window so that the grammar uses trigrams, 4-grams, or maybe even something bigger. We will see today that this generalization is a natural one in the sense that it preserves the essential properties of bigram grammars while increasing their empirical coverage.

1 Generalization to Strictly Local Grammars

Lifting the notion of bigrams to arbitrary k-grams works almost exactly as one would suspect. The set of trigrams of the string *abcd*, for example, includes $\rtimes ab$, *abc*, *bcd*, and $cd\ltimes$. However, it also includes $\rtimes \rtimes a$ and $d \ltimes \ltimes$. Why the extra trigrams with multiple edge markers? For one thing, we still want to directly express basic facts like "a word may start with *a*", while the trigram $\rtimes ab$ encodes "a word may start with *a* followed by *b*". More importantly, as the length of k-grams increase, more and more words won't have a single k-gram unless we pad them with $k-1$ edge markers. For example, the augmented version of the empty string would just be $\rtimes\ltimes$, which is too short for a trigram. Similarly, the string *ab* would be $\rtimes ab\ltimes$, which is too short any k-gram with $k > 4$. The simplest solution is to simply add $k-1$ edge markers instead of just a single one. For bigrams, the number of edge markers does not change since $2-1=1$.

On the formal side, the generalization from bigrams to k-grams requires but a few minor modifications in our original definition. We also use this opportunity to slightly change our terminology: instead of *k-gram grammar* and *k-gram languages*, we will speak of *strictly k-local grammars* and *strictly k-local languages*. This will make it easier later on to distinguish this formalism from a related one that also operates with k-grams but interprets them differently.

Definition 4.1 (k-grams). Let k be some natural number. A *k-gram*, or *k-factor*, over alphabet Σ is an element of $(\Sigma \cup \{\rtimes, \ltimes\})^k$. Given a string w over Σ, its *k-augmented* counterpart $\hat{w}_k := \rtimes^{k-1} \cdot w \cdot \ltimes^{k-1}$ consists of w with $k-1$ left edge markers and $k-1$ right edge markers, and its set of k-grams is given by $k\text{-grams}(w) := \{s \in (\Sigma \cup \{\rtimes, \ltimes\})^k \mid \exists u, v \in \Sigma^* \text{ s.t. } u \cdot s \cdot v = \hat{w}\}$.

You can see that this definition of k-gram is a natural extension of the concept of bigrams for if we replace k by 2 in the definition, we get exactly the original definition of bigrams. The concept of bigram languages is generalized in the same fashion to k-gram languages, which jointly form the class of *strictly local languages*.

Notice that the definition of strictly k-local also allows for $k = 0$. Can you list all strictly 0-local languages? *Hint*: There's only two of them.

Definition 4.2 (Strictly Local Languages). A finite set of k-grams is called a *strictly k-local grammar*. A positive strictly k-local grammar G generates the language $L(G) := \{w \mid k\text{-grams}(w) \subseteq G\}$. A negative strictly k-local grammar G generates the language $L(G) := \{w \mid k\text{-grams}(w) \cap G = \emptyset\}$. A language L is *strictly k-local* iff it is generated by some strictly k-local grammar. The class of *strictly local languages* is given by $\bigcup_{k \geq 1} \{L \mid L \text{ is strictly } k\text{-local}\}$.

Intuitively, a language is strictly local iff all its well-formedness conditions are restricted to a locality domain of finitely bounded size. This also means that every strictly local language can be recognized by a scanner that can adapt the size of its search window to any finite size depending on the grammar the scanner operates with. Figure 4.1 shows such a scanner working with a positive strictly 4-local grammar.

At first glance it seems that the grammars in Fig. 3.1 and 4.1 generate the same language, but that is not the case. What is the difference between the respective languages?

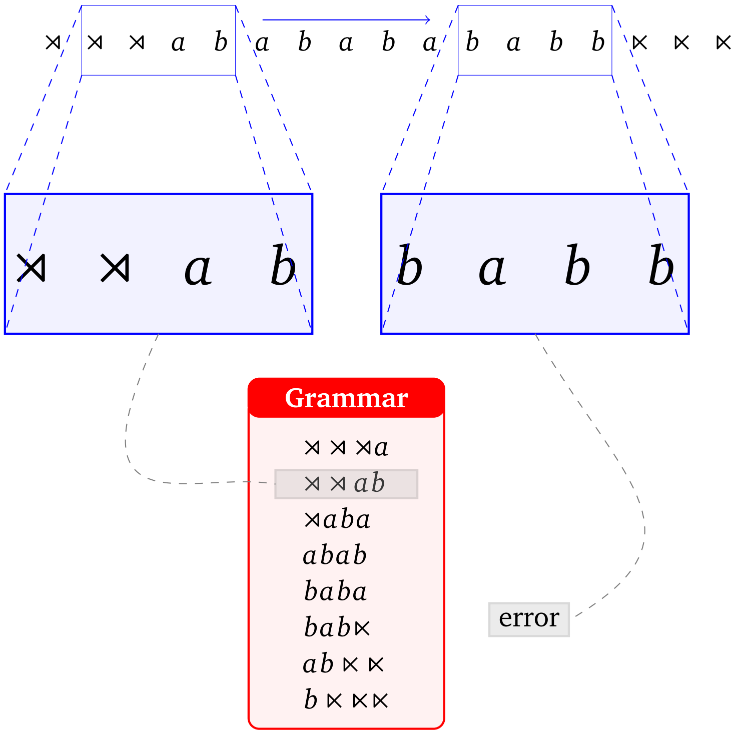


Figure 4.1: Scanner for a positive strictly 4-local grammar

Example 4.1 A Strictly 3-Local Grammar for $(aba)^+$

Consider the language $(aba)^+$, which consists of the strings *aba*, *abaaba*, *abaabaaba*, and so on. This language is not strictly 2-local, because any positive strictly 2-local grammar that generates the string *abaaba* must contain the bigram *aa* and thus would also generate any string of the form $abaa^+ba$. But there is a positive strictly 3-local grammar that generates the language:

$$\begin{array}{lll} \rtimes\rtimes a & aba & ba\ltimes \\ \rtimes ab & baa & a\ltimes\ltimes \\ & aab & \end{array}$$

Substituting C for a and V for b, we can conclude from this example that natural languages with a CVC syllable template require a locality domain of at least 3.

The phonological processes we have looked at so far — word-final devoicing, nasal assimilation, and intervocalic devoicing — are all strictly local. The first two are strictly 2-local, the third one strictly 3-local.

The example only shows that no *positive* strictly 2-local grammar generates $(aba)^+$, but this implies that no negative strictly 2-local grammar can generate it either. If this is not immediately apparent to you, reread chapter 3.

Example 4.2 Intervocalic Voicing is Strictly 3-Local

It is easy to see that intervocalic voicing cannot be captured with strictly 2-local grammars. Suppose the language under investigation contains the words bɪsta and bɪtsa. Any positive strictly 2-local grammar generating these two strings contains at least the bigrams ⋊b, bɪ, ɪs, sa, and a⋉. Consequently, whatever language it generates must contain the string bɪsa, which violates intervocalic voicing.

A negative strictly 3-local grammar, on the other hand, can easily enforce intervocalic voicing. It only needs to contain all trigrams of the form UsV, where U and V are vowels and s is some voiceless consonant.

It is also readily apparent that constraints on syllable structure are strictly local. Take some language that only allows for syllables of the form V, VC, CV, and CVC, but not CCV, VCC, CVCC, or CCVCC. In other words, a well-formed word cannot contain more than two consecutive consonants, and these consonants cannot occur at the beginning or the end of the word, where they would necessarily be part of the same syllable. The set of illicit substrings, then, consists of ⋊CC, CCC, and CC ⋉. This can be compiled out into a set of illicit sequences of phones by substituting for each C the phones that are specified for [+cons]. So this example of a restricted syllable template does note exceed the power of a negative strictly 3-local grammar.

If desired, the negative strictly local grammar can be converted into a positive one using our standard procedure. Go back to the proof of our theorem that positive and negative strictly 2-local grammars are equivalent, and you will see that it does not rely on the length of the k-grams being 2. Therefore the theorem can be trivially generalized to all strictly k-local grammars ($k \geq 1$).

Why does equivalence of positive and negative grammars break down for strictly 0-local grammars?

We could spend much more time designing strictly local grammars for other local processes in phonology, e.g. assimilation across word boundaries, vowel harmony, umlaut, or spirantization. As long as these processes describe surface true generalizations — that is to say, they do not make reference to an underlying form and can be stated purely in terms of which output forms are licit — writing down the grammars might require quite a bit of ink, but very little genuine thought. Notice that these processes must be local but can nonetheless be globally unbounded. Vowel harmony, for instance, may apply throughout an entire word via a sequence of local vowel harmony steps.

Example 4.3 Strictly 4-Local Vowel Harmony

Suppose language L has an alphabet with two vowels, i and u, and two consonants p and l. All words match the syllable template (CVC)$^+$. In addition, i triggers progressive vowel harmony that turns each u following an i into i. Consequently, no surface form may contain an i followed by a u. Assuming that no other processes or constraints apply in L, the language is strictly 4-local.

We first construct a positive strictly 4-local grammar ^+S for the syllable template. This grammar contains all 4-grams, and only those, that can be obtained from the entries below by substituting i and u for occurrences of V and p and l for occurrences of C (which yield a total of 76 4-grams).

$$\begin{array}{lll}
\rtimes\rtimes\rtimes\mathrm{C} & \mathrm{C\,V\,C\,C} & \mathrm{C\,V\,C}\ltimes \\
\rtimes\rtimes\mathrm{C\,V} & \mathrm{V\,C\,C\,V} & \mathrm{V\,C}\ltimes\ltimes \\
\rtimes\mathrm{C\,V\,C} & \mathrm{C\,C\,V\,C} & \mathrm{C}\ltimes\ltimes\ltimes
\end{array}$$

Vowel harmony is enforced by the negative strictly 4-local grammar ^-V with the 4-grams *ilpu* and *iplu*. We then convert ^+S into an equivalent negative grammar and take their union. A brief moment of reflection reveals that $L(^-\overline{S} \cup ^- V) = L$.

Give a rough approximation of the size of $^-\overline{S}$ and compare it to ^-V. Does the size difference match your intuition about the complexity of the respective phonotactic constraints encoded by these grammars?

Locally bounded processes giving rise to locally unbounded dependencies is exactly what we expect given how strictly local grammars operate: the k-grams only regulate the shape of local domains, but the well-formedness of the word is evaluated by moving through the word and checking each local domain. So not only are strictly local grammars powerful enough to capture local well-formedness conditions in phonology, the way they enforce them mirrors linguists' intuitions about how local processes can produce global patterns.

The connection between strictly local grammars and local processes in phonology allows us to study the latter through the former. Since strictly local grammars generate strictly local languages, this implies that the properties of strictly local languages can tell us something about local processes in phonology. Let us repeat this for emphasis:

The properties of our formal objects
are properties of (specific aspects of) language.

The theoretical task of proving formal properties of strictly local languages has suddenly morphed into an empirically minded investigation of phonology.

2 Exploring Strictly Local Languages

2.1 A Proper Hierarchy of Strictly Local Languages

The examples discussed so far suggest that the power of strictly local grammars increases with the size of the locality domain. This is indeed the case, but is best proved by recourse to the strictly local languages. If we order the strictly local languages by the size of their locality domain, we get a proper hierarchy: one level properly subsumes the next. Denoting the class of strictly k-local languages by SL_k, we have $\mathrm{SL}_k \subsetneq \mathrm{SL}_{k-1}$ for all $k \geq 0$.

This is a complex claim, as it asserts that each strictly k-local language is strictly $k+1$-local, but that the opposite does not hold for some languages. Following our standard strategy, we establish lemmata for these weaker claims and then combine them into the original theorem.

Lemma 4.3. It holds for every $k \geq 0$ that if language L is strictly k-local, then L is also $k+1$-local. ⌟

The proof for this lemma is slightly more complicated than anything we have seen so far, but behind the notation lies a very simple idea: the size of the locality domain can be increased from k to $k+1$ by padding the edge markers and by combining two overlapping k-grams into a single $k+1$-gram. Consequently, for every strictly k-local grammar there is an equivalent strictly $k+1$-local one.

Proof. If L is strictly k-local then it is generated by some positive strictly k-local grammar G over some alphabet Σ. Let G' be the smallest set such that for all k-grams $g_1, g_2 \in G$

- if g_1 starts with $\rtimes$, then $\rtimes g_1 \in G'$,
- if g_1 ends with $\ltimes$, then $g_1 \ltimes \in G'$,
- if $g_1 := a_1 a_2 \cdots a_k$ and $g_2 := a_2 \cdots a_k a_{k+1}$, then $a_1 a_2 \cdots a_k a_{k+1} \in G'$.

Clearly G' is finite and can therefore be interpreted as a positive $k+1$-local grammar. We show that $L = L(G')$, thus establishing that L is $k+1$-local.

If $w \in L(G')$, then $k+1$-grams(w) is a subset of G'. All $k+1$-grams with multiple edge markers have a corresponding k-gram with one edge marker less, which is contained in G. All other $k+1$-grams are split into two k-grams by removing the first or the last symbol. Each k-gram is once again contained in G, so k-grams(w) $\subseteq G$ and hence $w \in L(G) = L$. Since w was arbitrary we have $L(G') \subseteq L$.

If $w \in L$, then k-grams(w) is a subset of G. Assume towards a contradiction that $k+1$-grams(w) is not a subset of G'. Then w contains some $k+1$-gram g_3 that is not a member of G. If g_3 starts or ends with two edge markers, then the corresponding k-gram with only one of the two markers cannot have been part of G, contradicting our initial assumption. In all other cases, g_3 is built from two overlapping k-grams g_1 and g_2, at least one of which is not contained in G. But then k-grams(w) is not a subset of G, contradicting once more our initial assumption. It follows, then, that $k+1$-grams(w) is a subset of G' after all, wherefore $L \subseteq L(G')$. □

Example 4.4 Converting Bigrams Into Trigrams

Consider the strictly 2-local language $(ab)^+$ and its positive strictly 2-local grammar G with the bigrams $\rtimes a$, ab, ba, and $b\ltimes$. In order to construct an equivalent strictly 3-local grammar, we have to pad out the edge markers and combine overlapping bigrams. So $\rtimes a$ and $b\ltimes$ become $\rtimes \rtimes a$ and $\ltimes \ltimes b$, respectively. We also see that $\rtimes a$ overlaps with ab, so we can combine them into $\rtimes ab$. The same procedure produces

$ab\ltimes$ from ab and $b\ltimes$. Finally, ab and ba overlap in two ways depending on which one is put in front of the other, so that we obtain two trigrams from them: aba and bab. This exhausts the number of possible combinations. The strictly 3-local grammar is shown below:

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

This grammar generates all strings of $(ab)^+$, and only those.

Lemma 4.4. For every k there is some strictly $k+1$-local language that is not strictly k-local.

This proof uses a finite language, but the lemma holds even if we only consider infinite languages. Try to generalize the proof along these lines.

Proof. Consider the finite language L that contains only the string a^k, i.e. the string with k consecutive as. It is generated by the strictly $k+1$-local grammar $\left\{\rtimes a^k, a^k\ltimes\right\}$. However, k-grams$(a^k) = \left\{\rtimes a^{k-1}, a^k, a^{k-1}\ltimes\right\} = k$-grams$(a^n)$ for every $n \geq k$, so a strictly k-local grammar that generates a^k also generates all these a^n and thus a proper superset of L. □

Theorem 4.5. For all $k \geq 0$, $\mathrm{SL}_k \subsetneq \mathrm{SL}_{k+1}$. ⌟

Now we know for sure that the size of the locality domain has a direct effect on generative capacity. This is hardly surprising, but still far from trivial — in Chapter 9 we will encounter a very similar formalism for which all $k \geq 2$ have exactly the same power.

2.2 Relation to Finite Languages

Remember that the list phonology model of Lecture 2 was restricted to lists of finite length, so it could only generate finite languages. This restriction is empirically inadequate as it conflicts with the assumption that the list of phonological words in a given natural language is infinite and fails to handle nonce words and linguistic creativity in general. But not only is the list phonology model restricted to finite languages, it adds insult to injury with its ability to generate all finite languages. Every finite language is a viable natural language phonology according to the list model, and we have already seen why this is typologically untenable. The class of finite languages is the class that is least likely to provide an insightful or empirically adequate model of language, so we should strive to work with formalisms that cannot generate this class.

The strictly local languages are a marked improvement over the list phonology model in this respect, but only if one adopts the right perspective. First, it is obvious that strictly local languages can be infinite, so not every strictly local language is finite. Let us make this claim fully explicit via a proof. We already know that every strictly k-local language is strictly $k+1$-local, so all we need is an example of a language that is both infinite and strictly 1-local language.

Further reflection reveals that almost every 1-local language is infinite. In fact, there are only two 1-local languages that are finite. Can you define them? *Hint*: One of them is strictly 0-local.

Lemma 4.6. There is a strictly 1-local language that is infinite. ⌟

Proof. Let ${}^{+}G := \{\rtimes, a, \ltimes\}$. Then $L({}^{+}G) = \{\varepsilon, a, aa, aaa, \ldots\} = a^*$, which is infinite. □

Corollary 4.7. For every $k \geq 1$, SL_k contains an infinite language. $\lrcorner$

This is a welcome result because it shows that no matter what size of locality domain we pick, we are never restricted to just finite languages. An even more appealing property of strictly local languages is that for every level of the infinite hierarchy there are some finite languages that cannot be defined. Consequently, strictly local grammars improve on the list phonology model in that they cannot define just about any arbitrary phonological systems.

Theorem 4.8. For every $k \geq 1$, there is some finite language that is not contained in SL_k. $\lrcorner$

Proof. Pick an infinite language that is generated by some strictly k-local grammar G over alphabet Σ. Discard from G all k-grams that start with $\rtimes$ or end in $\ltimes$. The resulting set is a finite language L. We show that L is not strictly k-local.

Since G generates an infinite set, it must contain (not necessarily distinct) k-grams $a \cdot u$ and $u \cdot b$, where $a, b \in \Sigma$ and $u \in \Sigma^{k-1}$. Note that both $a \cdot u$ and $u \cdot b$ belong to L, whereas $a \cdot u \cdot b$ does not. But k-grams$(a \cdot u) \cup k$-grams$(u \cdot b) = k$-grams$(a \cdot u \cdot b)$, and consequently every strictly k-local grammar that generates $a \cdot u$ and $u \cdot b$ also generates $a \cdot u \cdot b$. Hence L is not strictly k-local. $\square$

The proof above is rather sneaky. It exploits the fact that every strictly k-local grammar is a finite set of strings of length k, but this very set can also be viewed as a finite language. This is an interesting perspective: a strictly k-local grammar G is a finite language L_G coupled with a specific algorithm A for creating a new language from L_G. If we try to generate L_G via a second strictly k-local grammar, this grammar automatically uses the algorithm A, so for certain choices of L_G the grammar generates additional strings via A that do not belong to L_G.

Viewing grammars as languages will seem odd to many linguists, who think of the grammar as the knowledge the speaker has acquired about their language, whereas the language is the output produced from this knowledge. But that is an ontological distinction. It is a useful distinction to make when outlining the research program of linguistics, its goals, problems, and promises. It highlights that modern linguistics is about human cognition rather than merely describing individual languages as abstract systems of structures and rules that exist independent of the speaker's cognitive ability to use them. The latter view dates back to Structuralism, and there are good reasons why it was abandoned. As useful as the ontological mandate may be, though, it has no say over what mathematical tricks we may avail ourselves of. Equating grammars with languages can be useful in some cases. Here it furnishes a proof about the limits of strictly local languages, and in Ch. 17 it will open up completely new methods of constructing syntactic grammars.

If you still find all of this horribly confusing, do not despair: there is a much simpler proof that does not blur the distinction between grammars and languages and instead uses the same trick that we already encountered in the proof of Lem. 4.4.

Proof. Consider the finite language that consists only of the string a^k. Note that k-grams$(a^k) = \left\{\rtimes a^{k-1}, a^k, a^{k-1}\ltimes\right\} = k$-grams$(a^{k+1})$. Thus every strictly k-local grammar that generates a^k also generates a^{k+1}. This entails that no strictly k-local grammar generates the finite language $\left\{a^k\right\}$. $\square$

Why would anyone ever want to use the more complicated proof if there is a much simpler one? In poetic terms: because the journey is the destination. Proofs aren't just about establishing results, they tell us why a specific result holds and hence provide us with deeper insights. The simpler proof does not highlight that each strictly local grammar it itself a finite language, an obvious yet nonetheless surprising fact. New perspective like this are always useful. Remember the Keenan-Moss credo:

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

This isn't restricted to definitions, it also extends to proofs. The more routes take us towards a specific result, the better.

One important point that both proofs share is that they presuppose that the size of the locality domain is fixed to some k. Hence they do not apply if we consider the whole class of strictly local languages, which turns out to properly subsume the class of finite languages.

Theorem 4.9. Every finite language is strictly local. ⌟

Proof. Suppose that L is a finite language, the longest string of which has length $k-1$, $k \geq 1$. We define a k-local grammar G that consists of the k-grams $\rtimes^i \cdot w \cdot \ltimes^j$, where $w \in L$, w has length $l < k$, and $i + j + l = k$. Since every k-gram starts with $\rtimes$ or ends in $\ltimes$, $L(G) = L$, wherefore L is strictly k-local. □

We see that the relation between strictly local languages and finite languages is more involved than one would expect. Without restrictions on the size of the locality domain, the strictly local languages include all finite languages. However, the class of strictly k-local languages and the class of finite languages are incomparable — they have a non-empty intersection, but neither subsumes the other (cf. Fig. 4.2). So assuming that k is fixed for natural language phonology, e.g. as part of Universal Grammar, strictly local grammars are a good approximation of local processes in phonology. They can handle infinity and do not incorrectly predict languages to vary freely across all dimensions. In particular, they naturally give rise to iterated local processes such as progressive vowel harmony.

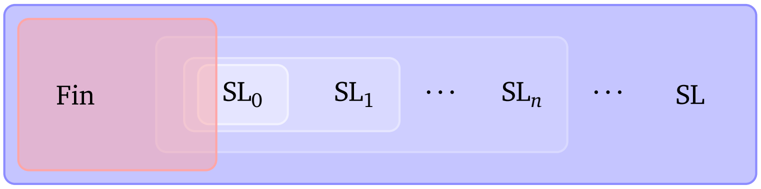


Figure 4.2: Relation between strictly local and finite languages

2.3 Substring Substitution Closure

In several of the preceding proofs we have used the fact that a strictly local grammar sometimes "overshoots the target". A finite language L may not be in SL_k because a strictly k-local grammar that tries to generate L will also end up generating other strings outside of L. But a strictly $k+1$-local grammar may be able to do a point landing and generate all and only those strings that are members of L. What this shows is that a strictly k-local grammar only has perfect precision within its locality domain of size k, beyond that it has to generalize. This generalization step is what allows strictly local grammars to generate infinite languages, making it the true source of their power.

Crucially, though, strictly local grammars don't just generalize randomly, quite to the contrary: all strictly local grammars generalize in the same fashion, irrespective of the size of their locality domain. Their generalization strategy is already implicit in the definition of strictly local languages, which categorizes strings as well-formed or ill-formed according to their set of k-grams. If two strings have exactly the same set of k-grams, then either both are well-formed or both are ill-formed.

But it is quite hard to tell what this condition implies for the overall shape of the string languages. Given a string language L, how can we tell whether L is strictly local? Fortunately strictly local languages are uniquely characterized by a property called *substring substitution closure*.

Definition 4.10 (Local Substring Substitution Closure). A language L satisfies *k-local substring substitution closure* iff there is some $k \geq 1$ such that if L contains both $u \cdot x \cdot v$ and $u' \cdot x \cdot v'$, where x has length $k-1$, then L also contains $u \cdot x \cdot v'$.

Theorem 4.11. A language is in SL_k iff it satisfies k-substring substitution closure. $\lrcorner$

Let us look at a couple of examples first before wading through the proof of the theorem.

Example 4.5 A Substring Substitution Closed Language

We have already seen that the language $(ab)^+$ is strictly 2-local as it is generated by the grammar $\{\rtimes a, ab, ba, b\ltimes\}$. Now we can also verify this via substring substitution closure. For instance, we can line up *abab* and *abababab* to show that the language must also contain *ababab*.

$$\begin{array}{rcll} & x & & \\ ab & a & b & \in L \\ abab & a & bab & \in L \\ \hline ab & a & bab & \in L \end{array}$$

Notice how x is a single symbol since its length must be $k-1 = 2-1 = 1$. Also, we could have established the membership of *ababab* more succinctly using just *abab* or *abababab*.

$$\begin{array}{rcll} & x & & \\ ab & a & b & \in L \\ & a & bab & \in L \\ \hline ab & a & bab & \in L \end{array} \qquad \begin{array}{rcll} & x & & \\ ab & a & babab & \in L \\ abab & a & bab & \in L \\ \hline ab & a & bab & \in L \end{array}$$

Since substring substitution closure fails if even a single string is missing from the set, it is usually not a good way of showing that a language is strictly local — if the language in question is infinite, one cannot show via specific substitutions that it is suffix substitution closed. However, suffix substitution closure is an excellent way of showing that a language is **not** strictly local by giving a single example of a missing string.

Example 4.6 A Language that Fails Subtree Substitution Closure

Consider the language $(aa)^+$, the variant of $(ab)^+$ where all bs have been replaced by as. This language is not strictly local as it fails k-local substring substitution closure for any choice of k. Suppose k is an even number:

$$\begin{array}{cccc}
 & x & & \\
a & a \cdots a & a & \in L \\
 & a \cdots a & & \in L \\
\hline
a & a \cdots a & & \notin L
\end{array}$$

A minimally different pattern is used if k is odd. No matter what the value of k, the language is not suffix substitution closed and thus not strictly local. You might find this surprising given that the only difference to $(ab)^+$ is the replacement of b by a. This highlights another property of strictly local languages: the alphabet plays a crucial role in what languages are definable.

Proof. We now show that a language L is strictly k-local iff it is closed under k-local substring substitution.

Left to right Note first that substring substitution closure is trivially satisfied if L contains no strings of length strictly greater than $k-1$, for then all strings take the form $\varepsilon \cdot x \cdot \varepsilon$ with respect to k-local substring substitution. Suppose, then, that s_1 and s_2 are strings of L with length strictly greater than $k-1$ such that $s := u_1 \cdot x \cdot v_1$ and $s_2 := u_2 \cdot x \cdot v_2$. If $s_1 = s_2$, then $u_1 \cdot x \cdot v_2 = s_1 = s_2$, so substring substitution closure is not violated. If $s_1 \neq s_2$, then it must be the case that k-grams$(u_1 \cdot x \cdot v_2) \subseteq$ k-grams$(s_1) \cup k$-grams$(s_2) \subseteq G$, where G is a strictly k-local grammar with $L(G) = L$. It follows immediately that $s_1 \cdot x \cdot v_2$ is a member of $L(G)$ and thus a member of L. This exhausts all possible cases, showing that L is indeed closed under k-local substring substitution.

Right to left Suppose L is closed under k-local substring substitution, and let $G := \bigcup_{w \in L} k$-grams(w) be a positive strictly k-local grammar. It suffices to establish $L(G) = L$, which entails that L is strictly k-local.

It is easy to see from the definition of G that $L \subseteq L(G)$. Showing that $L(G) \subseteq L$ requires a fairly lengthy proof that is omitted here. The curious reader is referred to Rogers (2007: 19–21). □

2.4 Closure Properties

Suffix substitution closure is — as its name implies — a *closure property*. One says that an object o is closed under an operation iff applying this operation to elements of o yields only elements of o. In other words, the operation never takes us outside of o. The natural numbers, for instance, are closed under addition since the sum of two natural numbers is yet again a natural numbers. But they are not closed under subtraction because, say, $2-5$ yields -3, which is an integer but not a natural number. Suffix substitution closure simply means that a language is closed under the operation of substituting suffixes in a specific way.

But of course there are many other operations that can be applied to a language, and it will be interesting to see whether strictly local languages are closed under them. In particular the basic set-theoretic operations of intersection, union, and relative complement are of interest since they tell us how we can build strictly local languages from smaller ones.

Let us look at closure under intersection first. This one is particularly important because of the close correspondence that negative strictly local grammars establish between constraints on the one hand and languages on the other. If every k-gram corresponds to a well-formedness constraints, then one would expect that one can get the intersection of two languages by simply conjoining their respective well-formedness constraints. That is indeed the case.

Lemma 4.12. The class of strictly k-local languages is closed under intersection, $k \geq 0$. $\lrcorner$

Proof. We prove that for any two strictly local languages generated by (positive) grammars G_1 and G_2, $L(G_1) \cap L(G_2) = L(G_1 \cap G_2)$. We first show $L(G_1) \cap L(G_2) \subseteq L(G_1 \cap G_2)$. Let w be an arbitrary string belonging to both $L(G_1)$ and $L(G_2)$, i.e. $w \in L(G_1) \cap L(G_2)$. Then every k-gram of w is contained in both G_1 and G_2, so $w \in L(G_1 \cap G_2)$. Since w is arbitrary, we have $L(G_1) \cap L(G_2) \subseteq L(G_1 \cap G_2)$. The same reasoning can be applied in the other direction, yielding $L(G_1 \cap G_2) \subseteq L(G_1) \cap L(G_2)$. These two facts jointly imply $L(G_1) \cap L(G_2) = L(G_1 \cap G_2)$. □

Give an analogous proof using negative grammars.

Closure under intersection, too, can be used to prove that a language is not strictly local. Suppose we know that L is strictly local, but we have a hard time showing that L' is not strictly local. Then we can instead try to show that $L \cap L'$ is not strictly local, as this immediately implies the non-locality of L', too.

One might expect that closure under union holds too since one can simply take the union of the grammars, but this does not work as expected. The union of two grammars G_1 and G_2 often generates a superset of the union of $L(G_1)$ and $L(G_2)$. Do not even try to look for a smarter strategy to build a grammar for the union of two strictly local languages, there is none that works in all cases.

Lemma 4.13. The class of strictly k-local languages is not closed under union. $\lrcorner$

Proof. We give an example of two strictly 2-local languages whose union is not strictly 2-local. The proof can easily be adapted for arbitrary values of k.

Let $L_1 := \{ab\}$ and $L_2 := \{b, bb, bbb, \ldots\}$. Assume w.l.o.g. that all strictly 2-local grammars are positive. Then any strictly 2-local grammar that generates L_1 must contain the bigrams $\rtimes a$, ab, and $b\ltimes$. Similarly, a strictly 2-local grammar generating L_2 must contain the bigrams $\rtimes b$, bb, and $b\ltimes$. Therefore a strictly 2-local grammar that generates all strings in $L_1 \cup L_2$ must contain at least these bigrams. But such a grammar also generates the string abb, which is not part of $L_1 \cup L_2$. Hence there is no strictly 2-local grammar that generates all the strings in $L_1 \cup L_2$ and nothing else, so $L_1 \cup L_2$ is not a strictly 2-local language. □

"w.l.o.g." is short for "without loss of generality" and is meant to indicate that the proof can easily be adapted to cases that are not covered by the assumption.

The proof above only shows that the union of two strictly k-local languages is not guaranteed to be strictly k-local. However, it will still be strictly $k+1$-local. So it is still conceivable that the whole class of strictly local languages is closed under union, that is to say, the union of two strictly k-local languages may still be strictly m-local for some $m > k$. But this is not always the case, either. Intuitively, union can give rise to long-distance dependencies that cannot be enforced with strictly local grammars.

Lemma 4.14. The class of strictly local languages is not closed under union. ⌟

Proof. Consider the two strictly 2-local languages $L := ab^+$ and $L' := b^+a$. Their union is not closed under k-local substring substitution closure for any $k \in \mathbb{N}$ since $ab^k a \notin L \cup L'$, so it is not strictly local. □

Abstract as it may be, this result is of immediate importance for our investigation of phonology. If the union of two arbitrary strictly local languages were also strictly local, then we would expect that phonological processes can always apply disjunctively. Consider the following example: some languages have sibilant harmony, while others have vowel harmony. Each process can be equated with the set of strings that satisfy the respective harmony pattern. If we take the union of these two sets, we get the set of strings that satisfy sibilant harmony or vowel harmony. If that set were intersected with some natural language's phonology — which is an abstract way of saying that we force the constraint represented by that set onto the language — then words in that language are well-formed as long as they obey sibiliant harmony or vowel harmony, but they need not obey both. This is extremely unnatural. If a language has two phonological processes, then words must usually obey both. It is not the case that satisfying one constraint grants you *carte blanche* to ignore the other. The closure properties of strictly local languages can explain this fact: intersection (= constraint conjunction) always yields a potential natural language phonology, union (=constraint disjunction) may not. The assumption that local phonological processes fit within the bounds of strictly local languages thus makes a typological prediction: if two constraints apply in a disjunctive fashion as indicated above, then these two constraints must be such that their union is also strictly local — which rules out a great number of strictly local languages.

The previous two lemmata also imply that closure under relative complement does not hold.

Lemma 4.15. The class of strictly local languages is not closed under (relative) complement. ⌟

Proof. By De Morgan's law $L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$. If the class of strictly local languages were closed under both complement and intersection, it would thus be closed under union, too. Since closure under intersection holds while closure under union does not, closure under relative complement cannot hold, either. □

You may find this result surprising because the complement of a strictly local grammar is a strictly local grammar. But in general $\overline{L(G)}$ differs from $L(\overline{G})$, just like $L(G_1) \cup L(G_2)$ is not guaranteed to be $L(G_1 \cup G_2)$. These differences illustrate why it is so important to distinguish between grammars and the languages they generate.

Another property not enjoyed by strictly local languages is closure under concatenation. Given two languages L_1 and L_2, their concatenation contains all strings that can be obtained by concatenating a string of L_1 with a string of L_2. In mathematical jargon: $L_1 \cdot L_2 := \{u \cdot v \mid u \in L_1, v \in L_2\}$.

Lemma 4.16. The class of strictly local languages is not closed under concatenation. ⌟

Proof. Consider the strictly 2-local languages $L_1 := (a^+c)^+$ and $L_2 := (a^+d)^+$. Their concatenation is not closed under k-local substring substitution closure:

$$\begin{array}{llll} a^{k-1}cad & a^{k-1} & d & \in L_1 \cdot L_2 \\ & a^{k-1} & cada^{k-1}d & \in L_1 \cdot L_2 \\ \hline a^{k-1}cad & a^{k-1} & cada^{k-1}d & \notin L_1 \cdot L_2 \end{array} \qquad \square$$

Closure under string concatenation is yet another unnatural closure property since it implies that the rules of a grammar can switch completely at some point in the string. The fact that English and French are natural languages would make the ludicrous prediction that a system where the first part of the word follows the rules of English phonology and the second part the French rules is a possible natural language phonology. Lack of closure under string concatenation thus is a very robust universal of the class of natural languages and one that we do not want in our formalism, either.

We finish with yet another missing closure property. A relabeling of a language is a function that replaces each symbol in all strings by some other symbol. More precisely, let Σ and Ω be two alphabets and $r : \Sigma \to \Omega$ be a total function (that is to say, it produces some output for every symbol of Σ). Then for every string $w = w_1 w_2 \cdots w_n$, we let $r(w) := r(w_1)r(w_2)\cdots r(w_n)$ and $r(L) := \{r(w) \mid w \in L\}$.

Lemma 4.17. The class of strictly local languages is not closed under relabelings. ⌟

Proof. We have already seen that $(ab)^+$ is strictly 2-local whereas the relabeling $(aa)^+$ is not strictly k-local for any $k \in \mathbb{N}$. But the former is the image of the latter under a relabeling that replaces all bs by as. □

This is a very important result as it ties directly into the abstractness debate in generative phonology. Suppose that we are allowed to have all kinds of hidden structure in our phonological representation, e.g. a basic syllable template or feet. Then we could assume that the language $(aa)^+$ is underlyingly equipped with a CV-template, so that we should rather think of it as $(a_C a_V)^+$. This language is strictly local (it is a notational variant of $(ab)^+$), so $(aa)^+$ is strictly local under a mapping that removes unpronounced structure. In Chapter 9 we will see that this is a very dangerous route to take: hidden structure pushes strictly local languages to a level of power where all phonological processes are equally simple. This eradicates all distinctions between processes and takes us back to the undesirable egalitarianism of the list phonology model.

A summary of all the (non-)closure properties is given in Tab. 4.1. Notice that the closure properties of strictly local languages is closely in line with what we see natural languages, whereas the finite languages get almost everything wrong.

3 Implications for Phonology

Our mathematical expedition has taught us a surprising amount about phonology. First of all, all local phonological dependencies (including those spanning across word boundaries) can be handled by strictly local grammars, a very simple formalism with few cognitive requirements. The model comes with a minimal memory burden as it only requires the speaker to keep track of a small number of k-factors. It is also very fast: recognition via a scanner takes linear time and can be carried out in an incremental

Closure under...	**FIN**	**SL**	**Natural**	**Typological Issue**
Intersection	✓	✓	✓(?)	all constraints can apply conjunctively
Union	✓	✗	✗	all constraints can apply disjunctively
Complement	✓	✗	✗	constraint opposites are possible constraints
Concatenation	✓	✗	✗	distinct constraints for word halves
Relabeling	✓	✗	✗(?)	abstractness debate

Table 4.1: Comparison of closure properties and their linguistic relevance

online fashion. Lookup of k-factors in the grammar takes at most logarithmic time using binary search, and even faster search methods can be implemented. With a hash table or prefix tree, lookup is practically instantaneous.

But strictly local grammars also capture essential properties of phonological competence. They are analytic in nature and thus capable of generating infinite languages, which solves the problems the list phonology model had with linguistic creativity and nonce words. The special relation between strictly k-local languages and finite languages also means that certain typological pitfalls are avoided. Not every random collection of strings is predicted to be a valid phonological system, and grammars generating an infinite language generalize in a linguistically plausible fashion by determining the well-formedness of strings as a composite function of the local domains.

We have also seen that strictly local languages lack closure properties that do not hold of natural languages either. The union or relative complement of a natural language's phonotactics is not guaranteed to be a valid phonological system for a natural language, just like the class of strictly local languages is not closed under these operations. The increase of power brought about by relabelings also cautions us against using hidden structures and highly abstracted alphabets, an issue that has been discussed at length in phonology and that will occupy us at various points for the rest of the course.

Relevant literature for Unit 4

add more info

Rogers, James. 2007. *Formal Description of Syntax*. Lecture Notes. http://www.cs.earlham.edu/~jrogers/files/reader.pdf.

Unit 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. **??** 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

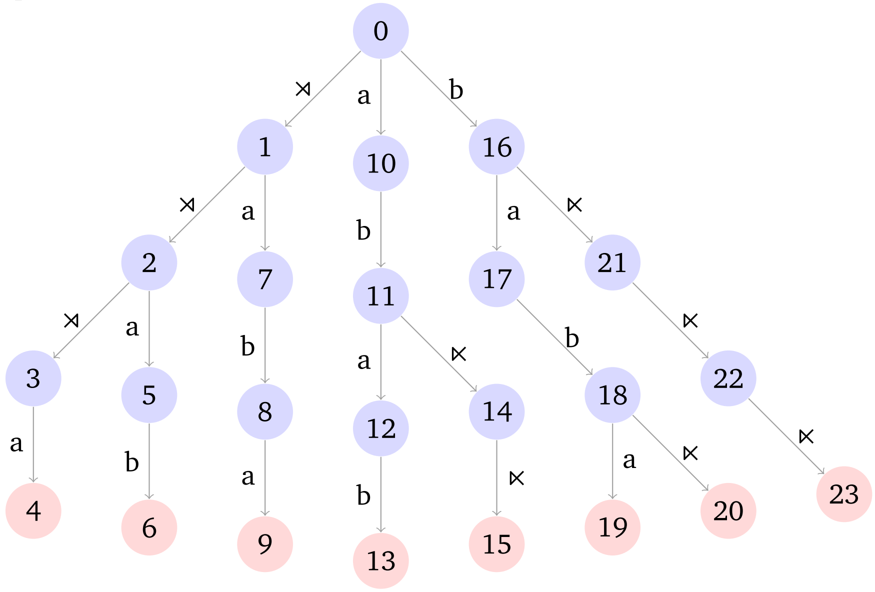
The discussion of the list phonology model in Ch. 2 spent quite some time on how a list of surface forms can be efficiently searched and stored. Prefix trees turned out to be ideal for this purpose as they provide a more compact representation but can also be searched very quickly — finding an item only 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

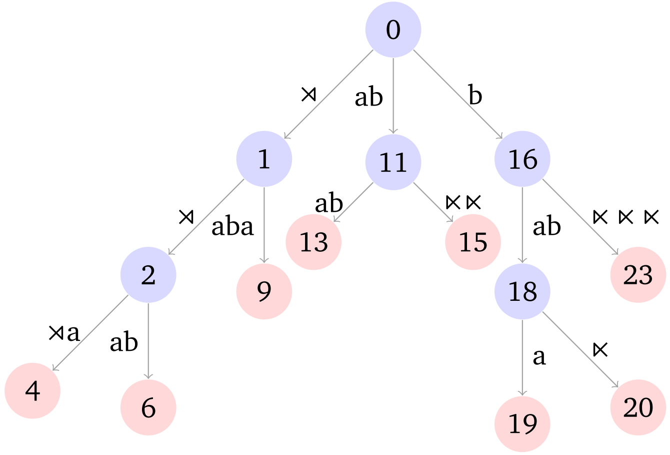
Consider the strictly 4-local grammar, which was also used in Fig .4.1.

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

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.

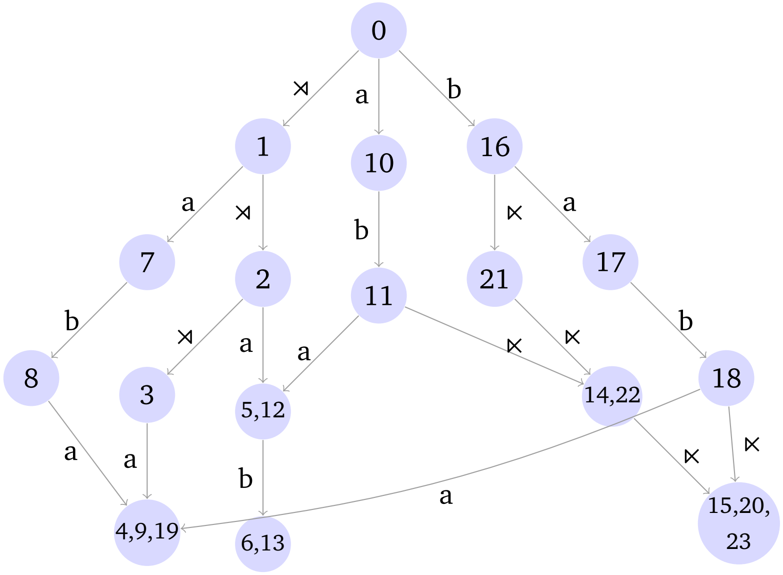
rather simple and can be captured with very small grammars. Also keep in mind that few processes require more than bigrams and trigrams, so if we have one grammar for each process and then enforce them all in parallel, we can reduce storage needs enormously compared to one monolithic strictly 5-local 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).

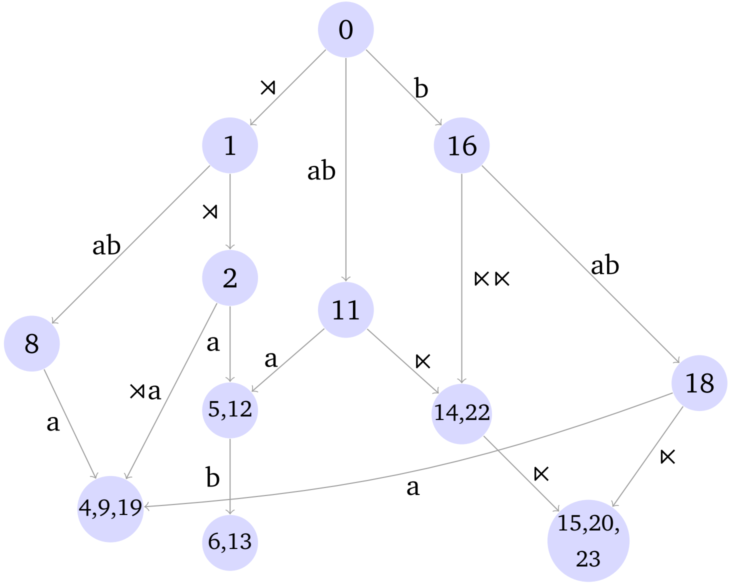
You might have also noticed that some branches are duplicated. To give but one example, the nodes 11 and 21 in the prefix tree of example 5.1 share the same subbranch ⋉-⋉. We can combine these branches by first merging the nodes 14 and 22, and then 15 and 23. This does not change the grammar because 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.

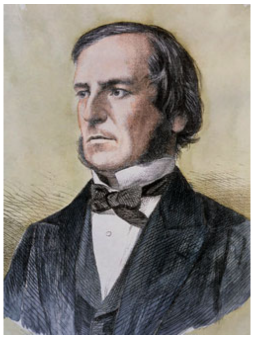
Example 5.3 Adjacency Matrix for a Radix Tree

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 $i_1, \ldots, i_k$, 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

```
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, $\ell(\langle u, v \rangle) = \sigma$,

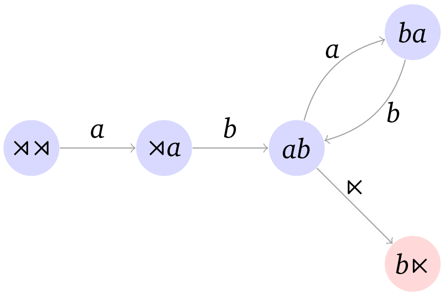
- there is a unique root $u \in V$, which must be $\rtimes^k$.

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 $(ab)^+$.



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. ⌟

Relevant literature for Unit 5

add more info

Unit 6

Learning Local Dependencies

Strictly local grammars provide a model for local phonological processes that is sufficiently powerful (all local processes can be correctly described), cognitively plausible (low memory load, efficient runtime behavior), and captures some essential properties of natural language phonology (linguistic creativity, generalization from short strings to longer ones, some processes are more complex than others, constraints can apply conjunctively but not disjunctively).

Even in the best case, though, this is insufficient to attain full *explanatory adequacy* in the terminology of Chomsky (1965). Recall the three levels of adequacy:

1. **Observational Adequacy**
   The theory correctly accounts for all the observed data.

2. **Descriptive Adequacy**
   The theory describes a native speaker's knowledge of their language. Hence it gives a full specification of their grammar and thus makes the right predictions even for unobserved data.

3. **Explanatory Adequacy**
   The theory explains how a native speaker acquires their knowledge from the primary linguistic data.

Strictly local grammars are certainly observationally adequate since they can handle every phonological process as long as its application domain is bounded in size. Descriptive adequacy requires the formal model to mirror the speaker's internal grammar; a standard which is rather hard to evaluate. At the very least strictly local grammars can generalize beyond finite data, make typological claims, and offer a rigorous definition for what grammars look like (they are finite sets of *n*-grams). So they certainly go beyond observational adequacy and might even be descriptively adequate — provided that phonology involves no mappings from underlying forms to surface forms, which is something these grammars do not handle.

Explanatory adequacy is all about the acquisition of the grammar. Given a (possibly infinite) collection of grammars to choose from, how does a speaker arrive at a descriptively adequate grammar from a finite sample of input data? This is an issue we haven't addressed at all yet. As we will see today, strictly local grammars can be learned rather easily given that a crucial piece of information is already specified via our genetic endowment (i.e. Universal Grammar in generative terminology). So strictly local grammars do offer a nativist account of how local phonological processes can be acquired.

Linguists commonly use the term explanatory adequacy in a more general sense nowadays to describe any kind of account that explains a given phenomenon rather than just stipulating a grammar with a bunch of properties that correctly account for the data. Chapter 1 of Chomsky (1965), on the other hand, clearly ties it to matters of learnability.

1 Machine Learning versus Learnability

The question of how grammars can be learned is also of great importance in computational linguistics and NLP. On a purely practical level, it isn't feasible to hand-write grammars for all the languages in the world, not even if we only focus on their phonology. And even if that were possible, hard-coded grammars cannot adapt to local dialects or ongoing changes in the speech community. So it is preferable to have an algorithm that constructs and modifies grammars based on the input it gets. The development, evaluation, and efficient deployment of such algorithms is the purview of *machine learning*.

Unsuprisingly, learning algorithms that work well in real-world applications are very complex beasts that can be hard to understand and analyze. So it is often more insightful to study a more abstract problem: given a possibly infinite space of languages, can we identify properties of this space that guarantee the existence of a learning algorithm provided certain types of input? This question is studied in *learnability*.

Just like the difference between computational linguistics and NLP, the difference between learnability and machine learning is often subtle. The most important distinction, however, is that learnability is not directly about learning/language acquisition but rather about studying the structural properties of language classes and how they facilitate generalization.

In a certain sense, learnability is linguistics at the level of languages rather than the objects these languages describe. In linguistics, we identify structural generalizations for sentences and words — encoded via a grammar — that can be invoked to determine well-formedness and make typological predictions. In learnability, we also identify structural generalizations, but we do not look at the objects within a language but rather at all languages within a given class. The goal is to identify structural properties of the class of languages that are to be learned. More formally, linguistics looks at sets of objects, and learnability looks at sets of such sets. This point of view highlights that learnability is not directly concerned with learning as such, although its result have important ramifications for learning (just like computational linguistics is not about processing language with computers but its results can be applied this way).

Machine learning, on the other hand, is about solving any kind of problem that involves a learning component. This naturally makes it a much messier affair. If you want to teach a computer to filter spam mails, that is a learning problem, but it is not immediately obvious how one could formalize this as a language learning problem or what it even means for the learner to work correctly — although it is very easy to tell for the user when their spam filter screwed up. At any rate the problem certainly isn't approached in terms of, say, a space of spam languages and how its properties can be exploited for learning. That's not to say that such considerations never factor into the initial design of the learning algorithm, but they are not the central object of study.

There are also huge methodological differences between learnability and machine learning. Learnability is all about mathematical theorems and proofs, whereas experiments and simulations are much more common in machine learning. Just like with computational linguistics versus NLP, that is often a matter of necessity: the learning algorithms used in machine learning are so complex, and the problems so difficult to define in formal terms, that proofs simply aren't feasible. And of course machine learning has a stronger focus on solving real-life problems, so pretty much anything goes as long as it gets the job done, even if nobody can tell why it gets the job done.

So which one of the two perspectives, learnability or machine learning, is more insightful for linguistics? It depends on what kind of linguistics we have in mind. Linguistic performance is a tricky problem with lots of noise, few clear-cut generalizations, and no clear success state for the learner, so a problem like, say, speakers' use of slang words and how it is contingent on the social setting is probably better off with machine learning. Linguistic competence, on the other hand, is sufficiently abstracted that we can have rigorous mathematical definitions of the object of study and what it means to successfully solve a given learning problem. And since mathematical proofs offer the highest standard of truth that we should strive for whenever possible, learnability is the better perspective for our purposes.

2 A Closer Look at Learnability

2.1 General Remarks on Learning

The definition of explanatory adequacy given at the beginning require the theory to explain how a native speaker acquires their grammatical knowledge. But what exactly does that mean? For the sake of simplicity, let's assume that "explain" here just means that we have a learning algorithm, or simply *learner*. Chomsky might have had a stronger condition in mind, e.g. that the learner must somehow be defined via the grammar formalism, but with highly abstract issues like this it is always more instructive to consider the simplest case first before piling on additional complicating factors. Assuming then that any learner will do, what exactly is it the learner has to accomplish? What does it mean to acquire grammatical knowledge, how can we translate that into a mathematical statement?

There are of course many perfectly valid answers to this, so we will have to choose between these alternatives according to what exactly it is we are interested in. First of all, there are two ways of thinking about acquiring grammatical knowledge: learning a language, and learning the grammar for that language. In the first case, a learner of phonology just has to arrive at a state where one can tell for every word whether it is well-formed. In the second case, the learner must also use the right grammar. This is usually how linguists think about acquisition, but it is a rather curious notion because many grammars generate exactly the same language.

For example, if G is a strictly 2-local grammar, and $G' := G \cup \{\ltimes\rtimes\}$, then the two generate exactly the same language because $\ltimes\rtimes$ is a useless bigram, it never occurs in any string. So a learner that acquires G' instead of G is still learning the correct language, only the grammar is slightly larger than it needs to be. Similarly, if strictly local grammars are actually formalized as lists rather than sets, then two grammars can be distinct despite containing exactly the same n-grams just because the order of n-grams differs. So for every grammar G there are $(|G|-1)!$ many alternative grammars that generate exactly the same language. As a result there is no way to distinguish among these grammars based on well-formedness judgments.

The tacit assumption among linguists is that in such cases smaller grammars are to be preferred, and that differences in grammar can be detected with more sophisticated methods, e.g. psycholinguistic experiments. But the first one is a stipulation at best and a category mistake at one: while it is true that as scientists we should strive to find the simplest solutions and thus the smallest grammar, it is far from obvious that this scientific principle should be a condition on the learner. There are many cases where constructing a grammar for a formal language is possible yet one cannot determine whether there is a smaller grammar. If grammar size is an important factor in distinguishing adequate from inadequate learners, explanatory adequacy is unattainable under such circumstances. So if human language happens to be such a case, explanatory adequacy could never be achieved, which seems rather odd. The natural conclusion, then, is that grammar size is not an essential criterion.

The second assumption is that grammars can be distinguished even if they generate exactly the same language. Given a suitable linking theory between competence and performance, it is indeed conceivable that two grammars may make different predictions, e.g. in parsing, word recognition or possibly even the brain patterns on can observe. But to date there is no consensus on what such a linking theory has to look like, and more importantly, we do not know that speakers all have the same grammar in this sense. For example, whether a somebody is right-handed or left-handed has a profound difference on brain function, so we cannot rule out that right-handed people's grammars are different from that of left-handed people. Similarly, a psycholinguistic experiment such as a reading tasks does not measure the grammar in isolation but rather the compound output of working memory load, attention span, and so on. Unless there is a way to control for all these variables to a level of degree where one can detect even subtle differences in grammar, there is no evidence that speaker's learn the same grammar. In fact, behavioral data suggests the opposite: even when speakers agree on all relevant data judgments for a given phenomenon, they can still have different preferences for certain structures, or they may be more likely to use one than the other. Overall, then, there is no strong reason why "acquiring grammatical knowledge" must mean finding the right grammar rather than the right language.

Another important factor missing from the definition of explanatory adequacy are the parameters of learning, i.e. what kind of evidence the learner can draw from and how quickly learning must proceed. If our main interest is modelling human learning, then those issues are first and foremost an empirical matter, and whatever is available to an infant learner should also be available to our formal learner. The problem, though, is that it often unclear what evidence infants can draw from. We know for sure that they get positive evidence by virtue of constantly being exposed to natural language input. But they might also have access to negative evidence, for if you never hear a specific form or construction, that makes for strong probabilistic evidence that it is ill-formed. Setting aside the question of what counts as input, we also do not know how much information is extracted from that input. The actual surface forms — sequences of sounds for phonology, sequences of words for syntax — are definitely accessible, whereas it is much less clear how much syntactic structure can be inferred from prosody and semantics. The only thing we know for sure, then, is that language acquisition involves at least collections of surface forms. The best starting point, then, is to see what can or cannot be learned with such a restricted amount of information.

2.2 The Gold Paradigm

The learnability literature is full of different learning paradigms that make varying assumptions about the input available to the learner and the criteria that determine learning success. One of the oldest and most influential is the *Gold paradigm*, also known as *learning in the limit* or *learning in the limit from positive text*. The Gold paradigm makes very minimal assumptions about the input while enforcing a very strict notion of learning success. The learner is presented with strings from the target language, one after another. After each string presentation the learner formulates a theory (represented by a grammar) as to what language that string was drawn from. The learner is successful iff it guesses the right language after a finite number of steps and, crucially, does not change its guess at a later point.

Let us make these ideas more precise so that we know what exactly we are talking about before moving on.

Definition 6.1 (Text). Given a language L, a *(complete, positive) text* over L is a total mapping T from the natural numbers onto $L \cup \{\#\}$ (that is to say, it is an infinite string in which ever member of L occurs at least once and random noise is represented by $\#$). We denote by $T(i)$ the $i+1$-th element in the text, whereas $T[i]$ represents the length i prefix of T, i.e. $\langle T(0), T(1), \ldots, T(i-1)\rangle$. For every sequence s, *content*(s) is the set of elements of L that occur in s.

Definition 6.2 (Gold learning). A *learner* is a (possibly partial) function ϕ from sequences of expressions and # to languages. The learner *converges* on text T iff there is some i such that $\phi(T[i]) = \phi(T[j])$ for all $j \geq i$. The learner *identifies* language L iff it holds for every text T over L that ϕ converges on T and guesses the correct language (that is to say $L(\phi(T)) = \mathit{content}(T)$). Similarly, ϕ identifies a class $\mathscr{L}$ of languages iff it identifies every language in that class. A class of languages is *identifiable* or *learnable (in the limit)* iff there is a learner that identifies it.

Example 6.1 Every Language is Learnable

Here is a quick example that shows all the terminology and concepts of Gold learning in action. It also illuminates a common source of confusion: learnability is a property of language classes, not individual languages. When considered in isolation, every language is learnable.

Let L be some language. We show that $\{L\}$ is learnable in the limit by exhibiting a learner that identifies $\{L\}$. This learner is the constant function ϕ such that for every expression i $\phi(i) := L$.

Since ϕ is a constant function it makes the same guess for every prefix of any text T over L, so we have $\phi(T[i]) = L$ for every $i \geq 0$. This immediately implies that the learner converges on T, and since T is drawn from L we also have that $L(\phi(T)) = \mathit{content}(T)$. This will be the case for every text T over L, so ϕ identifies L, which means that it also identifies the class $\{L\}$.

The order of steps in the example is representative of learnability proofs in general.

1. Give a learner.
2. Show that the learner converges, that is to say, it does not alter its guess after a certain point.
3. Show that after it converges, the learner correctly guesses the target language. In other words, it identifies the language.

4. Show that the learner can do this for all languages in the class and all texts over them.

At this point you might already have a hunch that the Gold paradigm is not a fully realistic representation of language acquisition. You would be right, many factors are omitted or simplified. There are no constraints on how quickly a learner has to converge, so a learner that takes 300 years to find the target language is perfectly acceptable. The learner also has perfect recall since its guess at position i in the text is not just determined by the element at this position, which we denote $T(i)$, but by all the elements up to this position, which is represented by the minimally different $T[i]$. Moreover, the text contains a lot of information: noise is explicitly marked as such so that the learner is never mislead by faulty input, and every sentence of the language appears must appear in every text at some point. These four factors simplify the learning problem a lot.

At the same time, some factors make the Gold paradigm harder than the real life task. In particular, the learner has to succeed on every text, no matter how defunct and degenerate that text is. It is highly unlikely that a child could learn from input that consists predominantly of sentences from Wall Street Journal op-ed pieces — child-directed input starts out easy with simple short sentences and increases gradually in complexity. The Gold paradigm requires the learner to succeed even under highly adversarial circumstances. And the restriction to strings that are completely stripped off prosody and semantics also heightens the difficulty of the task.

It is important to keep in mind though that these aren't so much issues of empirical inadequacy as they are consequences of mathematical abstraction. Putting restrictions on the structure of the text makes proofs more involved, and including prosodic or semantic information requires more complex objects than just strings. By keeping the paradigm as mathematically simple as possible, we can quickly gain insights into why certain classes of languages are learnable and others are not, and these insights can be adapted to more adequate paradigms in subsequent work. The process isn't all that different from what we are currently doing in our analysis of phonology: set out the bare essentials that need to be accomplished, see how one might go about this with the least amount of assumptions, and where necessary revise the model in a conservative fashion so that one can easily tell which properties are being preserved, and why.

2.3 Most Classes of Languages are not Learnable

One of the key insights of the Gold paradigm is that a surprisingly large number of language classes is not learnable. More precisely, ever proper superset of the class of finite languages is not identifiable in the limit. This is even more surprising because the class of finite languages is learnable.

Theorem 6.3. The class Fin of finite languages is identifiable in the limit. ⌟

The key idea is that every finite language can be learned within a finite number of steps by simply memorizing all the forms encountered so far. Remember that this was our original idea for the list phonology model, so this theorem tells us that list phonology is a learnable model of phonology.

Proof. Let ϕ be a function that, for every text T and index i, maps $T[i]$ to $\mathit{content}(T[i])$. For every finite language L and text T over L, there must be a smallest i such that *content*$(T[i]) = L$. Thus $\phi(T[i]) = \phi(T[j])$ for all $j \geq i$, wherefore the learner converges on T. But we also have $\phi(T[i]) = \mathit{content}(T[i]) = L$, and since T was arbitrary ϕ identifies L. But L, too, was arbitrary, showing that ϕ identifies Fin. □

The learner used in the proof is rather boring because it only memorizes forms without any kind of generalization — one of the biggest qualms we had with list phonology model. But generalization is risky as it involves formulating a hypothesis that goes beyond the mere data. Depending on how the space of languages is structured, it may be more or less safe to generalize in certain ways. In the worst case, safe generalization is impossible, and that is exactly the problem with any language class that properly includes all finite languages.

Theorem 6.4. Every class of languages that properly subsumes Fin is not identifiable in the limit. ⌟

Note that every class of this type must include at least one infinite language. So a learner that tries to learn this class must guess this infinite language after having seen only a finite amount of data — that's the convergence criterion. But there is no guarantee that this finite amount of data isn't actually sampled from a finite language that is included in the infinite one, in which case the learner would have accidentally overgeneralized. Since there is no negative evidence in the Gold paradigm, the learner has no opportunity to revise its decision and makes the wrong guess.

Proof. We show that $\mathscr{L} := \text{Fin} \cup \{L\}$ is not identifiable, where L is some infinite language. This immediately implies that no proper superset of Fin can be identified in the limit.

Suppose that ϕ identifies $\mathscr{L}$. Then ϕ must identify every member of $\mathscr{L}$, including L. By definition, then, it must hold for every text T over L that there is some i where ϕ converges on T. Clearly *content*$(T[i])$ is a finite language F. Let T_F be a text over F such that $T_F[i] = T[i]$. Then ϕ must also converge on T_F at index i, but since ϕ identifies L over T it must be the case that $\phi(T_F)$ is L rather than F. So ϕ does not identify F over T_F, wherefore it does not identify F or $\mathscr{L}$ either. □

The two theorems home in on the central problem of learning a given space of languages in the Gold paradigm: on the one hand the learner has to generalize from a finite amount of input to an infinite language, but on the other hand this generalization step needs to be conservative enough that the learner can avoid from overgeneralization. Other learning paradigms may give the learner some means of recovering from overgeneralization to some degree — e.g. via probabilistic reasoning, negative evidence, or an omniscient oracle that can be asked specific questions — but the tension between finite data and infinite target languages remains. This tension can be resolved correctly only if the space is structured in a specific way and the learner is aware of that structure.

There are some clear parallels to Universal Grammar (UG) in linguistics. However, UG does not really talk about the structure of the language space but rather puts restrictions on what the elements of each language in that space look like. By itself that is not enough to guarantee learnability (unless the restrictions are severe enough to render the space of languages finite, see Sec. 3.2). Nonetheless UG considerations and learnability can be linked in intriguing ways, as we will see next in our look at the (non-)learnability of strictly local languages.

3 Learning Strictly Local Languages

3.1 SL$_k$ is Learnable

Remember that the class of strictly local languages properly subsumes the class of all finite languages, so this immediately tells us that the full class is not Gold-learnable.

Corollary 6.5. SL is not identifiable in the limit. ⌟

But the full class of strictly local languages isn't all that interesting for our purposes anyways. By definition there is an upper bound k on the domain of local phonological processes, otherwise they would be non-local processes. It seems reasonable that the value of k is part of UG so that human learners only entertain hypothesis that fall within the realm of strictly k-local languages (or maybe there is some language external factor that limits the class of hypothesis to SL$_k$; at any rate there is some learning prior in place). And in contrast to SL, SL$_k$ is learnable for every choice of k.

Theorem 6.6. SL$_k$ is identifiable in the limit for every $k \geq 0$. ⌟

By now you probably know already what the learner has to do: memory all k-grams of all strings seen so far.

Proof. Let ϕ_k be such that for every text T of strictly k-local language L, $\phi_k(T[i]) = L(G)$, where $G := \bigcup_{w \in content(T[i])} k\text{-grams}(w)$ is a positive strictly k-local grammar. Since there are only finitely many distinct k-grams, ϕ_k is guaranteed to converge at some index i. At this point, G contains all k-grams that occur in some string in T, and only those, so $L(G) = L$. Since L and T were arbitrary, ϕ_k identifies SL$_k$. □

While the result is stated in terms of learning, we can also view it as a description of the language space that ϕ_k induces. That is to say, given some finite input sample, ϕ_k will generalize to a (possibly infinite) language, namely the smallest strictly k-local language that includes the input. So if all phonological dependencies are strictly k-local, that might not necessarily be due to a restriction of the grammar. Maybe phonological computations could work just as well with much larger values, or even the whole class of strictly local languages. But the learner ϕ_k only infers SL$_k$-grammars. No matter what input it is presented with, it will always generalize to a specific type of grammar. This allows us to factor language as an empirical object into at least two distinct components: the class of languages that can be generated by the grammar formalism, and the class of languages that are identified by the learner. Natural languages are the special type of language that belongs to both. So you see, learnablity is not just tied to language acquisition, it can also be invoked to explain typological facts.

3.2 Every Finite Class is Learnable

On a mathematical level, the learnability of SL$_k$ is actually a corollary of a more fundamental fact: every finite class of languages is learnable. Intuitively, that's because the languages in a finite class can be tried one after another to test if they fit the text. As long as one does not pick a needlessly large language, overgeneration can be avoided and the right language will be found eventually. Since there are only a finite number of k-grams for every fixed alphabet, SL$_k$ is finite as well and thus can be explored in this fashion.

Theorem 6.7. Every finite class $\mathscr{C}$ of languages is learnable. ⌟

Proof. Assume w.l.o.g. that $\mathscr{C}$ contains only n languages. Put these languages into order $L_1 \cdots L_n$ such that for all L_i, L_j, $i < j$ implies that L_i is not a proper superset of L_j. We now define a learner ϕ that always picks the "left-most" language that subsumes the content of the text: for all i, $\phi(T[i]) = L_k$, where $L_k \supseteq \mathit{content}(T[i])$ and there is no L_j such that $L_j \supseteq \mathit{content}(T[i])$ and $j < k$. Obviously ϕ converges on every text.

To see that ϕ also identifies $\mathscr{C}$, suppose that T is some text over $L_k \in \mathscr{C}$. Once ϕ converges at $T(i)$, it conjectures some $L_j \supseteq \mathit{content}(T[i])$. It cannot be the case that $j > k$ because ϕ picks the language with the lowest index. If $j < k$, then $L_j \not\supseteq L_k$, so there is some string $w \in L_k$ such that $w \notin L_j$. But w must occur in T at some point p, so $w \in \mathit{content}(T[p]) \nsubseteq L_j$, contradicting the earlier assumption that ϕ converges at $T(i)$ with $\phi(T[i]) = L_j \neq L_k$. The only possible case, then, is that $j = k$, wherefore $L_j = L_k = \phi(T[i])$. □

This theorem has several important implications. First of all, finite language classes are trivial to learn because we can always induce a suitable structure over this class that guarantees its learnability. Interesting learnability results are about infinite language classes, or home in on properties of finite language classes that make learning more efficient or would also guarantee learnability if the class were in fact infinite. Fortunately that is the case for the SL$_k$-learner.

3.3 Generalizing the Learner

The proof that SL$_k$ is identifiable in the limit is deceptively simple. In just five lines it establishes the learnability result, but it does not quite home in on why this works, what aspects of strictly local languages are essential for the proof, and what exactly the explored space looks like. In particular, is the learnability of SL$_k$ simply due to the fact that every finite class of languages is learnable, or is there more to SL$_k$ that would guarantee learnability even for These are important points; if we want to revise our formalism at some later point without losing the learnability property, we need to know what we can safely change.

Let's first think about the overall shape of SL$_k$. One useful trick is to identify each language with a grammar that generates it and think about that space of grammars instead. After all, one is just a different representation of the other. Suppose we are interested in the strictly 1-local languages over a and b. There are 4 different 1-grams over this alphabet: $\rtimes$, $\ltimes$, a, and b. Since every strictly 1-local grammar is a set of these 1-grams, the class of SL$_1$ grammars over a and b is the powerset of the set of these 4 bigrams. This means that there are $2^4 = 16$ different SL$_1$-grammars, and these grammars are partially ordered by the subset relation. We can actually represent this space graphically as a Hasse diagram, see Fig. 6.1.

Background Partially ordered sets

A (weak) partial order is a binary relation R over a set S that satisfies three conditions:

reflexive for all $x \in S$, $x \mathrel{R} x$

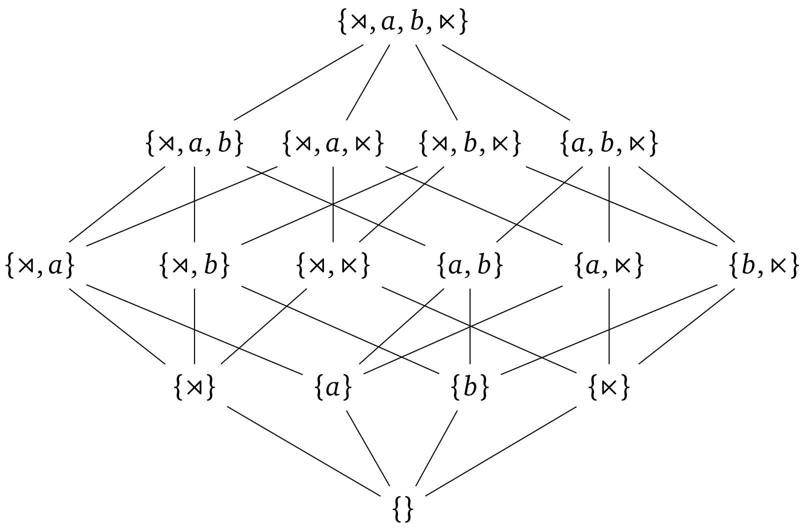


Figure 6.1: Lattice representation of the space of SL_1-grammars over a and b

antisymmetry for all $x, y \in S$, $x \mathrel{R} y$ and $y \mathrel{R} x$ jointly imply $x = y$

transitivity for all $x, y, z \in S$, $x \mathrel{R} y$ and $y \mathrel{R} z$ jointly imply $x \mathrel{R} z$

A partially ordered set, or *poset*, is a set with a partial order defined over it.

As you can see this space has a lot of structure. First of all, there are unique bottom and top elements, which are $\{\}$ and $\{\rtimes, a, b, \ltimes\}$ in this case. It also holds that any two nodes have a unique greatest lower bound (*infimum*) and a unique least upper bound (*supremum*). For example, $\{\rtimes, a\}$ and $\{\rtimes, b\}$ have the greatest lower bound $\{\rtimes\}$ and the least upper bound $\{\rtimes, a, b, \}$. A partially ordered set where all nodes x and y have both an infimum and a supremum is called a *lattice*, and if it also has unique bottom and top elements it is a *bounded lattice*.

It is fairly easy to see that the class of SL_k grammars always forms a bounded lattice, the size of which depends on k the number of symbols in the alphabet. For example, there are 9 distinct bigrams over a, b (strictly speaking there are more, like $\ltimes a$, but those can never occur in a string), so the corresponding lattice has $2^9 = 512$ distinct nodes, which is way too much to draw. Still, they are all lattices.

The strategy used by ϕ_k can be viewed as a particular way of moving through such a lattice. The learner starts at the bottom of the lattice and the moves upwards whenever the current input cannot be accounted for by the grammar represented by the current node in the lattice. In the specific case of SL_k learning, this means that the input string contains a k-gram that is not part of the current grammar. Crucially, though, the learner can determine the shortest upward move that will take it to a grammar that can deal with all the input so far. Suppose that ϕ_1 is currently at the node $\{\rtimes, a, \ltimes\}$ and is now presented with the input b. That string is not generated by the grammar $\{\rtimes, a, \ltimes\}$, as its set of 1-grams is $\{\rtimes, b, \ltimes\}$. The learner infers from this that the grammar is $\{\rtimes, a, b, \ltimes\}$, but notice what is happening in the lattice: the learner isn't just moving upwards, it's moving to the supremum of the nodes that correspond to these two grammars!

By moving to the supremum rather than just any random higher node, the learner generalizes in the most careful and conservative way possible that is consistent with the data, which avoids overgeneralizing. If the supremum isn't the right grammar, then there will be more evidence further down the line and we can move to the next higher supremum, and so on. If, on the other hand, the learner simply moved to some arbitrary higher node, there might be no positive evidence that it has the wrong grammar and thus no way for it to correct its mistake.

The lattice-based learning perspective is so appealing because it is both intuitive and general. It doesn't really matter that the nodes in the lattice correspond to strictly local grammars, what matters is that the learner has a way to explore this space in a principled fashion without overgeneralizing. This can be done as long as the learner can derive a second node from the input such that the supremum of that second node and the learner's current position corresponds to the most conservative, empirically adequate revision of the learner's current hypothesis. So if a given class of languages can be shown to have a lattice structure — which is possible for both finite and infinite classes — and if we can define a method for picking the right suprema, then we have a learner for that class of languages.

This is also highly appealing from a psycholinguistic perspective since one would assume that learning strategies do not differ significantly for, say, local and non-local phonological processes. Once we move from local to non-local processes, we will see that this is indeed the case: both processes involve exactly the same lattice-based learning algorithms, with the only difference being what each lattice represents.

Relevant literature for Unit 6

add more info

Chomsky, Noam. 1965. *Aspects of the theory of syntax*. Cambridge, MA: MIT Press.

Unit 7

Beyond Well-Formedness

Suppose we want to implement the SL$_k$ learner from the previous lecture in Python. This involves two components, a mechanism for collecting all the k-grams in a string, and a data structure for memorizing all the k-grams seen so far across all read strings. A bigram collection mechanism was already part of the set-based scanner we devised in Cha. 3.

```python
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
```

This function can easily be extended to arbitrary k. In order to get a full learner, we only need a function that takes a finite sample of strings and runs the k-gram collection function on each (augmented) string in that sample. Below is one implementation of this for a bigram leaner.

```python
def bigram_learner(language):
    """Induce a strictly 2-local grammar from a finite sample of strings"""
    grammar = set([])
    for w in language:
        grammar = grammar.union(set(string_to_bigrams(augment_string(w))))
    return grammar
```

While this works, it is somewhat inelegant. Intuitively, the SL$_2$ learner above still has to use the scanner to extract the list of bigrams, yet the function does not directly call the bigram scanner but merely some of its subroutines. Consequently the program does not correspond to how we think about the actual computation. On a more practical level, all optimization steps and safeguards that might have been added to the scanner will be lost if they're not part of these subroutines. A better approach would thus be one where the scanner handles the job of extracting the bigrams in the most efficient way possible, and the learner directly calls the scanner.

You might be wondering how this is supposed to work if the scanner works incrementally, the way we originally described it. In this case the scanner never uses the function for collecting a set of bigrams, so how could it possibly supply such a set to the learner? Today we will see that there is a way to write the scanner in a more abstract way that allows it to carry out various tasks. The basic principles of the scanner will be preserved, but we add a few parameters to the mix. For the purposes of software engineering, this makes for a very versatile and easily maintained code base. But it is also interesting from a theoretical perspective, because it shows among other things that it does not matter all that much for formalisms whether they are categorical (well-formed VS ill-formed) or probabilistic (degrees of acceptability).

1 The Algebra of Composite Values

1.1 A Functional Recipe for Composite Values

Suppose that we want to compute the well-formedness of a given string with respect to some grammar. We have already seen two ways of doing this. In the definition of (positive) strictly k-local grammars, we let $L(G) := \{w \mid k\text{-grams}(\hat{w}) \subseteq G\}$, which is also the foundation of the set-based scanner: check whether the set of k-grams of the string is a subset of the grammar. The incremental scanner, on the other hand, checks at each step whether the current bigram is in the grammar. If one abstracts away from the temporal component of the scanner while keeping the piece-wise application, one can also define a well-formedness function f that computes the grammaticality value of a string from the grammaticality values of its k-grams.

Below is an example of a formula evaluating the string aba with respect to the bigram grammar $\{\rtimes a, ab, ba, b\ltimes\}$, which generates the language $(ab)^+$.

$$\begin{aligned} f(aba) &= g(\rtimes a) \times g(ab) \times g(ba) \times g(a\ltimes) \\ &= 1 \times 1 \times 1 \times 0 \\ &= 0 \end{aligned}$$

Here g is a function that returns 1 iff the bigram is contained in the grammar, and 0 otherwise. The function f then multiplies all these values in order to determine overall well-formedness. Note that we could have used other mathematical operations:

$$\begin{aligned} f(aba) &= 1 \times (1 \times (1 \times 0)) & &= 1 \min (1 \min (1 \min 0)) & &= 1 \wedge (1 \wedge (1 \wedge 0)) \\ &= 1 \times (1 \times 0) & &= 1 \min (1 \min 0) & &= 1 \wedge (1 \wedge 0) \\ &= 1 \times 0 & &= 1 \min 0 & &= 1 \wedge 0 \\ &= 0 & &= 0 & &= 0 \end{aligned}$$

So there are actually three different functions that all compute the grammaticality of a string.

While the functions behave exactly the same if g returns only 1 or 0 as values, they diverge in other cases. Suppose, for instance, that g tells us for each k-gram it's acceptability value, encoded as some real number in the interval $[0, 1]$. Then the choice of f produces different values. If f uses the logical operator $\wedge$ for composing values, we won't any output at all in most cases because $\wedge$ is only defined for argument that are 0 or 1. For the other two functions, we get vastly different values.

$$
\begin{aligned}
f_\times(aba) &= g(\rtimes a) \times (g(ab) \times (g(ba) \times g(a\ltimes))) \\
&= 0.9 \times (0.25 \times (0.5 \times 0.75)) \\
&= 0.9 \times (0.25 \times 0.375) \\
&= 0.9 \times 0.09375 \\
&= 0.084375 \\
f_{\min}(aba) &= g(\rtimes a) \min (g(ab) \min (g(ba) \min g(a\ltimes))) \\
&= 0.9 \min (0.25 \min (0.5 \min 0.75)) \\
&= 0.9 \min (0.25 \min 0.5) \\
&= 0.9 \min 0.25 \\
&= 0.25
\end{aligned}
$$

If we interpret g as a function from k-grams to acceptability values, then the function $f_\times$ treats overall acceptability as the product of the individual acceptability values of each k-gram. The function $f_{\min}$, on the other hand, tells us the value of the least acceptable k-gram that occurs in the string. So these functions do very different things, but they do not look all that different from any of the other we have seen so far. They may compute different values, but they apply a helper function g to each k-gram and the compute some compound value from the individual values returned by g.

This strategy isn't even restricted to numerical values, either, just about anything of interest can be computed by some f in this fashion. For example, here is a function $f_\cup$ that computes the set of bigrams for a string.

$$
\begin{aligned}
f_\cup &= g(\rtimes a) \cup (g(ab) \cup (g(ba) \cup g(a\ltimes))) \\
&= \{\rtimes a\} \cup (\{ab\} \cup (\{ba\} \cup \{a\ltimes\})) \\
&= \{\rtimes a\} \cup (\{ab\} \cup \{ba, a\ltimes\}) \\
&= \{\rtimes a\} \cup \{ab, ba, a\ltimes\} \\
&= \{\rtimes a, ab, ba, a\ltimes\}
\end{aligned}
$$

What we have here, then, is a general template for computing a string's compound value from the values of its parts, which happen to be k-grams. Depending on what those base values are (regulated through the choice of function g) and how we combine them (determined by the choice of f), we get different compound values.

1.2 Monoids

Despite the differences in values, the compound functions all behave pretty much the same on an abstract level. First, f operates on a fixed domain of values, and this domain is closed under f. Remember that a set S is closed under an operation o iff $o(s)$ is an element of S for every $s \in S$. The domain of grammaticality values, for instance, is $\{0, 1\}$, and applying $f_\times$, $f_{\min}$, and $f_\wedge$ within this domain can only produce 0 and 1 as outputs. Similarly, the real interval $[0, 1]$ is closed under $f_\times$ and $f_{\min}$. For $f_\cup$, the domain is the set of all sets of k-grams over alphabet Σ — $\wp(\Sigma^k)$ in mathematical notation — and obviously $f_\cup$ cannot produce a value that lies outside that set. We see, then, that each function is a map from a fixed domain into that very same domain.

It is also the case that the operation used by f to compute the composite values is associative. Remember, a function $\circ$ is associative iff $a \circ (b \circ c) = (a \circ b) \circ c$; in other words, brackets can be dropped without changing the composite value. This is clearly the case for ×, min, $\land$, and $\cup$. In all the examples above, we could have computed the values from left to right instead without affecting the final outcome. The order of application thus is irrelevant for all the variants of f in our examples.

Finally, every operation $\circ$ has an identity element $\mathbf{1}$ such that $a \circ \mathbf{1} = \mathbf{1} \circ a = a$ for all a. For ×, min and $\land$ this identity element is literally 1:

$$\begin{aligned} 0.75 \times 1 = 1 \times 0.75 = 0.75 \\ 0.75 \min 1 = 1 \min 0.75 = 0.75 \\ 0 \land 1 = 1 \land 0 = 0 \end{aligned}$$

Can you explain why $\emptyset$ must be in the domain of $f_\cup$?

For $\cup$, the identity element is $\emptyset$ instead:

$$\{ba, a\ltimes\} \cup \emptyset = \emptyset \cup \{ba, a\ltimes\} = \{ba, a\ltimes\}$$

If you have some background in higher algebra, you may already know what kind of object is characterized by these properties: *monoids*.

Definition 7.1 (Monoid). A *monoid* $M := \langle S, \otimes \rangle$ is a set S with an operation $\otimes$ such that:

- S is closed under $\otimes$, and
- $\otimes$ is associative, and
- $\otimes$ has an identity element $\mathbf{1} \in S$.

Use $\otimes$ as a general placeholder, we can combine all the functions we have seen so far into a single template for a k-gram function f that computes composite values.

$$f_\otimes(a_1 \cdot a_2 \cdots a_{n-1} \cdot a_n) := g(a_1 \cdots a_k) \otimes \cdots \otimes g(a_{n-k} \cdots a_n)$$

Each $g(a_i \cdot a_{i+k})$ must be an element of S, and $\otimes$ is some function over S such that $\langle S, \otimes \rangle$ is a monoid.

The monoid perspective is appealing for two reasons. On a theoretical level it highlights computational similarities between superficially different processes. Computing the grammaticality of a string isn't all that different from computing its set of k-grams. The switch from a categorical grammar to a probabilistic one — which is a contentious issue among linguists — is tantamount to replacing the set $\{0, 1\}$ by the real interval $[0, 1]$, everything else stays exactly the same. Therefore every property that relies only on $\langle \{0, 1\}, \times \rangle$ being a monoid immediately carries over to $\langle [0, 1], \times \rangle$ and is consequently preserved by the switch from a categorical to a probabilistic formalism. Quite simply, monoids are yet another tool for keeping our results general and widely applicable without sacrificing mathematical rigor or linguistic substance.

Their generality is also what makes monoids interesting for a very different crowd: programmers. Programmers want their code to be simple, easily maintainable, and free of unnecessary code duplication. Yet the code should also be flexible so that it can be quickly adapted to new problems as they arise. Monoids offer a way of attaining both goals. One can write a basic program that works with any kind of monoid, and applying this program to a specific problem instance requires no more than defining the appropriate monoid. Going back to our problem at the outset of this chapter, this strategy allows us to write an incremental scanner that can be used directly by the learner to retrieve a string's set of k-grams.

2 Implementing a Monoid Scanner

A monoid scanner works exactly like the function $f_\otimes$ above: given a function for computing the base values and a suitable operation for combining the base values, it determines the composite value in a recursive fashion. This can be translated into python almost literally, with the actual code being only 5 lines.

```python
def monoid_scanner(base_value, compose, grammar, w):
    """
    Generalized bigram scanner that assigns strings
    a value over a given monoid.

    Arguments:
    base_value -- maps a bigram to an element of the monoid
    compose    -- implements the monoid operation
    grammar    -- set of licit bigrams
    w          -- the input string
    """
    # [Base Step]
    # value of a single bigram
    if len(w) == 2:
        return base_value(w, grammar)
    # [Recursion]
    # scan string from left to right and compose values
    else:
        return compose(
            base_value(w[0:2], grammar),
            monoid_scanner(base_value, compose, grammar, w[1:len(w)]))
```

Note that for the sake of exposition, the monoid scanner only works with bigrams, but it could easily be generalized to work with arbitrary *k*-grams. The string augmentation function has also been omitted here, and the scanner does not include any safety checks for the input (e.g. to avoid type errors).

In order to turn the monoid scanner into a standard recognizer, we have to supply two functions that, respectively, determine for each bigram whether it is licensed by the grammar and compute compound grammaticality values.

```python
def boolean_base(w, grammar):
        return w in grammar


def boolean_compose(s, t):
        return s * t
```

The probabilistic version is almost exactly the same, we only have to switch to a different function for the base value (assuming that a probabilistic grammar is a dictionary where every *k*-gram acts as the key for a real number).

```python
def prob_base(w, grammar):
    return grammar.get(w, 0)
```

The set of all bigrams in the string is computed by feeding the two functions below into the monoid scanner.

```python
def set_base(w, *args):
    return set(w)


def set_compose(s, t):
    return s.union(t)
```

And if we want to know how many bigram tokens occur in the string, we switch to yet another pair of functions.

```python
def token_base(w, *args):
    return 1


def token_compose(s, t):
    return s + t
```

So now we have four different algorithms, in less than 20 lines of code! Every function is incredibly small and easy to understand, which makes debugging almost trivial. We can also mix and match these functions to create new algorithms. We could even combine them into bigger functions, for instance if we wanted to keep track of the number of tokens for each individual bigram. The modularity of this approach also means that any optimizations and tweaks we do to the monoid scanner benefits every algorithm we implement this way. This saves a lot of time and effort, and there's no unnecessary duplication of code or work.

That's not to say that the monoid scanner is always the best solution. Generality always comes at a price, and in this case it means that optimizations primarily take the form of optimizing the monoid scanner. This may make it impossible to implement certain speed hacks that work well for one problem but fail miserably for another one. This can be alleviated to some extent by activating these tweaks only if specific functions are being used, but that makes the code base more complicated, of course. So there is no elegant way around the trade-off between speed and generality, but for our purposes speed is an afterthought at best, whereas generality is one of the main goals.

The monoid perspective is an incredibly fruitful one for computational linguistics, and we have only seen the tip of the iceberg. Monoids can be combined in a specific manner to form an algebraic structure known as a *semiring*, and these play a very important role in parsing and the analysis of OT. We won't have time to explore these topics in great detail, unfortunately, so for now we have to be content with the fact that monoids at the very least simplify the code for the bigram learner at the beginning of this section:

```
def bigram_learner(language):
    """Induce a strictly 2-local grammar from a finite sample of strings"""
    grammar = set([])
    for w in language:
        grammar = grammar.union(
            monoid_scanner(
                set_base,
                set_compose,
                grammar,
                augment_string(w)))
    return grammar
```

Relevant literature for Unit 7

add more info

Unit 8

Non-Local Dependencies

Our project of devising a computational model of phonological processes has made great progress. We started out with phonology as a list of well-formed surface forms, but quickly realized that this cannot account for essential properties like linguistic creativity or that there are strong constraints on what is a possible phonological system in natural language. So we moved to a more abstract perspective where phonology is still a list, but rather than full surface forms we list the *n*-grams that may occur in a string. A string's well-formedness is no longer an idiosyncratic property determined by a single list lookup — it now is contingent on the well-formedness of its parts.

We saw that this perspective equates phonological systems with strictly local languages, which allows us to explore formal properties of phonology through the study of the class of strictly local languages. This class seems to be a good approximation of local processes: every local phonological process is captured by some strictly local language, and the class exhibits closure properties that replicate very basic typological facts. Given a UG-specified restriction on the size of locality domains, we even get a positive learnability result under highly adversarial circumstances. And a natural algebraic generalization in terms of monoids revealed that many of these insights do not hinge on a categorical understanding of well-formedness, they also apply in a probabilistic setting.

But linguists have discovered a lot of evidence that there is more to phonology than just local processes. If strictly local languages do not encompass all of phonology, then the properties we have discovered so far characterize only a subpart of phonology. The question is whether there are indeed aspects of phonology that are not strictly local, and if so, how we could revise our model in an empirically adequate fashion while maintaining the majority of the insights that we have worked so hard for.

1 Long-Distance Dependencies in Phonology

1.1 Navajo Sibilant Harmony is not Strictly Local

While many phonological processes have clearly delineated application domains of bounded size — e.g. two adjacent segments, within a syllable or across a single word boundary — some processes seem to apply across arbitrary distances. Navajo, for instance, exhibits *sibilant harmony*, where all sibilants in a word must match the anteriority of the right-most one. The sibilants in the Navajo phoneme inventory are specified for anteriority as follows (Martin 2005: 9):

[+anterior]	[−anterior]
s	ʃ
z	ʒ
tsʰ	tʃʰ
ts	tʃ
ts’	tʃ’

The presentation here abstracts away from certain complicating factors. In particular, the likelihood of sibilant harmony decreases with distance, but crucially sibilant harmony never becomes ungrammatical.

Sibilant harmony applies in a variety of cases in Navajo, but it is most apparent in compounds (Martin 2005: 10; the data is presented using a mixture of Navaja orthography and IPA).

xoʃ	‘cactus’	tʃaa	‘ear’
ts’óóz	‘slender’	nééz	‘long’
xosts’óóz	a specific type of cactus	tsaanééz	‘mule’

Since there is no upper bound on how many words a compound may consist of, and since the evidence suggests sibilant harmony can hold across an unbounded number of words within a compound, we are dealing with a truly unbounded process. We can express this as a formal theorem.

Theorem 8.1. Sibilant harmony is not strictly local. ⌟

Proof. This could be proved by exhibiting a productive pattern of Navajo compounding such that two sibilants with distinct anteriority values can be arbitrarily far apart from each other. Instead, we first convert sibilant harmony into a more abstract string language, and we then show that this string language is not strictly k-local for any choice of k.

Let l be a relabeling that replaces anterior sibilants by a, non-anterior sibilants by b, and all other segments by c. Then sibilant harmony is satisfied in all strings over alphabet $\{a, b, c\}$, and only those, that do not contain both a and b. Let L be the set of all such strings. Then L contains strings $s_{a,i} := ac^ia$ and $s_{b,i} := bc^ib$. Suppose L is strictly k-local. By k-local suffix substitution closure, $s_{a,k} \in L$ and $s_{b,k} \in L$ jointly imply $ac^kb \in L$. But this string violates sibilant harmony and thus cannot be a member of L, showing that L is not strictly k-local. Since k was arbitrary, L is not strictly local. □

Note that the proof starts out with a relabeling. At an earlier point we proved that strictly local languages are not closed under relabelings, so strictly speaking this proof cannot establish that sibilant harmony is not strictly local. Rather, it shows that sibilant harmony is not strictly local if one assumes that segments that aren’t sibilants contribute no useful information for determining well-formedness with respect to sibilant harmony.

From a linguistic perspective, this means that any tricks that might allow us to decompose the long-distance dependency into a local one are banned unless they have a visible effect on the surface form. For example, we cannot use anything like “invisible spreading” where the right sibilant has some feature that invisibly spreads to the adjacent segment to the left, and from there to the next one, and so one, until it reaches another sibilant and changes the value of anteriority feature. This is important to keep in mind: our claims about the complexity of various phonological processes are stated with the shared understanding that no hidden structure or distinctions can be invoked. Such hidden structures might exist, of course, but positing them blurs important boundaries and ties our claims to a very specific theoretical interpretation of the data — one that might easily be false. Before we try to unify distinct phenomena by recourse to more elaborate structures, we should try to discern to which degree these phenomena are actually distinct.

1.2 Strictly Piecewise Languages

Now that we have encountered a process that is not strictly local, we have to start looking for a model that is sufficiently powerful but not too different from strictly local languages and grammars. Let us take another gander at the proof above — determining which property of strictly local languages was used to establish their inadequacy might give us an idea what needs to be changed.

Adopting for a moment the perspective of a strictly k-local scanner, we quickly see the problem. In order to regulate the dependency between a and b, both have to be within the scanner's search window, but that is impossible whenever the distance between the two exceeds the size of the search window. But what if our search window wasn't in one solid piece, but could be broken up into smaller parts that move around independently? That is to say, what if a strictly k-local scanner didn't have 1 window of size k, but k windows of size 1 so that we could inspect the string not in contiguous blocks but rather *piece by piece*?

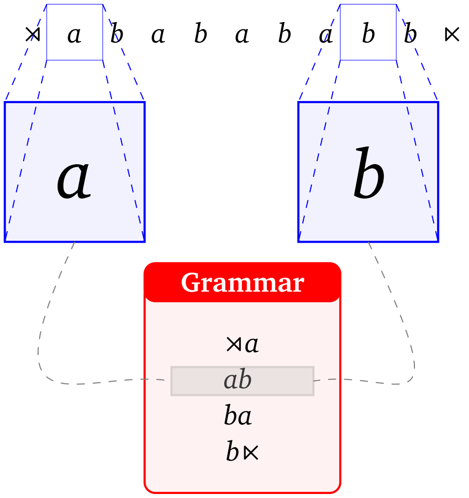


Figure 8.1: A scanner with multiple search windows

Suppose that we have such a scanner, with 2 windows of size 1. Then a string satisfies sibilant harmony iff it can never be the case that one of the windows is on an anterior sibilant a and the other window on a non-anterior sibilant b. And this is the case iff the string contains no *subsequence* ab.

Definition 8.2 (Subsequence). Given strings u and $v := v_1 \cdots v_m$, v is a *subsequence* of u iff every v_i occurs in u and for all $1 \leq i, j \leq m$, if $i < j$ then v_i precedes v_j in u (but they need not be adjacent). If v has length $k \geq 0$, it is a k*-subsequence*. The set of all k-subsequences of u is denoted k-seqs(u).

Example 8.1 Subsequences and Najavo Sibilant Harmony

The word xoʃ has the subsequences xo, oʃ, and xʃ, as well as x, o, ʃ, and xoʃ. Note that every substring is also a subsequence, but not the other way round. The 2-subsequences of tsaanééz are listed below:

ts	sa	aa	né	éé
ta	sn	an	nz	éz
tn	sé	aé		
té	sz	az		
tz				

Remember that strictly k-local grammars are defined in such a way that a string is well-formed iff every substring of length k is contained in the grammar. It seems that in order to account for long distance processes like Navajo sibilant harmony, we have to look at subsequences instead. So all we have to do is make a minor change to the definition of strictly local grammars.

Definition 8.3 (Strictly Piecewise). A *strictly k-piecewise grammar* is a finite set of k-grams. A positive strictly k-piecewise grammar ^{+}G generates the language $L(G) := \{w \mid k\text{-seqs}(w) \subseteq G\}$. A negative strictly k-piecewise grammar ^{-}G generates the language $L(G) := \{w \mid k\text{-seqs}(w) \cap G = \emptyset\}$. A language L is *strictly k-piecewise* iff it is generated by some strictly k-piecewise grammar. The class SP of *strictly piecewise languages* is given by $\bigcup_{k \geq 1} \{L \mid L \text{ is strictly } k\text{-piecewise}\}$.

As you can see the definition for strictly piecewise is almost exactly the one for strictly local, except that the definition of the generated language use k-seqs(w) instead of k-grams(w).

Example 8.2 A Negative Strictly 2-Piecewise Grammar for Sibilant Harmony

Navajo sibilant harmony is violated whenever a string contains a 2-subsequence consisting of sibilants with opposite anteriority features. Since there's only finitely many sibilants, we can list all forbidden 2-subsequences (for the sake of brevity let's assume that affricates count as two segments rather than one).

sʃ	sʒ	sʃ^{h}	ʃs	ʒs	ʃ^{h}s
zʃ	zʒ	zʃ^{h}	ʃz	ʒz	ʃ^{h}z
s^{h}ʃ	s^{h}ʒ	s^{h}ʃ^{h}	ʃs^{h}	ʒs^{h}	ʃ^{h}s^{h}

A negative strictly 2-piecewise grammar with these 2-subsequences will never generate a string that violates sibilant harmony.

1.3 Monoid-Based Implementation

In the previous chapter we generalized the bigram scanner for strictly 2-local languages to a monoid-based scanner that can carry out a variety of tasks besides acting as a recognizer. The code is repeated here for your convenience:

```python
def monoid_scanner(base_value, compose, grammar, w):
    """
    Generalized bigram scanner that assigns strings
    a value over a given monoid.

    Arguments:
    base_value -- maps a bigram to an element of the monoid
    compose    -- implements the monoid operation
    grammar    -- set of licit bigrams
    w          -- the input string
    """
    # [Base Step]
    # value of a single bigram
    if len(w) == 2:
        return base_value(w, grammar)
    # [Recursion]
    # scan string from left to right and compose values
    else:
        return compose(
            base_value(w[0:2], grammar),
            monoid_scanner(base_value, compose, grammar, w[1:len(w)]))
```

With a little bit of ingenuity, the code can also be used to construct a bigram scanner for strictly piecewise languages. The idea is that the base value function returns two values, a set with the currently read bigram and a boolean indicating whether this subsequence is licensed by the grammar. The composition function then uses this information to construct a list of all seen subsequences and checks for each newly constructed subsequence that it is licensed by the grammar. So in a sense we actually have two monoids operating in a synchronized fashion: one for computing sets of seen subsequences, and the other one for computing string well-formedness based on these sets. The full code for strictly 2-piecewise languages is given in Listing 8.1 on the following page.

1.4 Properties of Strictly Piecewise Languages

Since strictly piecewise languages are the result of coupling the format of strictly local grammars with a different method for determining string well-formedness — precedence instead of adjacency — all the proofs that rely only on the grammar format carry over without modifications. This implies immediately that we can freely convert between positive and negative SP grammars, and that SP_k is learnable in the limit from positive text for every k. Other properties require only a little bit of extra work. The proof for closure under intersection is easily adopted, as is the proof for non-closure under union (and thus relative complement). The argument that all finite languages are strictly local also establishes that they are strictly piecewise, which furthermore entails that SP is not learnable in the limit from positive text.

The strictly piecewise languages thus retain all essential properties of the strictly local languages, which isn't too surprising seeing how they are both n-gram formalisms. Their properties line up very well with what we know about phonology — they do not give a full characterization of phonology, but what they do claim seems to be along the right track. It is very tempting to surmise that natural language phonology is somewhere in the union of SL_j and SP_k for some UG-specified values of j and k. In

```python
def piecewise_base(w, grammar):
    """
    Check that input bigram is in the grammar and return:

    1) set of seen bigrams (= just the input),
    2) boolean indicating well-formedness, and
    3) the grammar so that it can be referenced by the compose function
    """
    return set([w]), w in grammar, grammar


def piecewise_compose(s, t):
    """
    Each argument is a triple of values:

    first component  -- set of 2-subsequences
    second component -- boolean indicating well-formedness
    third component  -- 2-gram grammar for which well-formedness is tested

    Based on this argument, the function computes:

    1) the set of all 2-subsequences that have been seen so far,
    2) the compound boolean.
    """

    # check that nothing went wrong with the grammar and give it a name
    if s[2] == t[2]:
        grammar = s[2]
    else:
        raise Exception("grammars of arguments differ")

    # retrieve values
    set_s = s[0]
    set_t = t[0]
    boolean_s = s[1]
    boolean_t = t[1]

    # don't do anything if we already have an ill-formed subsequence
    if (boolean_s is False) or (boolean_t is False):
        return s[0], False, s[2]
    # otherwise proceed with subsequence construction
    else:
        checked_bigrams = set_s.union(set_t)
        new_bigrams = set([])

        for i in checked_bigrams:
            for j in checked_bigrams:
                subsequence = i[0] + j[1]
                if subsequence not in checked_bigrams:
                    if subsequence in grammar:
                        # found a new licit subsequence
                        new_bigrams.add(subsequence)
                    else:
                        # found a new illicit subsequence
                        return s[0], False, s[2]
        return checked_bigrams.union(new_bigrams), True, grammar
```

Listing 8.1: Monoid functions for a strictly piecewise scanner

other words, phonology can be factored into a finite number of local and non-local processes that each correspond to some strictly local or strictly piecewise language (closure under intersection ensures that these can all be combined into two systems of local and non-local processes, respectively).

Jeffrey Heinz has advocated this position in a series of papers (Heinz 2010, Heinz & Idsardi 2011), but only for segmental phonology. Suprasegmental phenomena, in particular stress, are exempt from this claim. And this is a good thing since some suprasegmental phenomena fall outside the purview of SL and SP.

Jeffrey Heinz

2 Suprasegmental Phonology

2.1 Unbounded Stress

One phenomenon we haven't considered so far is stress assignment. Every language comes with a fixed set of rules that determine which syllable in a word gets primary stress and is thus prosodically most prominent. Sometimes primary stress assignment can even resolve lexical ambiguities, such as in English with the noun *'export* and verb *ex'port*.

Most languages use rather simple stress assignment rules, such as *stress the first syllable* (which we will call **[0]** following Python's indexation system) or *stress the antepenultimate* (abbreviated **[-3]**). These simple systems are strictly local. Consider **[0]**, and assume that our alphabet consists of *s* and *u* for stressed and unstressed syllables, respectively. Then the set of (non-empty) strings that are well-formed with respect to **[0]** is exactly the language of the positive strictly 2-local grammar $\{\rtimes s, su, uu, u\ltimes\}$. For **[-3]**, we need a strictly 4-local grammar such that $suu\ltimes$ is the only 4-gram containing $\ltimes$ and no other 4-gram starts with *s*. Since there is an upper bound on the length of syllables (probably around 9, since some languages have very complex syllables like CCCVVCCCC) these grammars can be generalized to our standard string models where each segment represents a sound rather than a syllable. The values for *k* will be high (18 and 36, respectively), but there is a finite upper bound that renders these stress patterns strictly local.

The high values for *k* might actually indicate that phonological structures have two levels, one for segmental phonology and one for suprasemental phonology, with local processes on each level limited to some low *k*. This would explain why stress patterns require a high *k* over segmental structures while all other local segmental processes operate within considerably smaller domains.

But there are some stress patterns where the position of primary stress is more flexible and varies with the structure of the word. Kwakwala, for instance, exhibits what Hayes (1995) calls a *Leftmost Heavy Otherwise Right* (LHOR) pattern:

LHOR Primary stress falls on the leftmost heavy syllable in a word, and if there are no heavy syllables, on the final syllable.

Suppose that this pattern were strictly 2-local, given an alphabet of H and L for unstressed heavy and non-heavy syllables, respectively, and $\acute{H}$ and $\acute{L}$ for their stressed counterparts. Then the corresponding positive SL_2 grammar must contain at least the bigrams $\acute{L}\ltimes$ (stress falls on the final syllable) and $\acute{H}L$ (stress falls on a heavy syllable). But this grammar will also accept strings where stress falls on a heavy syllable that is not leftmost. Since there is no principled upper bound on the distance between two heavy syllables, or the distance between a heavy syllable and the left edge of a word, we cannot fix this issue by moving to an SL_k grammar. So we need to bring in aspects of SP. To this end, we combine the positive SL_2 grammar with a negative SP_2 grammar that consists only of the bigram $H\acute{H}$. Now we correctly capture the fact that the only valid targets for primary stress are the final syllable and the leftmost heavy syllable.

But this is not enough to enforce LHOR: The SL_2 grammar and the SP_2 grammar accept strings where primary stress falls on both the leftmost heavy syllable and the final syllable. In order to rule out this case, we add the bigrams $\acute{H}\acute{H}$ and $\acute{H}\acute{L}$ to the SP_2 grammar, so that a string can only contain one primary stress. Unfortunately this is still insufficient, because both grammars will accept strings containing no stressed syllable at all. Try as we might, there is no way around this due to how these grammars calculate well-formedness: if w is well-formed, then so is every string w' with k-grams(w') $\subseteq$ k-grams(w). For any string w with a stressed syllable, the grammars also accept every string w' that can be built from those k-grams of w that mention no stressed syllable. So SL allows us to tie stress to absolute positions in the string, and SP can identify relative positions as well as put an upper bound on how many stressed syllables a word may contain, but neither can enforce a lower bound in the general case. Our grammars can *allow* specific configurations to occur in a string, but they cannot *force* them to occur if a local alternative is available.

2.2 Culminativity and Threshold Testability

The problematic property of stress patterns like LHOR is that every string has to contain exactly one primary stress, which Heinz (2014) calls *culminativity*. The union of SL and SP can ensure the presence of at most one primary stress but fails to enforce the "at least one" half of the requirement. What we need, then, is a way to enforce lower bounds like this.

The most obvious solution is to pair a grammar G with a function τ that associates with each k-gram g a threshold $\tau(g)$. This threshold establishes the minimum number of times that a k-gram must appear in a string.

Definition 8.4 (Strict Threshold Testability). A *strictly k-local/piecewise m-threshold testable grammar* is a pair $\langle G, \tau \rangle$ such that

- G is a positive strictly k-local/k-piecewise grammar, and
- $\tau : G \rightarrow [0, m]$ is a total function assigning to each k-gram of G a *threshold* less than m.

Given a string w, $|g|_w$ denotes the number of occurrences of k-gram g in w. The language generated by G is $L(G) := \{w \mid |g|_w \geq \tau(g)\}$. A language L is *strictly threshold testable* (STT) iff it is strictly k-local/piecewise m-threshold testable for some fixed k and m.

Assuming an abstract alphabet where we distinguish only stressed and unstressed syllables, culminativity is expressed by the strictly 1-local 1-threshold testable grammar $G := \{\langle s, 1\rangle, \langle u, 0\rangle\}$ (this is a more compact representation that defines G and τ at the same time). Since the 1-gram s has threshold 1, every string must contain at least one primary stress, whereas it need not contain any unstressed syllables. The negative strictly piecewise grammar from the previous section blocks all strings with more than 1 strings, so the intersection of the languages generated by these two grammars contains only strings with exactly one primary stress. So culminativity can be factored into a strictly piecewise grammar and a 1-local 1-threshold testable grammar — in other words, it requires merely the smallest conceivable extension of the strictly local and strictly piecewise languages.

A problem arises, though, if we want both grammars to use the same alphabet and thus distinguish between stressed and unstressed heavy and non-heavy syllables. In this case G needs to be expanded into $G := \left\{\left\langle \acute{\text{H}}, 1\right\rangle, \left\langle \acute{\text{L}}, 1\right\rangle, \langle \text{H}, 0\rangle, \langle \text{L}, 0\rangle\right\}$. However, this grammar force every string to contain at least one stressed heavy syllable and at least one stressed non-heavy syllable. This is clearly not what we want; there isn't even a single string that can satisfy this requirement and at the same time be generated by the strictly piecewise grammar. Culminativity thus is strictly threshold testable, but only over an impoverished alphabet. This is, in a certain sense, the inverse of previous cases where certain languages turned out to be strictly local only if one uses a refined alphabet that provides hidden information. Nonetheless it shows once more that the choice of alphabet is an important factor for determining the complexity of a process or constraint.

Another debatable assumption is that the application domain of phonology is a single word. This is clearly not the case for segmental processes, which can apply across word boundaries. The default assumption is that suprasegmental processes thus should not be limited to a single word, either. But that means that the input may include, among other things, a sequence of two mono-syllabic words, both of which must carry primary stress.The string could thus contain, say, $\rtimes\acute{\text{H}}\ltimes\rtimes\acute{\text{L}}\ltimes$. This is blocked by the SP_2 grammar's bigram $\acute{\text{H}}\acute{\text{L}}$ but can be remedied by moving to a positive SP_4 grammar whose 4-grams may contain two stressed syllabes only if they are separated by $\ltimes\rtimes$. More problematically, the ill-formed $\rtimes\acute{\text{H}}\ltimes\rtimes\text{L}\ltimes$is predicted to be grammatical — neither the SP_4 grammar nor the strictly threshold testable grammar block it.

Our approach to culminativity thus hinges on specific assumptions about the alphabet and the domain within which suprasegmental phonological processes apply. Following our standard methodology of conservatively adding new mechanisms and testing their limits we could carefully map out a space of increasingly more powerful formalisms and see how culminativity fits into these classes depending on our assumptions. This would reveal that culminativity over the expanded alphabet requires the power of so-called *locally testable grammars* (a generalization of strictly local 1-testable grammars), while extending the domain beyond words means moving to the much more powerful class of *star-free grammars*. Instead of exploring all these subtly different formalisms, we will turn our attention to a more principled investigation of the role of hidden structure as it is commonly entertained by phonologists. In particular, how much power can we obtain by distinguishing identical phones via a hidden alphabet?

Relevant literature for Unit 8

add more info

Hayes, Bruce. 1995. *Metrical stress theory*. Chicago: Chicago University Press.

Heinz, Jeffrey. 2010. Learning long-distance phonotactics. *Linguistic Inquiry* 41. 623–661. https://doi.org/10.1162/LING_a_00015. http://dx.doi.org/10.1162/LING_a_00015.

Heinz, Jeffrey. 2014. Culminativity times harmony equals unbounded stress. In Harry van der Hulst (ed.), *Word stress: theoretical and typological issues*, 255–275. Cambridge, UK: Cambridge University Press.

Heinz, Jeffrey & William Idsardi. 2011. Sentence and word complexity. *Science* 333(6040). 295–297.

Martin, Andrew Thomas. 2005. *The effects of distance on lexical bias: sibilant harmony in navajo compounds*. UCLA MA thesis.

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:

$$\begin{array}{ccccc} q & p & p & r & q \\ a & b & b & c & a \end{array} \qquad \begin{array}{ccccc} p & q & r & p & q \\ a & b & b & c & a \end{array} \qquad \begin{array}{ccccc} p & p & p & p & p \\ a & b & b & c & a \end{array}$$

$$\begin{array}{ccccc} q & p & p & q & s \\ a & b & b & c & a \end{array} \qquad \begin{array}{ccccc} q & p & p & & \\ a & b & b & c & a \end{array} \qquad \begin{array}{cccccc} p & p & p & p & p & r \\ a & b & b & c & a & \end{array}$$

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^R_k.

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.

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

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.

$$\begin{array}{cccccc} \text{e} & \text{o} & \text{e} & \text{o} & \text{e} & \text{e} \\ \rtimes & \text{a} & \text{a} & \text{a} & \text{a} & \ltimes \end{array} \qquad \begin{array}{ccccccc} \text{e} & \text{o} & \text{e} & \text{o} & \text{e} & \text{o} & \\ \rtimes & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \ltimes \end{array}$$

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. SL $\subsetneq$ 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 $\ltimes$ 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 $\mathrm{SL}_2^R \nsubseteq \mathrm{SL}$ 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 $\subsetneq \mathrm{SL}^R_2$ ⌟

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^R_2. 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^R_2.

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^R_2: 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:

$$\begin{array}{llll}
\frac{eeee}{a\ltimes\ltimes\ltimes} & \frac{eeee}{aa\ltimes\ltimes} & \frac{eeee}{aaa\ltimes} & \frac{eoee}{abb\ltimes} \\
\frac{eeee}{b\ltimes\ltimes\ltimes} & \frac{eeee}{ba\ltimes\ltimes} & \frac{eeee}{baa\ltimes} & \\
\frac{oeee}{bb\ltimes\ltimes} & \frac{oeee}{bba\ltimes} & & \\
\frac{eoee}{bbb\ltimes} & & &
\end{array}$$

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

$$\begin{array}{ccccc}
\frac{eeee}{aaaa} & \frac{eeeo}{aaab} & \frac{eeoe}{aabb} & \frac{eoee}{abba} & \frac{eoeo}{abbb} \\
\frac{eoeo}{bbbb} & \frac{oeoe}{bbbb} & \frac{eoee}{bbba} & \frac{oeee}{bbaa} & \frac{eeee}{baaa}
\end{array}$$

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*.

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

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

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

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.

$$\begin{array}{cc}
\frac{eoe}{abb} & \frac{oee}{bba} \\
\hline
b & a
\end{array}$$

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):

$$
\begin{array}{ccccc}
\dfrac{\dfrac{eee}{\rtimes\rtimes\rtimes}\;\dfrac{eee}{\rtimes\rtimes a}}{\rtimes\quad a} &
\dfrac{\dfrac{eee}{\rtimes\rtimes a}\;\dfrac{eee}{\rtimes aa}}{a\quad a} &
\dfrac{\dfrac{eee}{\rtimes aa}\;\dfrac{eee}{aaa}}{a\quad a} &
\dfrac{\dfrac{eee}{\rtimes\rtimes\rtimes}\;\dfrac{eee}{\rtimes\rtimes\ltimes}}{\rtimes\quad \ltimes} & \\
\dfrac{\dfrac{eee}{aaa}\;\dfrac{eee}{aa\ltimes}}{a\quad \ltimes} &
\dfrac{\dfrac{eoe}{abb}\;\dfrac{oee}{bb\ltimes}}{b\quad \ltimes} &
\dfrac{\dfrac{eee}{baa}\;\dfrac{eee}{aa\ltimes}}{a\quad \ltimes} &
\dfrac{\dfrac{oee}{bba}\;\dfrac{eee}{ba\ltimes}}{a\quad \ltimes} &
\dfrac{\dfrac{eoe}{bbb}\;\dfrac{oee}{bb\ltimes}}{b\quad \ltimes} \\
\dfrac{\dfrac{eee}{aaa}\;\dfrac{eee}{aaa}}{a\quad a} &
\dfrac{\dfrac{eee}{aaa}\;\dfrac{eeo}{aab}}{a\quad b} &
\dfrac{\dfrac{eeo}{aab}\;\dfrac{eoe}{abb}}{b\quad b} &
\dfrac{\dfrac{eoe}{abb}\;\dfrac{oee}{bba}}{b\quad a} &
\dfrac{\dfrac{eoe}{abb}\;\dfrac{oeo}{bbb}}{b\quad b} \\
\dfrac{\dfrac{eoe}{bbb}\;\dfrac{oeo}{bbb}}{b\quad b} &
\dfrac{\dfrac{oeo}{bbb}\;\dfrac{eoe}{bbb}}{b\quad b} &
\dfrac{\dfrac{eoe}{bbb}\;\dfrac{oee}{bba}}{b\quad a} &
\dfrac{\dfrac{oee}{bba}\;\dfrac{eee}{baa}}{a\quad a} &
\dfrac{\dfrac{eee}{baa}\;\dfrac{eee}{aaa}}{a\quad a}
\end{array}
$$

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.

$$
\begin{array}{ccccccc}
\dfrac{eee}{\rtimes\rtimes\rtimes} & \dfrac{eee}{\rtimes\rtimes a} & \dfrac{eee}{\rtimes aa} & \dfrac{eee}{aaa} & \dfrac{eeo}{aab} & \dfrac{eoe}{abb} & \dfrac{oee}{bb\ltimes} \\
\rtimes & a & a & a & b & b & \ltimes
\end{array}
$$

$$
\begin{array}{ccccccccccc}
\dfrac{eee}{\rtimes\rtimes\rtimes} & \dfrac{eee}{\rtimes\rtimes a} & \dfrac{eee}{\rtimes aa} & \dfrac{eee}{aaa} & \dfrac{eeo}{aab} & \dfrac{eoe}{abb} & \dfrac{oee}{bba} & \dfrac{eee}{baa} & & & \\
\rtimes & a & a & a & b & b & a & a & b & b & \ltimes
\end{array}
$$

$$
\begin{array}{ccccccccccc}
\dfrac{eee}{\rtimes\rtimes\rtimes} & \dfrac{eee}{\rtimes\rtimes a} & \dfrac{eee}{\rtimes aa} & \dfrac{eee}{aaa} & \dfrac{eeo}{aab} & \dfrac{eoe}{abb} & \dfrac{oee}{bba} & \dfrac{eee}{baa} & \dfrac{eee}{aaa} & \dfrac{eeo}{aab} & \\
\rtimes & a & a & a & b & b & a & a & a & b & \ltimes
\end{array}
$$

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 $g := \dfrac{q_1 \cdots q_k}{\sigma_1 \cdots \sigma_k}$ is a k-gram of G_k, then G_2 contains $g' := \dfrac{\phi\rho}{\sigma_{k-1}\sigma_k}$, where $\phi := \dfrac{q_1 \cdots q_{k-1}}{\sigma_1 \cdots \sigma_{k-1}}$ and $\rho := \dfrac{q_2 \cdots q_k}{\sigma_2 \cdots \sigma_k} \in Q_k^{k-1}$. We give an inductive proof that $L(G_2) = L(G_k)$.

Suppose $w \in L(G_k)$, and that $g_k := \dfrac{q_1 \cdots q_k}{\sigma_1 \cdots \sigma_k}$ 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

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

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

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

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

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

iff $g_k \in G_k$. □

2 Regular Languages and Finite-State Automata

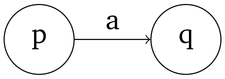
2.1 From Bigrams to Automata

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.

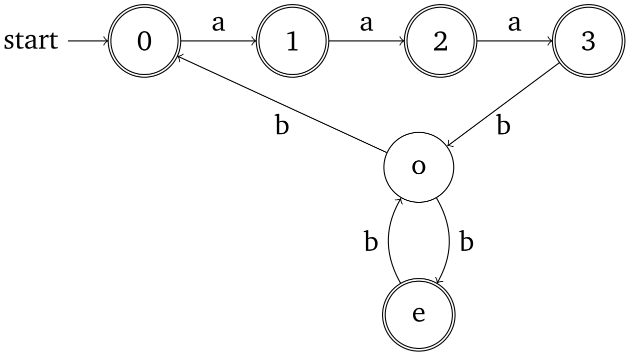
Regular languages are one of the most important classes of string languages in computer science and have been defined in numerous equivalent ways. It simply isn't feasible for us to look at all of them, but most are covered in every decent textbook on formal language theory such as Sipser (2005), Kozen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

$$\begin{array}{cccccc}
0 & 1 & 2 & 3 & o & e \\
 & a & a & a & b & b
\end{array}$$

$$\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & o & 0 & 1 & 2 & & \\
 & a & a & a & b & b & a & a & b & b
\end{array}$$

$$\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & o & 0 & 1 & 2 & 3 & o \\
 & a & a & a & b & b & a & a & a & b
\end{array}$$

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 $\left\langle q, a, q' \right\rangle \in \Delta$.

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

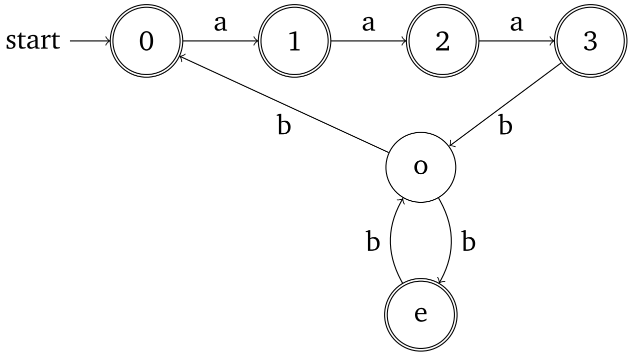
Determinism is an important property of FSAs. A deterministic FSA never has to pick between multiple valid states for a given symbol, at any given node in the string the state is uniquely determined by the symbol of the current node and the state of the preceding node (if it exists). Hence there is only 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.

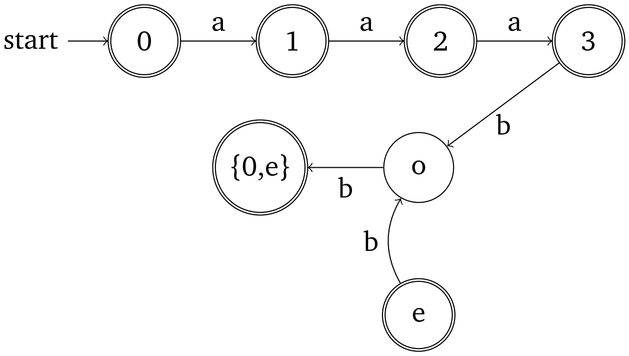
Example 9.5 Determinizing The Non-Deterministic Example Automaton

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

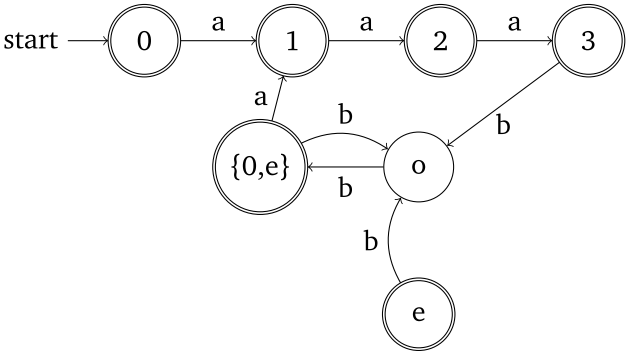


There is only one instance non-determinism, namely the two b-transitions out of state o. In order to make the automaton deterministic, we have to represent the two alternative routes that start at this point into one state. The states that can be reached from o via b are 0 and e, so we create a new state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

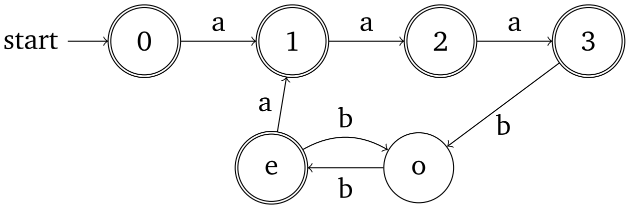
b-transition. All other b-arcs out of o are removed. Note that since $\{0,e\}$ contains at least one final state of the non-deterministic automaton, it is itself a final state.



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.



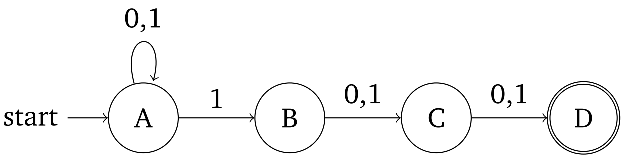
The example above is a very special case in that the determinization does not need increase the number of states and can easily be carried out with intuitive reasoning alone. In general, determinization is a more complex affair and best carried out via a specific conversion algorithm, known as the *powerset 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 $a \in \Sigma$, $\delta_d(q_d, a) := \left\{q' \mid \left\langle q, a, q' \right\rangle \in \Delta, q \in q_d\right\}$.

Example 9.6 Determinization via Powerset Construction

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



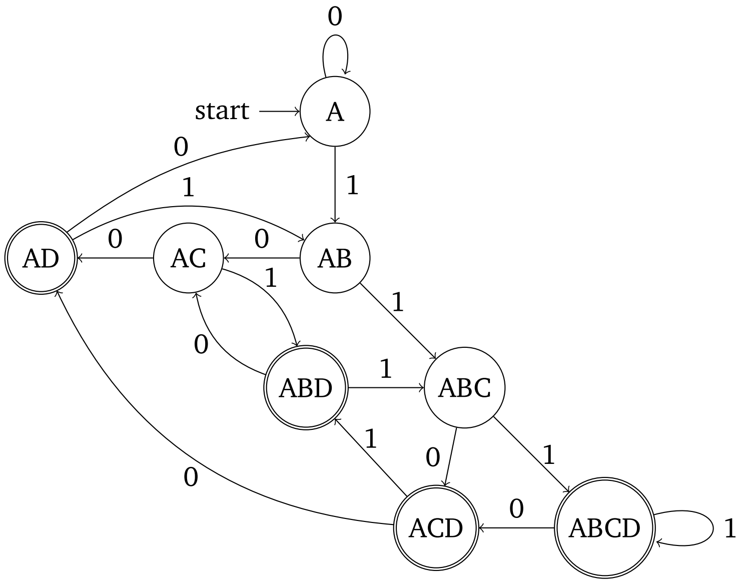
Once again we have only one case of non-determinism, where two arcs that leave A are both labeled 1. This time, however, determinization doubles the size of the state set. First we construct a transition table according to the powerset algorithm. A symbol s in row r and column c means that there is an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

Unit 10

The Power of SPE and OT

We have come a long way since we took our first look at phonology. Starting from a very naive perspective that deliberately ignored most linguistic assumptions and only considered only surface-true generalizations, we kept tweaking our computational model and arrived at the conclusion that most phonological patterns fall within the very weak classes of strictly local and strictly piecewise languages. Suprasegmental patterns required factoring out via culminativity, which can be expressed with 1-threshold testable grammars given a suitable alphabet. The dependence on specific alphabets was worrying though, so that we took a closer look at how much power alphabet refinement can grant us. The answer was rather shocking, as even the strictly 2-local languages see an immense increase in expressivity when they are supplemented with a hidden alphabet. We saw that a hidden alphabet pushes our model all the way up to the regular languages, with no strong empirical evidence that this enormous power is ever needed in phonology.

These findings are particularly troubling because alphabet refinement amounts to positing a more abstract underlying structure, which is an ubiquitous strategy in phonology. This suggests that the standard theories of phonology might actually be too powerful. This claim is difficult to evaluate with our current tools, however, because linguistic theories of phonology do not simply distinguish between well-formed and ill-formed strings, but rather specify a mapping from underlying forms to surface forms. So rather than jumping to conclusions, we should try to give faithful formalizations of these theories. If generative capacity still turns out to be problematic, some reflecting is in order as to why linguists may feel the need for this extra power.

1 Formalizing Rewrite Rules

1.1 Rewrite Rules in SPE

For many decades the dominant theory of phonology was SPE (Chomsky & Halle 1968). According to SPE, phonology is a set of rewrite rules that map underlying representations to surface forms. A rewrite rule takes the form

$$\alpha \to \beta \mid \gamma\_\delta,$$

which means that a substring α that occurs between γ and δ is rewritten as β. For instance, $aa \to \varepsilon \mid b\_bb$ rewrites all instances of *baabb* as *bbb*. The part before the vertical bar, i.e. $\alpha \to \beta$, is the actual rewriting step, whereas γ_δ is the context specification. A rule can apply from left-to-right, right-to-left, or in parallel to all licit target sites.

Example 10.1 Directionality of Rule Application

Consider the rule $a \rightarrow b \mid ab\_ba$, which turns a into b whenever it occurs between ab and ba (this is an abstracted version of the process of full, local assimilation). Depending on how this rule is applied, the string *abababababa* is mapped to different output strings.

directionality	**output string**
left-to-right	*abbbabbbaba*
right-to-left	*ababbbabbba*
parallel	*abbbbbbbbba*

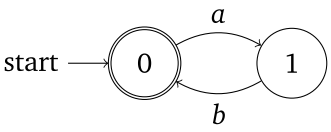
Multiple rules can be conflated into a single rule using round and curly brackets to indicate optionality and non-determinism, respectively. For example, $a \rightarrow \{b, c\} \mid c(d)c\_$ rewrites a as b or c iff it is preceded by cc or cdc. Finally, the collection of conflated rules is linearly ordered, determining in the sequence of rule applications.

As you can see, SPE features a dazzling array of technical tools, and that is actually just the tip of the iceberg. Segments are actually formalized as matrices of binary features, and sets of segments are represented via underspecified matrices. Many of the rewrite rules use this format to indicate the rewriting of segments as the change of feature values, and there is a lot of ancillary notation like α-values to keep rules as short as possible. Finally, segments and strings in the context specification can be subscripted with $^+$ such that w^+ stands for every string in the language w^+. It should be fairly obvious that we do not have the right tools to formalize all of these concepts. Let us start with the most basic notion, then: string rewriting.

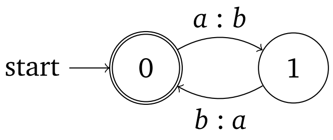
1.2 Finite State Transducers

In the previous chapter we encountered finite state automata (FSAs) as another formalism for talking about strictly 2-local grammars with hidden alphabets. Just like these grammars, an automaton only decides the well-formedness of strings, it is not a mechanism for rewriting a string as another one. However, automata can easily be turned into such a mechanism.

Suppose that we want to map every string of the language $(ab)^*$ to its counterpart where the order of as and bs has been switched. The automaton for $(ab)^*$ is given below.



We want this automaton to also establish a connection between each $(ab)^n$ and $(ba)^n$. An easy way of doing this is to extend the automaton so that it operates on two strings at the same time. Each arc now gets a label $\sigma : \omega$, with the first component indicating the symbol in the first string, and the second component the symbol in the second string. For the current example, we extend a to $a : b$ and b to $b : a$.



Such an extended automaton is called a *finite state transducer* (FST). We can view a transducer as

- verifying whether two strings are related (do both strings take the same path through the transducer?),
- building two strings in parallel (follow some path from an initial state to a final one and keep track of the first/second label of each arc), or
- rewriting the first string as the second one (follow the arcs through transducer that yield the first string and build the second string that is described by these arcs).

Obviously it is this third perspective that is of particular interest to us.

Example 10.2 Three Views of Finite State Transducers

Let us take a quick glance at the behavior of the FST above for the strings *abab* and *baba*. When viewed as a recognizer for a relation between strings, the transducer has to be able to find an identical state assignment for both strings. This is indeed the case.

0	1	0	1	0
	a	b	a	b
	b	a	b	a

But *abab* is not related to *babb*.

0	1	0	1	!
	a	b	a	b
	b	a	b	b

Similarly, the transducer can build *abab* and *baba* in parallel via the sequence of transitions $\langle 0, a : b, 1\rangle$, $\langle 1, b : a, 0\rangle$, $\langle 0, a : b, 1\rangle$, $\langle 1, b : a, 0\rangle$. And we can combine both views by first determining that *abab* is recognized via the sequence $\langle 0, a : b, 1\rangle$, $\langle 1, b : a, 0\rangle$, $\langle 0, a : b, 1\rangle$, $\langle 1, b : a, 0\rangle$, and that the string jointly described by the second components of the transitions is *baba*.

Formally, an FST is an FSA that has been extended with an output alphabet.

Definition 10.1 (Finite State Transducer). A *finite state transducer* (FST) is a 6-tuple $A := \langle \Sigma, \Omega, Q, I, F, \Delta \rangle$, where

- Σ is the input alphabet,

- Ω is the output 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,
- $\Delta \subseteq Q \times \Sigma \times \Omega \times Q$ is a finite set of transition rules.

The FST is *deterministic* iff I is a singleton set and Δ contains no two $\langle p, a, b, q \rangle$ and $\langle p, a, c, r \rangle$ with $q \neq r$ or $b \neq c$. Otherwise it is non-deterministic.

Note that in contrast to non-deterministic automata, non-deterministic transducers cannot always be made deterministic. Characterizing the subclass of non-deterministic transducers that can be determinized is too advanced a topic for us.

An FST does not generate a language, i.e. a set of strings, but a binary relation, i.e. pairs of strings. Such a binary relation between strings is also called a *transduction*, and a transduction that can be computed by an FST is called a *finite state transduction* or a *rational relation*. Note that if we only consider the first component of each pair in the transduction we get the input language, whereas restricting our attention to the second component yields the output language.

So now we have a mechanism for rewriting strings. We do not know yet if it can handle the full range of SPE rewrite rules, but it does provide us with a formal basis that we can explore with our computational techniques.

1.3 Properties of Finite State Transducers

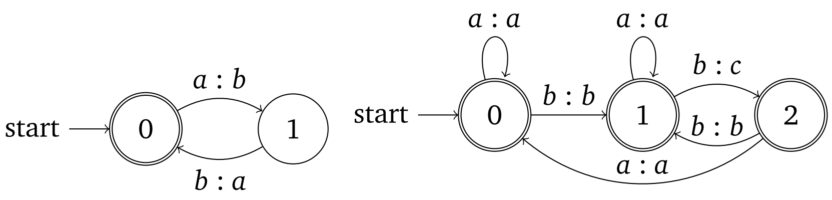
If we are to use FSTs as models of SPE rewrite rules, we will have to be able to combine FSTs, just like SPE combines multiple rewrite rules into a grammar. For SPE, this means in particular being able to use the output of one rewrite rule as the input of the next rewrite rule. So if rule R rewrites the string u as v, and then R' rewrites v as w, then a grammar consisting of those two rules rewrites u as v (assuming R applies before R').

We can do do something very similar with FSTs by constructing their *composition*. Given two FSTs that compute the relations R and R', respectively, their composition computes the relation $R \circ R' := \{\langle u, w \rangle \mid \langle u, v \rangle \in R \text{ and } \langle v, w \rangle \in R'\}$. The construction is very similar to the intersection of FSAs. Both automata are run in parallel, but whenever the first automaton takes an arc that is labeled $a : b$, the second automaton must take an arc whose first component is b.

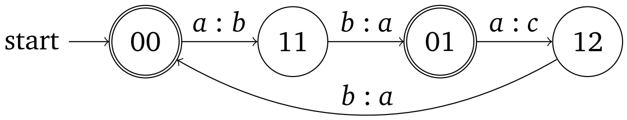
Composition Let $A := \langle Q_A, \Sigma, \Gamma, I_A, F_A, \Delta_A \rangle$ and $B := \langle Q_B, \Gamma, \Omega, I_B, F_B, \Delta_B \rangle$ be two FSTs. Their composition is $A \circ B := \langle Q_A \times Q_B, \Sigma, \Omega, I_A \times I_B, F_A \times F_B, \Delta \rangle$, where Δ is the smallest set containing all $\left\langle (q_a, q_b), \sigma, \omega, (q'_a, q'_b) \right\rangle$ such that $\left\langle q_a, \sigma, \gamma, q'_a \right\rangle \in \Delta_A$ and $\left\langle q_b, \gamma, \omega, q'_b \right\rangle \in \Delta_B$.

Example 10.3 Transducer Composition

Suppose that we want to combine our first example transducer with another transducer that replaces every second *b* by a *c*. So if both transducer are run in sequence, the string *abab* is no longer mapped to *baba* but rather *baca*. Each transducer is shown below.

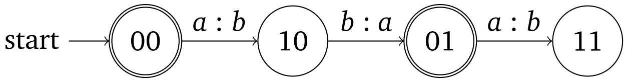


Following the procedure above, we obtain a transducer with 4 useful states.



It is easy to see that this transducer rewrites *abab* as *baca*.

Notice that composition is not commutative. Running the right transducer before the left one maps *abab* to nothing at all because *abac* is not in the domain of the first transducer. You can verify this by looking at their composition; in order to highlight the differences with the previous transducer, the states are still labeled such that their first component represents the state of the left transducer and the second one that of the right transducer.



Closure under composition entails another important property: the image of a regular language L under an FST T is always regular. This is implied by a few simple observations.

Why do we need FST composition for this result? That is to say, why isn't it enough to observe that every FST can be turned into an FSA by dropping the first component of every arc label?

1. If we drop the first component from each arc of the FST T, we get an FSA that defines a regular output language.
2. Every language L can be lifted to a transduction by looking at its identity function $\mathrm{id}(L) := \{\langle w, w\rangle \mid w \in L\}$. If L is regular, this transduction is obtained by taking an FSA for L (which must exist thanks to L being regular) and expanding each arc label a to $a : a$.
3. Since the class of FSTs is closed under composition, $\mathrm{id}(L) \circ T$ is an FST. Notice that the image of Σ^* under $\mathrm{id}(L) \circ T$ is exactly the image of L under T.
4. If Σ^* is the input language for an FST, then the output language consists of all strings that can be derived from some path through the FST. In other words, the output language is exactly the language of the FSA that is obtained by dropping the first component of each arc label.

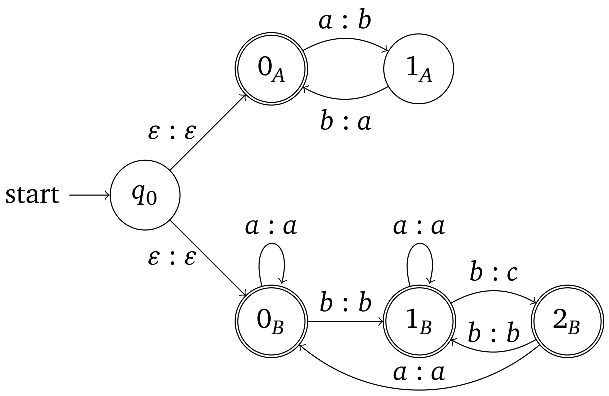
5. It follows that $T(L) = (\text{id}(L) \circ T)(\Sigma^*)$ is a regular language.

Besides composition, it is also of interest to look at the Boolean closure properties.

Union We can just use the automaton construction that adds a single initial state from which ε-transitions take us to the initial states of the transducers. That is to say, $A \cup B := \langle Q_A \cup Q_B \cup \{q_0\}, \Sigma \cup \Gamma, \Gamma \cup \Omega, q_0, F_A \cup F_B, \Delta \rangle$, where $\Delta := \Delta_A \cup \Delta_B \cup \{\langle q_0, \varepsilon, \varepsilon, i \rangle \mid i \in I_A \cup I_B\}$. Keep in mind that the states of Q_A and Q_B must be renamed if the two sets are not disjoint.

Example 10.4 Transducer Union

The union of the two FSTs from the previous example is given below.



Intersection Given two transducers, we can construct their intersection using the exact algorithm for intersection of automata. However, it is not guaranteed that the transduction computed by this new FST is the intersection of the transductions computed by the original two FSTs. This is witnessed by the following counterexample: let R and R' be the regular relations $\{\langle a^n, b^n c^* \rangle \mid n \geq 1\}$ and $\{\langle a^n, b^* c^n \rangle \mid n \geq 1\}$, respectively. Their intersection is $R \cap R' := \{\langle a^n, b^n c^n \rangle \mid n \geq 1\}$. It is well-known that $b^n c^n$ is not regular (a fact we cannot prove yet), but since the output language of an FST is always regular, $R \cap R'$ cannot be a finite state transduction.

(Relative) Complement Non-closure under intersection and closure under union jointly imply non-closure under (relative) complement via De Morgan's law $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

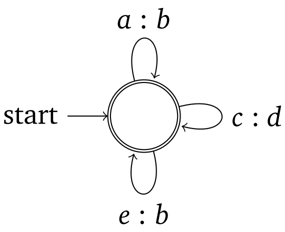
Theorem 10.2. The class of finite state transductions is closed under composition and union, but not intersection or (relative) complement. The class of regular language is closed under finite state transductions. ⌟

Closure under intersection does hold for a proper subclass of finite state transductions, though, namely those that are computed by *ε-free* FSTs: no arc label has ε as its first component ("do not insert new nodes") or as its second component ("do not delete any nodes"). Such transductions only relate strings of equal length, which is why they're called *equal length relations*. An equal length relation can be viewed not just as binary relation, i.e. a set over pairs of strings, but also as sets of strings of pairs of symbols. That is to say, the pair $\langle abba, baab \rangle$ can be viewed as the string $\langle a, b \rangle \langle b, a \rangle \langle b, a \rangle \langle a, b \rangle$ instead. It is fairly easy to prove that equal length relations are regular languages and thus inherit all their closure properties, including closure under intersection and (relative) complement.

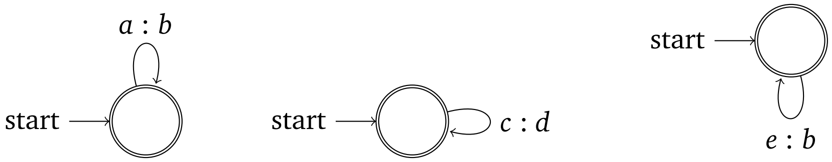
2 The Power of SPE

2.1 SPE Generates All Regular Languages

We already know that every regular language is a projection of some strictly 2-local language. A projection simply replaces every element in the input alphabet Σ by some element of Ω, irrespective of its surrounding. As Σ and Ω are finite, we can view a projection as a single-state FST with a loop labeled $a : b$ iff the projection maps a to b.



But this is just the intersection of three ε-free transducers.



Each one of these transducers corresponds to a rewrite rule $a \rightarrow b$, so an SPE grammar that consists only of the three rewrite rules for the respective FSTs above computes the original projection (careful: we have to ensure Σ and Ω are disjoint, otherwise a rewrite rule may accidentally rewrite some of the output symbols of one of the previous rewrite rules).

Example 10.5 Rewrite Rules for a Projection

Suppose we have a projection π between $\Sigma := \{a, b, c\}$ and $\Omega := \{c, d\}$ that is given by the following table:

input	**output**
a	c
b	c
c	d

This projection maps the string *abbacc* to *ccccdd*.

When converting this projection into an SPE grammar, we first have to make the two alphabets disjoint. This is accomplished by a rewrite rule that rewrite every $\sigma \in \Sigma$ by σ_i.

$$\sigma \rightarrow \sigma_i$$

Now all we have to do is write a rewrite rule for every row in the table.

$$a_i \rightarrow c$$
$$b_i \rightarrow c$$
$$c_i \rightarrow d$$

The table below shows how *abbacc* is rewritten as *ccccdd* by running one rule after another.

rule	output
input	*abbacc*
$\sigma \rightarrow \sigma_i$	$a_ib_ib_ia_ic_ic_i$
$a_i \rightarrow c$	$cb_ib_icc_ic_i$
$b_i \rightarrow c$	$ccccc_ic_i$
$c_i \rightarrow d$	*ccccdd*

If we hadn't rendered the two alphabets disjoint, the output string would have looked very different.

rule	output
input	*abbacc*
$a \rightarrow c$	*cbbccc*
$b \rightarrow c$	*cccccc*
$c \rightarrow d$	*dddddd*

This procedure works for every projection over arbitrary alphabets Σ and Ω, which means that SPE, coupled with a strictly 2-local language of underlying forms, generates all regular languages.

Lemma 10.3. SPE can compute every projection between two arbitrary alphabets. ⌟

Some phonologists might object, though, that using a strictly 2-local language for the set of underlying representations is rather generous and that this set is not as restricted. The *richness of the base* assumption, for instance, contends that every element of Σ^* is a well-formed underlying representation. But this objection is moot since SPE can generate all regular languages even with a rich base.

Richness of the base is an invention of OT and was never entertained during the reign of SPE. However, for practical applications that use rewrite rules in conjunction with a lexicon it is actually more efficient to turn the lexicon into a transduction as described here so that it can be composed with the rewrite rules.

Recall that every language can be turned into a transduction by looking at its identity function, so the strictly 2-local language L itself is just a finite state transduction $\text{id}(L)$. Therefore we can assume that the set of underlying forms includes every element of Σ^*, which is then restricted to L via the transduction $\text{id}(L)$. As long as $\text{id}(L)$ can be translated into a rewrite rule, SPE can restrict the set of underlying forms to this language and then compute its projection, yielding a regular output language.

Lemma 10.4. For every strictly 2-local language L over Σ, there is a sequence of SPE rewrite rules that generates the image of Σ^* under $\mathrm{id}(L)$. ⌟

Proof. We will use rewrite rules to construct a filter that eliminates all strings of Σ^* that contain a bigram g_i that is not a member of 2-grams(w) for any $w \in L$. The first rewrite rule inserts a special symbol $*$ at the beginning of all such strings:

$$\varepsilon \rightarrow * \mid \_(\Sigma^+) \begin{Bmatrix} g_1 \\ \vdots \\ g_n \end{Bmatrix}$$

Since this is a parallel, mandatory rewrite rule, every ungrammatical string now starts with $*$. All we have to do is delete all symbols that occur to the left of a $*$, as well as $*$ itself.

$$\begin{aligned} &\sigma \rightarrow \varepsilon \mid *(\Sigma^+)\_, \text{ for every } \sigma \in \Sigma \\ &* \rightarrow \varepsilon \end{aligned}$$

An SPE grammar that applies these rules from left to right is guaranteed to generate all members of L, and only those. □

Corollary 10.5. Every regular language is generated by some SPE grammar. ⌟

2.2 SPE Generates Only Regular Languages

We just proved that SPE can generate every regular language, but can it generate even more complex languages, languages that are not regular? There are two answers to this question. If we go by the definition of what an SPE grammar looks like, then SPE is shockingly powerful: every recursively enumerable (≈ computable) language is generated by some SPE grammar. That is about as powerful as it gets, and it is due to SPE using both context-sensitive rewriting and deletion of input segments. If every such SPE grammar where a possible natural language phonology, phonological dependencies could involve center embedding, crossing dependences, arbitrary copying, and restrictions that hold only if a word encodes a theorem of first-order logic. So from this perspective, SPE overgenerates to a ludicrous degree and utterly fails to make distinctions between natural and unnatural dependencies.

This view of SPE is at odds with how phonologists think of SPE. While it is true that SPE is considered too powerful, it is commonly assumed to be mostly in the right ballpark. And if we look at the analyses in the literature, it seems unlikely that any of them could be used to generate any of the patterns listed above. This is an important insight that was first pointed out by Johnson (1972) and later formalized by Kaplan & Kay (1994): SPE *as used by linguists* generates only regular languages.

The restriction that phonologists follow without even being aware of it is that the material rewritten by a rule may not be operated on by the very same rule. More precisely, suppose that we have a rewrite rule R that rewrites the substrings $w_{i,j}$ spanning from position i to j in R's input string w. Let $w^R_{i,j}$ be the substring of the output of R that corresponds to $w_{i,j}$. Then R may not rewrite any material of $w^R_{i,j}$.

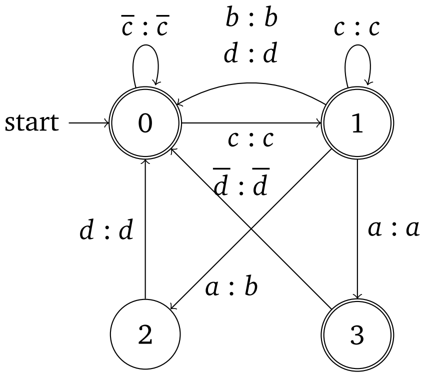
Intuitively, Johnson simply realized that we may think of a rewrite rule as a scanner window that moves through the string from left to right and rewrites material. If the window rewrites $w_{i,j}$ as $w^R_{i,j}$, then $w^R_{i,j}$ winds up immediately to the left of the window and thus cannot be operated on again.

Example 10.6 Recursive and Non-Recursive Rewrite Rule Application

Consider the rewrite rule $\varepsilon \rightarrow ab|\_b$, which inserts ab in front of b. With Johnson's restriction, this rule rewrites the string ab as $a(ab)^+b$. Without it, on the other hand, it also produces strings of the form a^nb^n, among others. See the table for an illustration, with the application domain highlighted in red.

recursive	**non-recursive**
ab	*ab*
aabb	*aabb*
aaabbb	*aababb*
aaaabbbb	*aabababb*

A rule that rewrites a single symbol and applies in parallel at all possible rewriting sites is fairly easy to translate into an FST. Suppose the rewrite rule in question is $a \rightarrow b \mid c\_d$. Then this rule can be thought of as a transducer that rewrites every symbol by itself except for as that are preceded by a c and followed by a d. The FST is depicted below, where $\overline{\sigma}$ is a placeholder for every $\sigma \in \Sigma \setminus \{\sigma\}$.



More sophisticated parallel rules are generalizations of this template, whereas left-to-right and right-to-left application of rules requires special diacritic symbols and is a lot more complicated. Nonetheless Kaplan & Kay (1994) showed successfully that all three types of rule application can be modeled with FSTs. The transduction carried out by a given SPE grammar with ordered rules $R_1 \cdots R_n$ thus is the composition of the FSTs corresponding to these rules. As FSTs are closed under composition, every SPE grammar can be represented by a single FST. And since FSTs generate regular languages, it follows that SPE generates only regular languages.

Theorem 10.6. SPE generates exactly the class of regular languages. ⌟

3 Comparing SPE and OT

3.1 A Formal Definition of OT

In the mid 90s, a new theory became the dominant paradigm in phonological research: Optimality Theory (OT; Prince & Smolensky 2004). OT uses violable ("soft"), ranked constraints in place of SPE's rewrite rules. It has been claimed that this limits its generative capacity in comparison to SPE and also allows it to state generalizations that could not be captured with SPE. We will see that at least the first claim is false.

First, the *generator* Gen maps every input string to its possible *output candidates*, which is usually taken to be every string that can be formed over the same alphabet. Since richness of the base is a standard assumption in OT, the generator computes the transduction $\Sigma^* \times \Sigma^*$. In addition, each constraint c restricts the range of a given transduction τ such that if $i\tau := \{o \mid \langle i, o \rangle \in \tau\}$ is the set of output candidates for input i, then c only keeps those $o \in i\tau$ that are *optimal* with respect to c. Optimality holds of o iff there is no $o' \in i\tau$ such that $c(o') < c(o)$, where $c(o)$ is the number of violations o incurs with respect to c. For example, the constraint $^*ab$ penalizes as being followed by b and is violated 2 times in $abab$, but 3 times in $ababab$. We denote the result of restricting transduction τ via c by $\tau \downharpoonleft c$. The transduction computed by an OT grammar with constraints $c_1, \ldots, c_n$, with c_i higher ranked than c_j for all $i < j$, is $\textsc{Gen} \downharpoonleft c_1 \downharpoonleft \cdots \downharpoonleft c_n$.

Example 10.7 A Small OT Grammar

Suppose that we have an OT grammar with two constraints, one being $^*ab$, the other one $2a!$, which requires at least two as to occur in the string. What are the optimal output candidates for ab under this grammar? This is easily determined via an *OT tableaux*.

ab	$^*ab$	$2a!$
ε		**
a		*
b		**
aa		
ab	*	*
ba		*
bb		**
$\vdots$		
$aaaa$		
$\vdots$		
$aabb$	*	
$\vdots$		
$abab$	**	
$\vdots$		

As you can see, the optimal output candidates are exactly those strings that contain at least 2 as and not a single instance of ab. Note that these optimal output candidates do not violate either constraint, so it does not matter how we rank the two constraints.

Now suppose that we also have a third constraint Id that punishes any deviation from the input string. Then the ranking of constraints does make a difference. For example, if Id is the highest ranked constraint, *ab* is the only optimal output candidate for *ab*. If it the lowest ranked constraint, then the sole optimal output candidate is *aa*.

ab	Id	$^*ab$	$2a!$
ε	**		**
a	*		*
b	*		**
aa	*		
ab		*	*
ba	**		*
bb	*		**
$\vdots$			
aaaa	***		
$\vdots$			
aabb	**	*	
$\vdots$			
abab	**	**	
$\vdots$			

ab	$^*ab$	$2a!$	Id
ε		**	**
a		*	*
b		**	*
aa			*
ab	*	*	
ba		*	**
bb		**	*
$\vdots$			
aaaa			***
$\vdots$			
aabb	*		**
$\vdots$			
abab	**		**
$\vdots$			

The constraints used in the previous example fall into two distinct classes. The first two, $^*ab$ and $2a!$, do not take the input into account. They are called *markedness constraints*. The constraint Id, on the other hand, is a *faithfulness constraint*, as it requires the output to deviate as little as possible from the input.

Definition 10.7 (OT Constraints). An OT constraint is a function $\Sigma^* \times \Sigma^* \times \mathbb{N}$ that maps input-output pairs to natural numbers. A constraint c is a *markedness* constraint iff there are no distinct inputs i and j such that $c(i,o) \neq c(j,o)$. Otherwise it is a *faithfulness* constraint.

Definition 10.8. An OT grammar is a pair $O := \langle C, < \rangle$, where $C := \{c_1, \ldots, c_n\}$ is a finite set of OT constraints and $<$ a linear order over C. The grammar O computes the transduction $\tau :=$ Gen $\downharpoonright c_1 \downharpoonright \cdots \downharpoonright c_n$, where

- $c_i < c_{i+1}$ for all $1 \leq i < n$, and
- $R \downharpoonright c_j := \left\{ \langle i, o \rangle \in R \mid \neg \exists o' \left[\left\langle i, o' \right\rangle \in R \wedge c_j(i, o') < c_j(i, o) \right] \right\}$.

The language generated by O is $O(L) := \bigcup_{i \in \Sigma^*} i\tau$.

3.2 Finite-State OT

It is obvious that the power of an OT grammar depends on its constraints. Without restrictions on what counts as a valid OT constraint, every recursively enumerable language can be generated by an OT grammar. Recall that the same is true for SPE if we do not block rewrite rules from rewriting their own output. In the case of SPE, phonologists intuitively followed that restriction, and likewise the set of commonly entertained OT constraints follows certain restrictions. For instance, there is no constraint in the literature that is satisfied by only those strings whose length is prime, or the Gödel number of a theorem of first-order logic. Nor has anybody proposed anything like a constraint that requires the same number of as and bs to occur in a string. In the following, we will show that the generative capacity of OT is exactly that of SPE as long as the constraints obey certain properties. We start out with a simple fragment of OT (Frank & Satta 1998, Karttunen 1998), which is subsequently extended to a more faithful formalization (Jäger 2002).

Categorical constraints Suppose that all constraints are markedness constraint and each string in Σ^* violates it at most once. In other words, each constraint c defines a language $L(c)$ of well-formed strings. If this language is regular, then the OT grammar generates a regular language.

Theorem 10.9. Let $O := \langle C, < \rangle$ be an OT grammar such that every $c \in C$ is a markedness constraint that defines a regular language. Then τ is a finite state transduction and $O(L)$ is regular. ⌟

Proof. The regularity of $O(L)$ follows immediately from the fact that Σ^* is regular and regular languages are closed under finite state transductions. The following is an inductive proof that τ is a finite-state transduction.

The base case is trivial, as $\textsc{Gen} := \Sigma^* \times \Sigma^*$ is obviously finite-state. Suppose, then, that R is a finite-state transduction. In this case, we have

$$R \downharpoonleft c_i := \begin{cases} R \circ \mathrm{id}(L(c_i)) & \text{if } L(c_i) \neq \emptyset \\ R & \text{otherwise.} \end{cases}$$

For every regular language L, one can test whether it is empty. So we can remove from O all c_i that define an empty language, yielding a compacted grammar O' that computes the same transduction as O. Since $\mathrm{id}(L)$ is a finite-state transduction if L is regular, and since finite-state transductions are closed under composition, the transduction computed by O' — and thus O — is finite-state. □

Testing for emptiness can be omitted by using a more complicated definition instead, see Karttunen (1998) for details.

Note that every regular language L (except the empty language $\emptyset$) can be generated by such a restricted version of OT: we simply limit the set of constraints to a single c with $L(c) = L$. Since the empty language is of no interest to linguists, we can already conclude that OT is at least as powerful as SPE.

FST constraints Virtually no constraint in the literature satisfies the requirement that every string violates it at most once. Fortunately, though, the result can be extended to constraints that have no such upper bound.

Every markedness constraint c defines a relation R_c over Σ^* such that $\langle o, o' \rangle \in R$ iff $c(o) < c(o')$ (by definition we can ignore the input i for markedness constraints).

So the optimal outputs are exactly those for which there is no o' such that $\left\langle o', o \right\rangle \in R_c$. Now suppose that c is the i-th constraint of the OT grammar O, and that $\tau := \textsc{Gen} \downharpoonleft c_1 \downharpoonleft \cdots \downharpoonleft c_{i-1}$. Then we can relativize R_{c_i} to τ by removing all rankings that pertain to an output candidate that has already been eliminated: $rel_c^\tau := R_c \cap (\tau^{-1} \circ \tau)$. The relation $\tau^{-1} \circ \tau$ holds between outputs o and o' iff they are competing candidates for some input i. In other words, τ contains both $\langle i, o \rangle$ and $\left\langle i, o' \right\rangle$.

Definition 10.10 (Rational constraint). A constraint c is *rational* with respect to transduction τ iff there is a finite state transduction S such that $S \cap \tau = rel_c^\tau$.

Intuitively, a constraint c is rational if we can find a finite state transduction S that describes the same ranking over the set of output candidates produced by τ. So the finite state transduction does not have to fully replicate c, the two only need to agree on the remaining candidates. Given such an S, the set of suboptimal candidates is simply $\operatorname{ran}(R \circ S) := \{o \mid \langle i, o \rangle \in R \circ S\}$. The relative complement $\operatorname{ran}(R) \setminus \operatorname{ran}(R \circ S)$ thus is the set of optimal output candidates. Notice that this is guaranteed to be a regular language thanks to the closure properties of finite state transductions and regular languages. Consequently, the identity function over this set is a finite state transduction, so it follows that

$$R \downharpoonleft c_i := R \circ \operatorname{id}(\operatorname{ran}(R) \setminus \operatorname{ran}(R \circ S_i))$$

is a finite state transduction, too.

Theorem 10.11. Let $O := \langle C, < \rangle$ be an OT grammar such that every $c_i \in C$ is a rational markedness constraint with respect to $\textsc{Gen} \downharpoonleft c_1 \downharpoonleft \cdots \downharpoonleft c_{i-1}$. Then τ is a finite state transduction and $O(L)$ is regular. ⌟

Two things should be kept in mind. First, if c_i can be defined in terms of a finite state transducer, then it is always a rational constraint since one valid choice for S in this case is the transitive closure of c_i.

Example 10.8 A Rational OT Constraint

The constraint $^*ab$ can be modeled by a transducer that non-deterministically inserts

- a b after an a, or
- ab at arbitrary points.

For instance a can be rewritten as ab, aba, or $abaab$, but not as ba. Hence we have $a < ab$, $a < aba$, and $a < abaab$, but $a \not< ba$ (note that $aba < abaab$, too).

More importantly, the generalized lenient composition gives the intended result only if optimality is a global property that is independent of the choice of input.

Definition 10.12 (Global Optimality). Given an OT grammar O, optimality is *global* iff it holds for all o that $\langle i, o \rangle \in \tau$ and $\langle j, o \rangle \in \textsc{Gen}$ implies $\langle j, o \rangle \in \tau$.

Global optimality requires for every output candidate that is optimal for input *i* that it is also optimal for every other input *j* that is an output candidate for. This is the case as long as $\text{G}\textsc{en} = \Sigma^* \times \Sigma^*$ and all constraints are markedness constraints, but not if the grammar also contains faithfulness constraints. So this approach still cannot handle faithfulness constraints.

Faithfulness constraints Riggle (2004) develops a very different formalization in terms of *weighted FSTs*. These are FSTs where every arc also has a weight attached it. Riggle uses these weights to encode constraint violations: whenever a string takes a specific path, it may incur a violation of a specific constraint that corresponds to the weight of the arc. Unfortunately we do not have the time to discuss this approach in greater detail.

4 Insights about SPE and OT

We have seen that both SPE and OT can generate every regular language, even if we consider only very weak fragments of the formalisms. For SPE, it suffices to have a strictly 2-local set of underlying forms and a few simple rewrite rules that compute the desired projection. If one wants to also eliminate the set of underlying representations, one needs three additional rules that act as a filter on Σ^*. For OT, a single markedness constraint is sufficient to generate every regular language. A realistic OT grammar with output and faithfulness constraints might even generate non-regular languages, disproving the common assumption that OT is more restricted than SPE. At any rate, both formalisms are more powerful than what is needed for phonology. In fact, they are so powerful that the language class they describe cannot be learned in the limit from positive text, and there is no obvious way to limit their expressivity.

Crucially, though, our notion of power focuses on the generated languages. But it could be argued that a phonological theory must go beyond describing the correct output forms and also has to specify the correct mappings from underlying forms to surface forms. After all, the knowledge that [’Ra:t] is a well-formed string of German is rather useless if the speaker cannot map the word to the lexical entry /Ra:d/ ‘wheel’. Maybe a theory that is expressive enough to describe all these mappings cannot do without a certain amount of power that also allows it to generate arbitrary regular languages. That seems rather odd — why, then, is the range of attested natural language phonologies such a narrowly restricted subset of the regular languages? But this scenario cannot be ruled out as too little is currently known about the computational properties of the mappings from underlying forms to surface forms. Fortunately this subject has gotten a lot more attention in recent years, see e.g. Chandlee (2014). Incomplete as the picture may be at this point, these first forays already provide tentative evidence that SPE and OT overgenerate even with respect to the mappings and that their increased power does not allow for significantly more succinct descriptions of the empirical phenomena.

Relevant literature for Unit 10

add more info

Chandlee, Jane. 2014. *Strictly local phonological processes*. University of Delaware dissertation. http://udspace.udel.edu/handle/19716/13374.

Chomsky, Noam & Morris Halle. 1968. *The sound pattern of English*. New York: Evanston.

Frank, Robert & Giorgio Satta. 1998. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics* 24. 307–315. http://www.aclweb.org/anthology/J98-2006.

Jäger, Gerhard. 2002. Gradient constraints in finite state OT: the unidirectional and the bidirectional case. In I. Kaufmann & B. Stiebels (eds.), *More than words. A festschrift for Dieter Wunderlich*, 299–325. Berlin: Akademie Verlag.

Johnson, C. Douglas. 1972. *Formal aspects of phonological description*. The Hague: Mouton.

Kaplan, Ronald M. & Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20(3). 331–378. http://www.aclweb.org/anthology/J94-3001.pdf.

Karttunen, Lauri. 1998. The proper treatment of optimality in computational phonology. In *Proceedings of the international workshop on finite state methods in natural language processing*, 1–12. Stroudsburg, PA: Association for Computational Linguistics. http://www.aclweb.org/anthology/W98-1301.

Prince, Alan & Paul Smolensky. 2004. *Optimality theory: constraint interaction in generative grammar*. Oxford: Blackwell.

Riggle, Jason. 2004. *Generation, recognition, and learning in finite-state optimality theory*. University of California, Los Angeles dissertation.

Part II

Morphology and Syntax

Unit 11

Beyond Phonology

At the very beginning of this course, we split language into the subareas phonetics, phonology, morphology, syntax, semantics, and pragmatics. Now that we have concluded our investigation of phonology, it is time to move on to a new area. We will start with morphology, which is very similar to phonology on a computational level, before moving on syntax, which will occupy us for the rest of the course.

1 Morphology

1.1 Morphology in Natural Language

Morphology is concerned with word formation, which can be further divided into two distinct subsystems. *Inflectional* morphology regulates the overt marking of agreement with other words in a sentence. For example, English has a very limited form of person and number agreement between the subject and the finite verb of a sentence.

(1) a. The man like**s** the children.
b. * The man like the children.
c. The men like the children.
d. * The men like**s** the children.

Other languages have a much richer system of inflectional morphology. In Icelandic, for example, adjectives agree with the noun they modify in number, gender, and case. In addition, they also agree in definiteness with the determiner. And Icelandic is still relatively tame from a typological perspective, thanks to its restriction to only two numbers, three person, three genders, and four cases. Other languages have over ten distinct genders, three numbers (singular-dual/paucal-plural), possibly four distinct persons, and a myriad of cases.

Derivational morphology consists of the rules for deriving new words from existing ones. This includes compounding — the adjective *smart* and the noun *phone* combine into the compound *smartphone*, which didn't exist until a few years ago — as well as changes in POS, such as turning the noun *sea lion* into the verb *to sea lion* (another neologism that refers to polite yet intrusive attempts to engage somebody in a debate). Derivational morphology thus provides a system for dynamically extending the lexicon of a language.

The term *to sea lion* was coined in 2014 following a popular Wondermark comic strip: http://wondermark.com/1k62/.

Note that the distinction between inflectional and derivational morphology has nothing to say about how these processes are marked. Number agreement in German, for instance, can be indicated via a suffix (*Frau* 'woman', *Frauen* 'women'), via a sound change (*Laden* 'store' and *Läden* 'stores'), a suffix and a sound change (*Haus* 'house', *Häuser* 'houses'), or neither (*Schlüssel* 'key' and 'keys'). Other languages may use prefixes, circumfixes, or infixes instead. The same goes for derivational morphology. English often uses a suffix (*quick* and *quickly*, *the haste* and *to hasten*), sometimes a shift in stress (*the export* and *to export*), and sometimes no marker at all (*the anger* and *to anger*). In addition, compounding has no overt marking in many languages beyond the adjacency of the words, while many languages use *reduplication* to create new words (Marshallese *kagir* 'belt' and *kagirgir* 'to wear a belt'). In sum, morphological processes can be realized overtly via affixation, compounding, reduplication, or some phonological process.

The interaction of morphology and phonology is symmetric, though, in the sense that morphological structure can also suspend or trigger phonological constrains and processes. For instance, consonant clusters can have greater complexity if they span a morpheme boundary. This can be seen in German, where /tstpR/ cannot occur in any monomorphemic word but is perfectly acceptable in the compound *Arztpraxis* 'doctor's office'. The interaction of morphology and phonology is also known as *morphophonology* or, slightly shortened, *morphonology*. The close interaction of those two domains, with morphology sometimes relying phonological processes and phonology being sensitive to morphological information, means that the two are often treated by one and the same mechanism in real-life applications.

1.2 Two-Level Morphology

Why is this a better strategy than simply listing every word with all its inflected forms?

Just like phonology, morphology has been analyzed in terms of rewrite rules for the largest part of recent history. To the best of my knowledge, morphologists instinctively followed the same ban against a rule rewriting its own output that guaranteed the regularity of phonology. This makes it very likely that morphology, or at least a large part of it, is regular, too. So the finite state methods we used for phonology can be used just as well for morphology, which makes it possible to have one big rewrite grammar that intersperses phonological and morphological rewriting to capture the interaction of the two. This collection of rewrite rules can then be converted into a single FST thanks to the closure of finite state transductions under composition.

Contrary to what one might expect, though, this is not quite the approach taken in most industrial applications. This is mostly for historical reasons. Composing a large number of transducers, many of which may have dozens of states, simply wasn't computationally feasible before the 90s. In addition, people had not figured out yet how such a transducer can be used efficiently for morphological analysis, rather than generation.

Suppose you have a phonological surface form *s* for a word that sounds like *wipeboard*. Then *s* could be simply a compound of *wipe* and *board*, or a compound of *white* and *board* where /t/ was replaced by /p/ due to the following /b/. Now if you take the FST for your rewrite grammar and construct its inverse, which maps surface forms to underlying forms, it can return either option as a possible underlying form (since the FST is non-deterministic, even getting this set of all possible underlying forms is not trivial). In order to determine the correct underlying form, you then have to lookup each form in the lexicon, where you will eventually find an entry for *white board* but none for *wipe board*. Keep in mind that a lexicon can contain hundreds of thousands of entries, so this search may take a long time unless one uses a hashtable,

Marker	**Process type**	**Example**	**Language**
no marking	derivational	*the anger — to anger*	
	inflectional	*I eat — you eat*	
		this sheep — these sheep	
prefix	derivational	*do — undo*	
	inflectional	*i-ausipu* (3SG-put.down)	Nuaulu
suffix	derivational	*do — doable*	
		quick — quickly	
	inflectional	*I eat — he eats*	
		dog — dogs	
infix	derivational	*unbelievable — un-fucking-believable*	English (marginal)
		Óscar — Osquítar (diminutive of ‘Oscar’)	Nicaraguan Spanish
	inflectional	*hafal* ‘celebrate’ — *h-t-afal* ‘celebrated’	Arabic
		(person and gender affixes omitted)	
circumfix	derivational	*adil* ‘fair’ — *ke-adil-an* ‘fairness’	Malay
	inflectional	*kauf-en* ‘to buy’ — *ge-kauf-t* ‘bought.PSTPART’	German
sound change	derivational	*biegen* ‘to bend’ — *Bogen* ‘bow, arch’	German (not productive)
	inflectional	*Laden* ‘store’ — *Läden* ‘stores’	German
		woman — women	English (not productive)
prosodic change	derivational	*the export — to export*	
	inflectional	*standen* ‘stand.PST’ — *stünden* ‘stand.SUBJ’	German
reduplication	derivational	*to like — to like-like* ‘love’	
		“Rules, schmules!”	American English
		kagir ‘belt’ — *kagir-gir* ‘to wear a belt’	Marshallese
	inflectional	*kanak* ‘child’ — *kanak kanak* ‘children’	Indonesian

Table 11.1: Overview of morphological marking strategies

which wasn't feasible back then due to memory limitations.

Nowadays the solution is obvious: take the identity function over the lexicon and compose it with the FST for the grammar. If this composed FST is run in reverse, it only outputs items in the lexicon, so that no search is needed. But this simple trick wasn't worked out until the mid 90s, and instead *two-level morphology* was developed as a more practical tool for morphological analysis (Koskenniemi 1983).

From a formal perspective, two-level morphology is a very simple modification of the rewrite paradigm. Rather than applying rewrite rules sequentially, one after another, they are all applied in parallel. Consequently, the grammar isn't a cascade of FSTs but runs all FSTs in parallel instead. The overall transduction is the intersection of these FSTs. Recall that the intersection of arbitrary FSTs is not guaranteed to be an FST. However, the class of ε-free FSTs is closed under intersection, and two-level morphology exploits this fact by using a special symbol 0 to indicate unpronounced nodes. That way, deletion of a symbol σ amounts to relabeling it as 0, and insertion of σ is emulated by relabeling 0 as σ.

The process of pluralizing *index* to *indices*, for instance, is modeled as first composing the stem *index* and the plural marker +00, and this string *index*+00 is rewritten as *indic*0*es*, which is pronounced *indices*. Formally, the transduction contains the pair $\langle index + 00, indic0es \rangle$, and since both strings have the same length, we are dealing with an equal length relation, which instead can be viewed as a string of pair symbols

$$\binom{i}{i}\binom{n}{n}\binom{d}{d}\binom{e}{i}\binom{x}{c}\binom{+}{0}\binom{0}{e}\binom{0}{s}$$

This shift in perspective turns the relation into a regular language, and closure under intersection holds. A rather inelegant trick, but it gets the job done.

As is implied by the name, two-level morphology posits only two levels, one for underlying forms, one for surface forms. The two forms are padded with zeros if necessary to ensure they have the same length, wherefore they can always be analyzed as a single string of pair symbols as above. The rewrite rules operate over such pair symbols $u : s$, where u is the underlying segment and s the surface segment. The rewrite rules are of the form $u : s \diamond \alpha\_\beta$, where α and β are strings over pair symbols (and may also include SPE-style notation like brackets for optionality and + for iteration). The symbol $\diamond$ is a placeholder for two different types of rewrite arrows: $\Rightarrow$ and $\Leftarrow$. If $\diamond$ is replaced by $\Rightarrow$, then the rule states $u : s$ can occur only in the specified context. This requirement is weakened with $\Leftarrow$, which states that if both $u : s$ occurs in the specified context and the surface form has an s, then the underlying form must have u. This is also called *surface coercion*. Sometimes $\Leftrightarrow$ is used as a combination of the two to express that $u : s$ occurs only in the given context and no distinct form $u' : s$ may occur there.

With a little bit of ingenuity, each rewrite rules can be converted into a regular language of pair symbols, and the whole grammar is simply the intersection of these regular languages. Analyzing a surface form is tantamount to finding a pair string whose second component matches the surface form, whereas generation instead looks for a pair string whose first component matches the desired surface form. Since a language of pair strings can be converted into an ε-free FST, this search is simply a matter of running the FST over the underlying form, or its reverse over the surface form.

In sum, two-level morphology may look very different from SPE or OT, yet it is just another way of defining finite state transductions. While it can generate all regular languages, just like SPE and OT, it is weaker than those two in the sense that it cannot handle deletion and insertion in an elegant way and must instead rely on padding out strings via the special symbol 0.

1.3 Complexity of Morphology

Two-level morphology has been successfully applied to a variety of typologically diverse languages, including English, Finnish, Turkish, and Japanese. Just like the overwhelming descriptive success of SPE is strong evidence that phonology is at most regular, the wide usage of two-level morphology suggests that morphology is regular, too.

This is actually not all that surprising, considering the local and finitely bounded nature of most morphological processes. In almost all cases it is sufficient to know the modified stem and which morphological process took place most recently. In addition, some morphemes can only be instantiated exactly once. All of these things fall under the purview of regular languages.

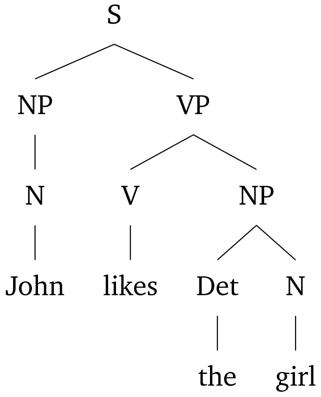
Only two aspects of morphology might be problematic. First, circumfixion requires the presence of both a prefix and a suffix of a specific type. This is illustrated by the German past participle, which consists of the stem of the verb and the circumfix *ge- -t* as in *ge-kauf-t* 'bought'. If this process were unbounded, then German would contain words of the form *ge*^{n}*-kauf-t*^{n}, but not, say, *ge-ge-kauft-t-t-t*. In our discussion of the complexity of syntax later on we will see that such a pattern is not regular. Intuitively, that's because an FST generating this pattern would have to first inserts *n* instances of *ge-*, followed by *n* instances of *-t*, but since the FST has only a finite number of states it can't keep track of the exact number *n* past a certain threshold and may end up inserting too few or too many suffixes. As you might expect, though, German past participle formation is not an unbounded process (probably because additional circumfixes would serve no function). To the best of my knowledge, unbounded circumfixion is universally unattested, lending strong support to the hypothesis that morphology is regular and thus incapable of generating such patterns.

The only remaining challenge to the regularity hypothesis is reduplication. If there is no upper bound on the size of the reduplicant, we run into a similar memory problem for the FST as above: with *n* states, the FST can memorize only a fixed amount of information, so if the material that should be reduplicated is so long that it does not fully fit in the state memory, the FST cannot insert an exact copy. Reduplication data has proven very difficult to analyze, and at this point it is not clear whether there are any cases of unbounded reduplication where the reduplicant must be an exact copy. In the cases where reduplication seems to be unbounded, it usually interacts with phonology in various ways that make it difficult to discern what the morphological well-formedness criterion is. Putting aside reduplication, though, all of known morphology seems to fall within the class of finite state transductions. And just like in phonology, the overwhelming majority of processes do not come close to exploiting the full power FSTs.

2 Syntax

2.1 A Weak Argument That Syntax is Not Regular

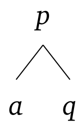
Syntax regulates the well-formedness of sentences, which includes basic factors like word-order but also much more arcane properties such as the distribution of so-called negative polarity items (*ever*) and anaphors (*himself*, *herself*, *itself*). It is usually believed to be more complex than phonology and morphology. For example, the basic data structure is assumed to be trees rather than strings, so that a simple sentence like *John likes Mary* actually involves a lot of hidden structure:



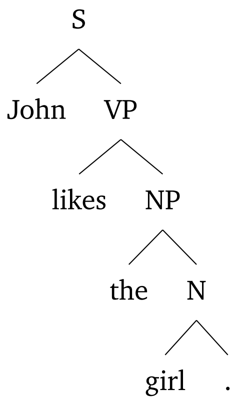
Trees of this form are generated by phrase structure grammars, which are finite sets of rewrite rules without context specifications. The tree above is generated by the grammar below:

S	→	NP VP	Det	→	the
NP	→	(Det) N	N	→	John | girl
VP	→	V NP	V	→	likes

At first sight it seems that this shift to trees — if it is indeed called for — renders our current tools useless because they generate strings, not trees. But this is not quite correct. Consider the transition $\langle p, a, q \rangle$ some FSA. We can represent this as a tree with p as the root and a and q as left and right sibling, respectively.



And every such tree corresponds to a rewrite rule $p \to a\ q$. So every FSA corresponds to a phrase structure grammar where every node has exactly two daughters, the first of which is a symbol from the alphabet whereas the second one is a state symbol. That is not enough to produce the tree above, but it allows us to generate a very similar version (the dot indicates the final state).

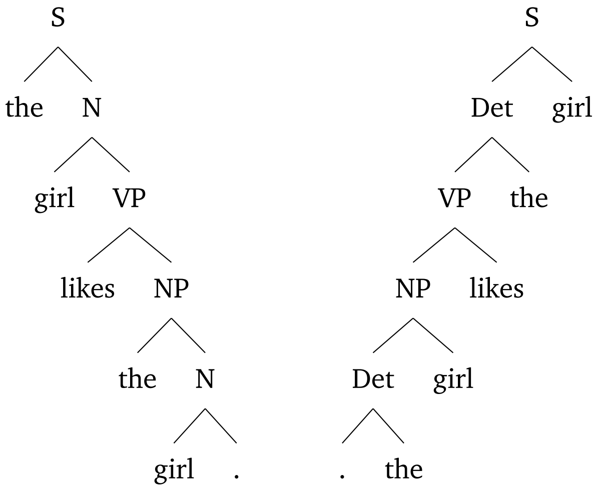


Every FSA can also be converted into a grammar that generates strictly left-branching trees. What does this translation look like?

Such a tree is called *strictly right-branching*, and its mirror image would be *strictly left-branching*. This terminology also applies to phrase structure grammars that only generate trees of this kind.

An argument that FSAs are insufficient for syntax cannot simply invoke the fact that syntax uses trees. Every FSA can be converted into a grammar that generates trees, albeit strictly right-/left-branching ones. Consequently, a tree-based argument has to show that these types of trees are unsuitable for natural language. That is a lot more difficult than one would expect.

For example, one may object that both the strictly right-branching and the strictly left-branching tree do not represent the true structure of *The girl likes the girl*.



But how does one determine what the true structure should be? In contrast to phonology, where we could check the generated strings against the observed surface forms, we have no direct access to the tree structure of a sentence, assuming it even exists. All we have is indirect evidence. In the case at hand, the two instances of *the girl* have different structures but seem to exhibit very similar behavior. For example, both can be coordinated with another noun phrase.

(2) a. The girl likes the girl.
b. The girl and the boy like the girl.
c. The girl likes the girl and the boy.
d. The girl and the boy like the girl and the boy.

Linguists have an enormous battery of tests to probe the *constituency* of a sentence, i.e. how words combine into subtrees. It is important to keep in mind, though, that these tests are theory-laden, starting with the very basic assumption that if two substrings show similar behavior, this is indicative of a structural similarity between the subtrees. Often it is also implicitly assumed that constituency tests identify the *unique* structure of a substring, rather than one of several varieties. There are licit scientific assumptions, but they invariably weaken any results obtained this way — if the assumptions are rejected or shown faulty, then the array of findings that build on them, no matter how impressive, is at risk of collapsing like a house of cards. A proof that does not rely on any of these assumptions is more reliable and general and insightful, and thus to preferred.

For example, many debates about English possessive constructions focus on whether the possessor or the possessee is a sibling of the possessive marker *'s*, when the simplest account is simply that both options are licit.

Instead of focussing on how well regular methods may describe the tree structure of sentences, it is a lot simpler to verify whether the string languages are even regular. If the set of well-formed English sentences, understood as strings of words, is not regular, then obviously no FSA can generate this set and it follows immediately that FSAs cannot produce the necessary tree structures, either.

2.2 A String-Based Proof that Syntax is Not Regular

In our discussion of morphology it was mentioned that a pattern like *ge*^{n}*-kauft-t*^{n}, where n instances of *ge* must be followed by n instances of *t*, is not regular. While it is unclear that such a pattern is ever instantiated in morphology, it is actually fairly easy to construct for syntax.

(3) a. That John surprised me annoyed me.
b. That that John surprised me surprised me annoyed me.
⋮
c. That^n John (surprised me)^n annoyed me.

The pattern above relies on the fact that each *that* introduces a sentential subject, and since each one of them must have its finite verb, the number of occurrences of *that* must be exactly the number of finite verbs, *modulo* the finite verb in the highest clause. The sentence is fairly unnatural, but the basic idea can easily be generalized to other constructions.

(4) a. The cheese was rotten.
b. The cheese that the mouse ate was rotten.
c. The cheese that the mouse that the cat chased ate was rotten.
d. The cheese that the mouse that the cat that the dog dislikes chased ate was rotten.
⋮
e. The cheese (that the *noun*)^n (*transitive verb*)^n was rotten.

Quite generally, any kind of *center embedding construction* will give rise to to these kinds of number matching conditions $a^n b^n$.

We still have to proof, though, that $a^n b^n$ and related languages are not regular. The intuition that FSAs do not have enough memory homes in on the essential problem, but it does not constitute a proof. We will flash it out via a so-called *pumping lemma*. First, suppose that A is an FSA with k states that generates an infinite language $L(A)$. Then for any $w \in L(A)$ with $|w| > k$ it must be the case that two nodes m and n of w

are assigned the same state q by A. But then A must contain a loop that leads from q back to q. More precisely, let $w[m:n]$ be the substring of w that spans from m to n. Then $w[m:n]$ describes a path through A from q to q. Clearly this path can be taken multiple times. Each instance of following this loop is called a *pump* p. Hence if $w := x \cdot w[m:n] \cdot y$ belongs to $L(A)$, so does every *pumped string* $x \cdot w[m:n]^p \cdot y$ for all $p \geq 1$.

Theorem 11.1 (Regular Pumping Lemma). If a language L is regular, then there exists an integer $k \geq 1$ such that every $w \in L$ of length at least k has a decomposition $w := x \cdot z \cdot y$, where

- $|z| \geq 1$,
- $|x| + |y| < k$,
- $xz^p y \in L$, $p \geq 1$. ┘

Proof. If L is finite, the claim is trivial. For L infinite, it follows from the representation via FSAs as discussed above. □

Note that the pumping lemma is a necessary condition for a language to be regular, but not a sufficient one. That is to say, there are non-regular languages that satisfy the pumping lemma. Consequently, the pumping lemma cannot be used to prove that a language is regular, but failure of the pumping lemma does prove non-regularity.

With the pumping lemma, it is fairly easy to show that $a^n b^n$ is not regular. All we have to do is consider all the possible decompositions of $a^k b^k$ into xzy and show that we never end up with the right kind of pump, irrespective of the value for k. First, if x contains at least one b, then z can only contain bs and thus pumping z introduces new bs but not new as, so we got from $a^k b^k$ to $a^k b^m$, $m > n$, which is not in $a^n b^n$. An analogous argument holds if y contains at least one a. So z must be some string of the form $a^i b^j$. But then $z^2 = a^i b^j a^i b^j$, and no string with z^2 as a substring can be contained in $a^n b^n$. It follows that $a^n b^n$ is not regular.

Showing that $a^n b^n$ is not regular is not enough to demonstrate that English is not regular, though. We have to translate the English patterns into the formal language $a^n b^n$. To this end, we let τ be a finite state transduction that deletes the first two words *the cheese*, maps each instance of *that the noun* to a, each instance of a transitive verb to b, and deletes the last two words *was rotten*. This turns the language *the cheese (that the noun)^n (transitive verb)^n was rotten* into $a^n b^n$. As regular languages are closed under finite-state transductions, $a^n b^n$ is regular if the fragment of English is regular. Since the former is not, the latter is not either.

It is tempting to stop at this point and conclude that English is not regular, but this would be a grave (and unfortunately very common) mistake. Showing that a fragment of a language has a certain complexity does not imply that the whole language is of the same complexity. For example, Σ^* is strictly 1-local even though it contains the fragment $(aa)^+$, which is regular. The whole language can be much simpler than a given fragment. In the case at hand, English may actually be regular if we can also have sentences of the form *the cheese that the mouse was rotten* or *the cheese that the mouse chased ate was rotten*, where the number of NPs and transitive verbs no longer match. The crucial property — which is implicit in our claim that English center embedding is captured by the pattern $a^n b^n$ — is that all strings that deviate from this pattern in a specific way are ungrammatical (obviously there are strings that deviate in some other way yet are well-formed, e.g. *John likes Mary*). Directly defining this set of related yet ungrammatical strings is tricky, so often it is more efficient to use closure properties to extend the proof to the entire language.

In the case at hand, we construct a fragment of English that contains all relevant center embedding constructions via regular intersection. Evidently the language given by the pattern *the cheese (that the noun)** *(transitive verb)** *was rotten* is regular, and its intersection with English is the language *the cheese (that the noun)*^{n} *(transitive verb)*^{n} *was rotten*. But we already know that this language is not regular and as regular languages are closed under intersection, it follows that English cannot be regular. And since English is a natural language, we can finally conclude that natural language syntax is not adequately modeled by FSAs and FSTs.

2.3 Myhill-Nerode Characterization of Regular Languages

The pumping lemma has the disadvantage that it may yield false positives in the sense that some non-regular languages may satisfy it. A much stronger result is the *Myhill-Nerode theorem*, which gives a full characterization of the regular languages.

Suppose that A is a deterministic FSA with state set Q that recognizes L. Then we can identify with every state $q \in Q$ a set *tails*(q) of *good tails* t such that t describes a path from q to a final state of A. In the same vein, we can identify the set *heads*(q) of *good heads* that describe a path from the initial state of A to q. Concatenating a good head of q with one of its good tails produces a string of L. In fact, L is exactly $\bigcup_{q \in Q} \mathit{heads}(q) \cdot \mathit{tails}(q)$. The Myhill-Nerode theorem generalizes this idea so that it can be stated over strings without referencing automata or states.

Given two strings u and v, we write $u \equiv_L v$ iff u and v have the same good tails in L:

$$u \equiv_L v \Leftrightarrow \{x \mid u \cdot x \in L\} = \{x \mid v \cdot x \in L\}$$

It is easy to see that $\equiv_L$ is an equivalence relation.

Background Equivalence relation

An equivalence relation is a binary relation R over a set S that satisfies three conditions:

reflexive for all $x \in S$, $x \, R \, x$

symmetry for all $x, y \in S$, $x \, R \, y$ implies $y \, R \, x$,

transitivity for all $x, y, z \in S$, $x \, R \, y$ and $y \, R \, z$ jointly imply $x \, R \, z$.

The equivalence relation partitions L into equivalence classes of strings that have the same good tails. But if these strings all have the same good tails, we can paraphrase this in FSA-terms: they all lead to the same state. So each equivalence class is the (possibly infinite) set of paths that lead to a state q. This basic insight is enough to show that a language is regular iff $\equiv_L$ induces a finite number of equivalence classes. One also says that $\equiv_L$ has *finite index*.

Definition 11.2 (Index). Let R be an equivalence relation over set S, and let $[a] = \{b \mid a \; R \; b\}$. The *index* of R is $|\{[a] \mid a \in S\}|$.

Theorem 11.3 (Myhill-Nerode). A string language L is regular iff the corresponding equivalence relation $\equiv_L$ has finite index. ⌟

Proof. If L is regular, then we have $u \equiv_L v$ if they lead to the same state in some deterministic FSA A. Hence, if A has k states, the index of $\equiv_L$ must be less than k and thus finite.

In the other direction, we use $\equiv_L$ to construct a deterministic FSA A. First, pick one string w from each equivalence class of $\equiv_L$. The state set of A contains q_w for every such w. We mark q_w as final iff $w \in L$, and q_w is initial iff $[w]$ contains ε. Finally, $\delta(q_w, a) = q_z$ iff there are $u \in [w]$ and $v \in [z]$ such that $v = u \cdot a$. Verifying that A is deterministic and recognizes L is left as an exercise to the reader. □

The Myhill-Nerode theorem furnishes another way of showing that $a^n b^n$ is not regular, one that is much closer to the original intuition that an FSA does not have enough memory for this pattern. For every string a^n, its set of good tails is given by $\left\{a^i b^j \mid i \leq 0, j = n + i\right\}$. Hence we have $a^m \equiv_L a^n$ iff $m = n$, i.e. each substring of as has its own equivalence class. Consequently, $\equiv_L$ cannot be of finite index.

So now we have yet another characterization of regular languages, but this one is particularly noteworthy because it completely abstracts away from grammars, automata and such devices to instead gives a direct characterization of regular languages. In that respect, it is very similar to our characterization of the strictly local languages via substring substitution closure.

Corollary 11.4. Let L be a string language. Then the following claims are equivalent:

1. L is generated by a refined strictly 2-local grammar,
2. L is a projection of a strictly 2-local language,
3. L is recognized by a finite-state automaton,
4. L is the output language of a finite-state transducer,
5. L is generated by a strictly right-/left-branching grammar,
6. $\equiv_L$ has finite index,
7. L is regular. ⌟

Relevant literature for Unit 11

add more info

Koskenniemi, Kimmo. 1983. *Two-level Morphology: A General Computational Model For Word-Form Recognition and Production*. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

Unit 12

Moving From Strings to Trees

We have been able to show that English is not a regular language due to the structural complexity of center embedding. This leaves us with two choices: we may either stick with finite-state models and claim that center embedding does not pose a challenge in practice, or we can once again move to a more expressive model. The first choice is a valid option in a variety of applications but requires some major sacrifices. We will see that these sacrifices cannot be reconciled with our project of exploring the properties of language from a computational perspective, and thus we will take a hint from linguists and expand our model by moving from strings to trees.

1 Finite-State Methods and the Role of Unboundedness

As mentioned last time, the reason that center-embedding patterns — abstractly represented as $a^n b^n$ — are beyond the capabilities of finite-state methods is that there is no upper bound on the depth of embedding. An FSA has to memorize the exact number of as in order to ensure that the same number of bs follow, but since an FSA has a fixed number k of states, it can only partition strings into $k + 1$ distinct equivalence classes (one equivalence class for each state, plus one for strings for which there is no defined transition). Hence some strings with distinct numbers of as must wind up in the same equivalence class and thus can be followed by the same number of bs according to the automaton, which is clearly not the case for the language $a^n b^n$.

This argument is mathematically flawless, but it faces a major empirical challenge: center-embedding is **not** unbounded in the data we have access to. No speaker spontaneously produces a sentence with ten levels of center-embedding, and even if we found a speaker with a propensity for convoluted center embedding constructions and convinced that person to spend the rest of their life uttering a single sentence full of center embeddings, that person could only produce a fixed number of embeddings before they die of old age. Even if we somehow could found a tradition of center-embedding performance art, where one speaker starts a sentence with center embeddings that are then continued by another speaker, and then another, on and on until the end of time, we could only reach a final level of embeddings before the universe collapses in on itself. We live in a finitistic universe, wherefore unboundedness is not an empirically verifiable property.

A slightly different, more application-oriented argument posits that even if humans might be theoretically capable of unbounded center-embedding, there is little reason to incorporate this property into our model if they never make use of it. Why make your model more powerful if that power is never needed? This is indeed a valid concern for industrial-grade applications, where time equals money and programs should run as quickly as possible. But even there this issue is not cut and dry. After all, "time equals money" also means that programs should be easy to extend, modify and maintain, since the manhours spent on these tasks do not come for free. When given a choice between an efficient but complicated tool on the one hand or a slower yet elegant tool on the other, the latter might be the better solution. In addition, the elegant tool might actually be the faster one in practice — just because it can theoretically perform much worse does not entail that is does so for the specific task at hand. As you can see, the unboundedness questions does not have a clear cut answer, it all depends on what your goals are.

The ideal solution would be an elegant tool that can be automatically translated into an efficient one if necessary, but that is not always feasible.

Our inquiry is driven by scientific curiosity, so questions of efficiency affect us only to the extent that the resource usage of our model has to be reconcilable with what we know about human cognition. And even this restriction does not outweigh our desire to state insightful generalizations. That is why we put such a high premium on abstraction: by deliberately excluding certain factors we can home in on broad, appealing generalizations. These generalizations still hold in a more detailed model that is closer to the wetware, but they are much harder to discern due to all the complicating factors that come with a more faithful model.

For our purposes, unboundedness is an essential assumption because boundedness acts as a great equalizer that pushes everything with the bounds of finite-state machinery. Recall that we assumed in our discussion of phonology that some long-distance processes are unbounded even though for all practical purposes the length of words is bounded in the same way as the number of center embeddings. We did this because it allowed us to formalize important differences between local and non-local processes without losing track of their commonalities. Similarly, assuming that center embedding is unbounded brings out an important difference between this non-regular kind of embedding and right embedding, which is still regular. This contrast is even reflected in human processing, where right embeddings are much easier to parse than center embeddings. Distinguishing bounded center embedding from bounded right embedding would be more involved.

A brief glance at an FSA for bounded center embedding also reveals that redundancy of this account. Each level of embedding corresponds to a specific subgraph of the automaton, but all these subgraphs look exactly the same. So we have a big number of states that all do exactly the same work, the only difference is that one is used in embedding level 2 and another one in level 5. This also raises the question why natural language grammars treat all those levels exactly the same — if the automaton distinguishes level 2 from level 3, why can't 3 use the opposite word order of 2? This level-sensitive automaton would have exactly the same number of states as the one that treats all levels of embedding the same. If we assume that center embedding is unbounded, though, then it might be possible to show that level-sensitive formalisms are more complicated than those that treat all levels the same.

To sum up, unboundedness is not an undisputable fact, first and foremost it is yet another abstraction in the service of exploring specific questions. However, what we find may serve as indirect evidence for unboundedness when embedded in a system of ancillary assumptions. For example, the enormous size of FSAs with bounded center embedding and the lack of level-sensitive embeddings in natural language suggest that unboundedness is cognitively real, at least if one assumes that small, succinct grammars are preferred for some reason. With other constructions, there may be less of a reason to assume that they are unbounded (for instance multiple wh-movement), so we will always have to weigh carefully what the benefits of unboundedness may be on a case-by-case basis.

2 Tree Languages

2.1 Context-Free Grammars

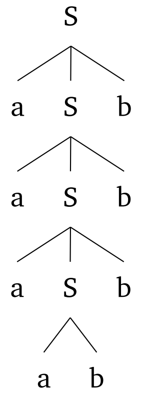
While unbounded center embedding exceeds the limits of finite-state methods, it is easily accounted for with phrase structure rules. The language $a^n b^n$, $n \geq 1$, is generated by two rules, which can even be conflated into a single one with some syntactic sugar:

$$\mathrm{S} \quad \rightarrow \quad \mathrm{ab} \mid \mathrm{aSb}$$

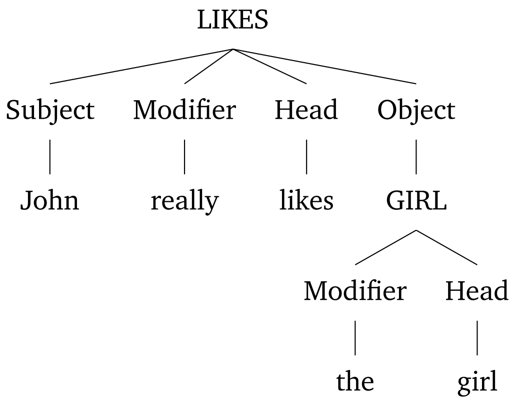
We can interpret this rule as a mechanism for rewriting strings, where we start with S and then apply rules until no more rewriting steps are possible.

Rule	**String**
start	S
S → aSb	aSb
S → aSb	aaSbb
S → aSb	aaaSbbb
S → ab	aaaabbbb

Linguists are more familiar with the tree-based representation of rule application.



Phrase structure grammars are also known as *context-free grammars* (CFGs). The latter term focuses on the rule format, which must be of the form $A \rightarrow \alpha$, where α is a string of symbols. These rules lack the context specification used by SPE, among others, so they are indeed context-free. The term phrase structure grammar instead focuses on what the grammar is meant to describe, namely the phrase structure of sentences. Obviously CFGs can be used to describe other kinds of structures. For instance, the sentence *John really likes the girl* could be assigned the tree below, which represents the functional relations between words (this tree is inspired by Dependency Grammar, which we will discuss at a later point).



As you can see, the term context-free grammars is slightly more general even though it refers to exactly the same kind of mathematical object. Fans of generality that we are, we will henceforth speak of CFGs rather than phrase structure grammars.

Definition 12.1 (CFG). A *context-free grammar* is a triple $G := \langle \Sigma, S, R \rangle$, where

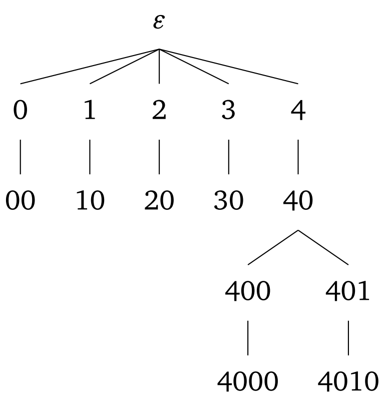
- Σ is an alphabet,
- $S \in \Sigma$ is the *start symbol*,
- R is finite set of rules $A \to \alpha$ such that $A \in \Sigma$ and $\alpha \in \Sigma^*$.

A symbol $a \in \Sigma$ is *terminal* iff R contains no rule of the form $a \to \alpha$. Otherwise a is *non-terminal*. The corresponding subsets of Σ are denoted T_Σ and N_Σ. We always require $S \in N_\Sigma$. The language $L(G)$ generated by G is the smallest set containing all strings that I) can be obtained from S by finitely many applications of rules in R, and II) contain no non-terminal.

Since CFGs can handle center embedding and are equivalent to the familiar linguistic formalism of phrase structure rules, they are a promising starting point for a formal model of syntax. We will soon see that they are indeed very closely related to a model that we have explored in great detail.

2.2 Trees and Tree Languages

Before we move on, it will be useful to formalize trees. This can be done in a plethora of ways, but we will opt for the definition in terms of *Gorn domains* (Gorn 1967) because it couples each node with an address that directly represents its location in the tree. Intuitively, a Gorn domain is a set of addresses of the form $m \cdot l$, where m is the address of the node's mother and l the number of left siblings it has. So the tree for *John really likes the girl* that was given in the previous section would be associated with node addresses as shown below:



The address for the root is ε since it has neither a mother nor a left sibling. For all other nodes, the formula $m \cdot l$ applies as described.

Notice that an entry like 40 is read "four-zero" since it is the leftmost daughter of the fifth daughter of the root, whereas 40 "forty" would refer to the 41st daughter of the node. This ambiguity is due to the decimal system being incapable of representing the difference between $4 \cdot 0$ and 40. Strictly speaking, an address like 401 should actually be written as $4 - 0 - 1$ to distinguish it from $40 - 1$ and 401, but this creates clutter that is best avoided when possible.

Definition 12.2 (Gorn domain). A *Gorn domain* D is a subset of $\mathbb{N}^*$ such that

dominance closure $ui \in D$ implies $u \in D$,

left sibling closure $ui \in D$ implies $uj \in D$ for all $0 \leq j < i$.

We call $u \in D$ a *leaf* iff there is no $i \in \mathbb{N}$ such that $ui \in D$. Given some subset S of D, ui is a *root* of S iff $u \notin S$.

A tree is a Gorn domain that maps each node address to a node label.

Definition 12.3 (Tree). A (finite) Σ-*tree* is a pair $t := \langle D, \ell \rangle$, where

- D is a (finite) Gorn Domain,
- $\ell : D \to \Sigma$ is a total function.

The *string yield* $\text{yd}(t)$ of t is the longest string $s_1 \cdots s_n$ such that

- s_i is a leaf of D $(0 \leq i \leq n)$,
- for $s_i := mu$ and $s_j := nv$ $(m, n \in \mathbb{N}, u, v \in \mathbb{N}^*)$, $i < j$ iff $m < n$.

The *depth* of subtree t with root u is the length of the longest i such that $ui \in D$.

Unless indicated otherwise, all trees are assumed to be finite.

Since we now have both strings and trees as mathematical objects, it makes sense to distinguish between *string languages* and *tree languages*. The former are sets of strings, the latter sets of trees. All the languages we have seen so far were string languages.

Definition 12.4 (Tree Language). A *tree language* L over Σ is a set of Σ-trees. Its string yield is the string language $\text{yd}(L) := \{\text{yd}(t) \mid t \in L\}$.

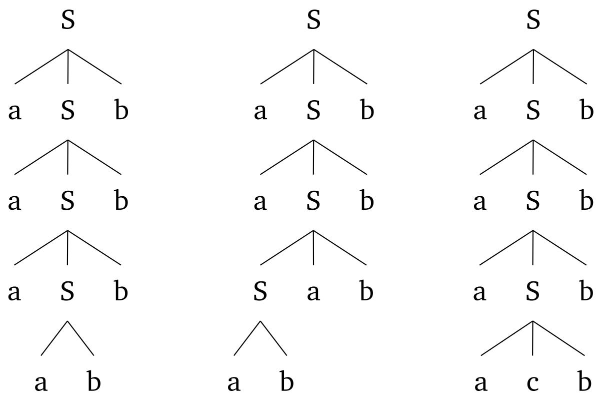
2.3 Tree Languages of Context-Free Grammars

Our definition of CFGs defines the string language generated by the grammar, but not the tree language even though we saw that sequences of rewriting steps can be represented as trees. The intuition for going from rewriting rules to trees is simple enough:

1. Draw a node with label S.
2. If A is rewritten as $\alpha := a_1 \cdots a_n$, add $a_1, \ldots, a_n$ as daughters of the corresponding node (in the same left-to-right order).

This way a CFG can be treated as a mechanism for building trees rather than just strings. But what exactly is the tree language generated by some random CFG?

Let us first think about whether certain trees are generated by a specific grammar, and why this is the case. Below you have several trees. The left one is generated by the CFG $\langle \{S, a, b\}, \{S \to aSb, S \to ab\} \rangle$, the others are not.



The left one satisfies all the conditions the rewrite rules establish with respect to the mother-of and the left-sibling relation. The tree in the middle contains a subtree where S is a left sibling of a, which can never happen with the specified rewrite rules. The third tree contains c as a daughter of S, which is also impossible. Notice that all these violations can be verified in a local fashion: we only have to look at a node and its daughters to find ill-formed subtrees. So we can postulate the following:

Definition 12.5 (CFG Tree Language). A CFG G generates a tree t iff

- the root of t is the start symbol, and
- all leaves of t are terminal, and
- for every subtree s of t such that s is of the form $[_A A_1 \cdots A_n]$, G contains a rule $A \to A_1 \cdots A_n$.

The tree language of G is the set of all trees that are generated by G.

This is a definition, not a theorem. We are simply describing what kind of trees are built via the translation procedure used by linguists. But we can show that this definition is sensible in the sense that the tree language yields the same string language as the grammar.

Theorem 12.6. Let G be a CFG that generates string language L and tree language T. Then $L = \text{yd}(T)$. ⌟

Proof. To see that $L \subseteq \text{yd}(T)$, take any string $w \in L$. By definition, w was obtained from S via a finite number of applications of rewrite rules of G. The standard translation from such rule applications to trees yields a tree that is generated by G and has w as its string yield.

In the other direction $\text{yd}(T) \subseteq L$ follows from the fact that if t is generated by G, then each subtree $[_A\, A_1 \cdots A_n]$ is matched by a rewrite rule of G. Hence there is a sequences of rewrite rules that produces the string yield of t. Since t's string yield consists only of terminal symbols, and t's root is S, $\text{yd}(t)$ is generated by G. □

2.4 Strictly Local Tree Languages

The condition that every subtree of depth 1 must be matched by a rewrite rule could be simplified by converting each rewrite rule into a tree of depth 1. Instead of a finite set of rewrite rules, one would then have a finite set of subtrees of depth 1, and a tree is generated by the grammar iff all its subtrees of depth 1 are included in this set. This idea should sound awfully familiar to you: it is the tree analogue of strictly local grammars.

Definition 12.7 (*k*-trees). A *k-tree* over alphabet Σ is a tree of depth $k-1$. A *k-augment* is a unary branching tree of depth $k-1$ where every node is labeled $\rtimes$. Given a Σ-tree t, its *k-augmented* counterpart $\hat{t}_k$ is the result of adding a k-augment above the root and below each leaf. The set of k-trees of a tree t is given by $2\text{-trees}(t) := \{s \mid s \text{ is a subtree of } t \text{ with depth } k-1\}$.

Definition 12.8 (Strictly Local Tree Language). A finite set of k-trees is called a *strictly k-local tree grammar*. A positive strictly k-local tree grammar G generates the tree language $L(G) := \{t \mid k\text{-trees}(t) \subseteq G\}$. A negative strictly k-local tree grammar G generates the tree language $L(G) := \{t \mid k\text{-trees}(t) \cap G = \emptyset\}$. A tree language L is strictly k-local iff it is generated by some strictly k-local tree grammar. The class of strictly k-local tree languages is denoted SL^{T}_k, and the class of all *strictly local tree languages* is given by $\text{SL}^{\text{T}} := \bigcup_{k \geq 1} \text{SL}^{\text{T}}_k$.

Many of the theorems that we established for strictly local string languages can easily be lifted to strictly local tree languages. Before we do that, though, let us verify that strictly local tree languages are indeed closely related to the tree languages generated by context-free grammars.

Theorem 12.9. The class of tree languages generated by CFGs is properly included in the class of strictly 2-local tree languages. ⌟

Proof. For every CFG G one can construct an equivalent strictly 2-local tree grammar G_2. For every rewrite rule $A \rightarrow A_1 \cdots A_n$ of G, G_2 contains the 2-tree $[_A\, A_1 \cdots A_n]$. For every terminal symbol a, we add $[_a \rtimes]$. Finally, G also contains $[_\rtimes S]$. It is easy to see that G_2 generates exactly the same tree language.

Inclusion is proper because a symbol can be both terminal and non-terminal in a strictly 2-local tree language. Consider for instance the grammar $\{[_\rtimes S], [_S \rtimes]\}$, which generates only the tree S. □

2.5 Properties of Strictly Local Tree Languages

Just as we did with for strictly local string languages, we first show that it does not matter whether a grammar is positive or negative. That way, we can freely choose between the two in all other proofs.

Lemma 12.10. Let $T(\Sigma, n)$ be the set of Σ trees that are at most n-ary branching (i.e. every node has at most n daughters). For every positive strictly k-local grammar ($k \geq 1$) that generates some tree language L over Σ, there is a negative strictly k-local grammar that generates a tree language L' over Σ such that $L \cap T(\Sigma, n) = L' \cap T(\Sigma, n')$. ⌟

Proof. Since there is only a finite number of at most n-ary branching k-trees over Σ, we can adopt the strategy we used in the string case: the negative grammar is defined as the set of all these k-trees that are not contained by the positive grammar. When restricted to $T(\Sigma, n)$, both grammars generate the same set of trees. □

The equivalence of positive and negative grammars is no longer as strong as in the string case because it now hinges on a restriction on the allowed maximum branching factor. If the branching factor is not fixed, then the two grammar formats diverge significantly: a positive grammar can only generate trees with a fixed branching factor (namely the maximum branching factor of all k-trees in the grammar), whereas a negative grammar has no such upper bound (if all its k-trees are binary branching, than all trees that are not binary branching are automatically well-formed). This was not an issue for strings, which are essentially unary branching trees. It won't affect us much either, as most theories of syntax assume that branching is binary, or at least less than 4-ary, and we will henceforth assume that the branching factor is restricted to some suitable number.

One exception is "flat" analyses of coordination, where each conjunct adds a new branch. There is plenty of evidence, though, that such an analysis is incorrect.

All other properties of strictly local string languages also carry over with some minor adjustments.

Theorem 12.11. For all $k \geq 1$, $\mathrm{SL}^{\mathrm{T}}_{k-1} \subsetneq \mathrm{SL}^{\mathrm{T}}_{k}$. ⌟

Proof. We use the fact that every string $w := a_1 a_2 \cdots a_{n-1} a_n$ can be viewed as a unary branching tree $t(w) := [_{a_1}[_{a_2}[\cdots[_{a_{n-1}} a_n]\cdots]]]$. Just like the singleton string language $L^S_n := \{a^n\}$ is strictly $n+1$-local but not strictly n-local, the singleton tree language $L^T_n := \{t(a^n)\}$ is in $\mathrm{SL}^{\mathrm{T}}_{n+1}$ but not in $\mathrm{SL}^{\mathrm{T}}_{n}$. □

By viewing strings as unary branching trees, several non-closure results also carry over immediately.

Theorem 12.12. The class of strictly local tree languages is not closed under union, relative complement, or relabeling. ⌟

Proof. Remember that the union of $\left\{a^k b^k\right\} \in \mathrm{SL}_{k+1}$ and $\left\{b^+\right\} \in SL_2$ is not strictly local. Since each string can be viewed as a unary branching tree, this also establishes non-closure under union for strictly local tree languages.

For relative complement, consider the tree analogue T of the language $\{ab^*a\}$, which is in $\mathrm{SL}^{\mathrm{T}}_{2}$. Yet $T(\Sigma, n) \setminus T$ is not strictly local.

Non-closure under relabeling is once again witnessed by the languages $(ab)^+$ and $(aa)^+$, lifted from strings to unary branching trees. The former is in $\mathrm{SL}^{\mathrm{T}}_{2}$, whereas the latter is not strictly local. Yet the latter is a relabeling of the former. □

Theorem 12.13. For every $k \geq 0$, $\mathrm{SL}^{\mathrm{T}}_k$ is closed under intersection. ⌟

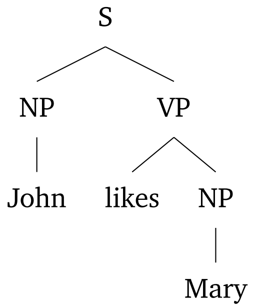
Proof. The strategy is once again to show that $L(G_1) \cap L(G_2) = L(G_1 \cap G_2)$, where G_1 and G_2 are positive strictly k-local tree grammars. This is left as an exercise to the reader. □

2.6 Local Subtree Substitution

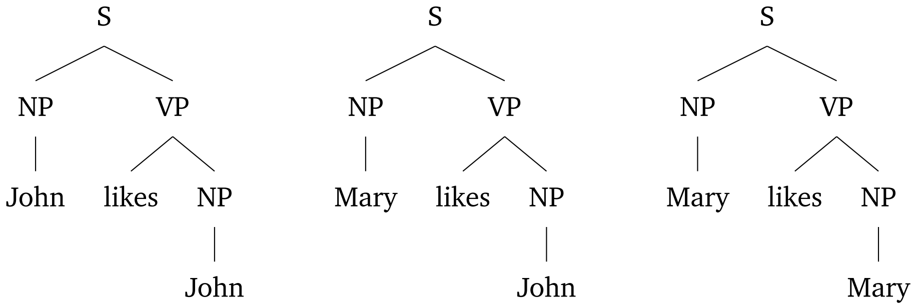
Strictly local string languages are fully characterized by local subtree substitution closure, as was discussed in Sec. 2.3. Intuitively, it is fairly obvious that a similar property holds for strictly local tree languages.

Example 12.1 Context-Free Grammars and Subtree Substitution

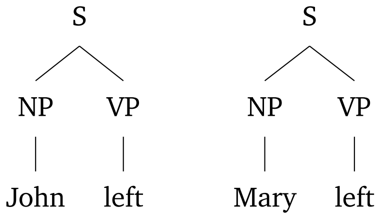
Suppose we have a context-free grammar that generates the tree below.



Then the grammar must generate at least three more trees:



This follows immediately from the fact that these three trees only use k-trees that also occur in the first one. If we furthermore know that the left tree below is generated by the grammar, then we know that it also generates the right one. In this case, there is no previous tree that contains all k-trees of the right tree, but its k-trees are a subset of the union of the k-trees of some trees that are already known to be grammatical.



For strictly 2-local tree languages, all trees with identical root labels can be freely substituted for one another. Note that these are exactly the subtrees that are identical up to depth 0. As the locality domain increases, so does the amount of material that two trees must have in common before they can be exchanged. For example, in a 4-local tree language we can substitute subtree t for u iff t and u are identical up to and including depth 2. We use the notation $s[u \leftarrow t]$ for the tree that is obtained from tree s by replacing the subtree of s rooted in address u by the tree t. So if s the tree $[_{\text{S}}$ $[_{\text{NP}}$ John] $[_{\text{VP}}$ left]], then $s[00 \leftarrow \text{Mary}]$ is $[_{\text{S}}$ $[_{\text{NP}}$ Mary] $[_{\text{VP}}$ left]]. The notation can also be nested:

$$s[00 \leftarrow \text{Mary}[\varepsilon \leftarrow \text{John}]] = s[00 \leftarrow \text{John}] = s$$

And several substitutions can be carried out at once as long as no address is a prefix of another (that is to say, the nodes at which we substitute do not stand in the dominance relation). For example, $s[00 \leftarrow \text{Mary}, 10 \leftarrow \text{laughed}] = [_{\text{S}}\ [_{\text{NP}}\ \text{Mary}]\ [_{\text{VP}}\ \text{laughed}]]$.

Definition 12.14 (Subtree Substitution). Let $s := \langle D_s, \ell_s \rangle$ and $t := \langle D_t, \ell_t \rangle$ be Σ-trees and $u \in D_s$. Then $s[u \leftarrow t]$ is the unique tree $\langle D, \ell \rangle$ s.t.

- $D := (D_s \setminus \{a \mid a = ui, i \in \mathbb{N}^*\}) \cup \{a \mid a = ui, i \in D_t\}$,
- if $a \in \{a \mid a = ui, i \in D_t\}$, then $\ell(a) = \ell_t(a)$,
- otherwise $\ell(a) = \ell_s(a)$.

Suppose $u_1, \ldots, u_n$ are elements of D_s such that no u_i is a prefix of some distinct u_j. Then

$$s[u_1 \leftarrow t_1, u_2 \leftarrow t_2, \ldots, u_n \leftarrow t_n] := (\ldots((s[u_1 \leftarrow t_1])[u_2 \leftarrow t_2])\ldots)[u_n \leftarrow t_n]$$

With that little bit of notation out of the way, we can finally characterize strictly local tree languages in terms of local subtree substitution closure.

Definition 12.15 (Local Subtree Substitution Closure). A tree language L^T satisfies *local subtree substitution closure* iff there is some $k \geq 1$ such that if

- L^T contains $s[u \leftarrow x[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]]$, and
- L^T contains $t[u' \leftarrow x[x_1 \leftarrow v'_1, \ldots, x_n \leftarrow v'_n]]$, and
- x is a tree of depth $k - 1$,

then L also contains $s[u \leftarrow x[x_1 \leftarrow v'_1, \ldots, x_n \leftarrow v'_n]]$.

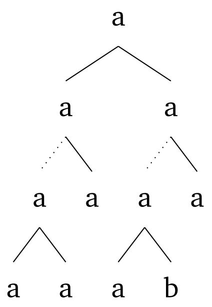
As in the string case, we omit the proof here, which is rather lengthy. The interested reader is referred to Rogers (1997).

Example 12.2 Some Non-Local Tree Languages

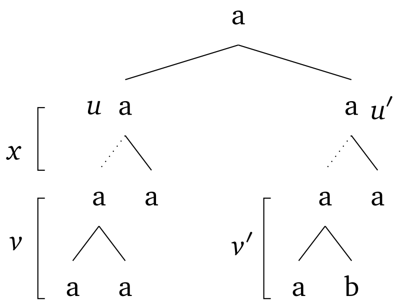
The simplest application of local subtree substitution shows that the language L of unary branching trees over the alphabet $\{a\}$ that contain an even number of as is not strictly local. The counterexample is exactly the one we already know from the string case. Suppose $k \geq 2$ is even. Then for s we pick $[_{\mathrm{a}}\ \mathrm{a}\ ]$ and for t just a; x is the tree $[_{\mathrm{a}}\cdots[_{\mathrm{a}}a]\cdots]$ with $k-1$ instances of a (an odd number). Finally, $v = t$ and $v' = s$. Then we have $s[0 \leftarrow x[0^{k-2} \leftarrow v]] \in L$, $t[\varepsilon \leftarrow x[0^{k-2} \leftarrow v']] \in L$, yet $s[0 \leftarrow x[0^{k-2} \leftarrow v']] \notin L$. Let us quickly verify this for $k = 4$:

$$\begin{array}{rll}
x[0^{k-2} \leftarrow v] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]][00 \leftarrow \mathrm{a}] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]] \\
s[0 \leftarrow x[0^{k-2} \leftarrow v]] & = [_{\mathrm{a}}\ \mathrm{a}\ ][0 \leftarrow [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]] \in L \\
\hline
x[0^{k-2} \leftarrow v'] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]][00 \leftarrow [_{\mathrm{a}}\ \mathrm{a}]] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]] \\
t[\varepsilon \leftarrow x[0^{k-2} \leftarrow v']] & = \mathrm{a}[\varepsilon \leftarrow [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]]] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]] \in L \\
\hline\hline
s[0 \leftarrow x[0^{k-2} \leftarrow v']] & = [_{\mathrm{a}}\ \mathrm{a}][0 \leftarrow [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]]] & = [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ [_{\mathrm{a}}\ \mathrm{a}\ ]]]] \notin L
\end{array}$$

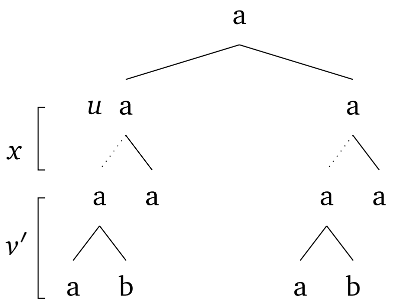
While the reasoning is simple, the notation makes it very hard to follow, so in general it is advisable to use a tree-based representation instead, as we shall do for our next example. Let $L_{b=1}$ be the language of all strictly binary branching trees over $\{a, b\}$ that contain exactly one instance of b. This language contains at least the following tree, where the dashed branch spans a subtree of depth $k-1$.



Then we can carve up the tree in a way so that we can assemble the parts into an illicit tree via local subtree substitution.



The whole tree is chosen for both s and t. Then we have $s[u \leftarrow x[0^{k-2} \leftarrow v]] = t[u' \leftarrow x[0^{k-2} \leftarrow v']] = s \in L_{b=1}$. But $s[u \leftarrow x[0^{k-2} \leftarrow v']]$ is not in $L_{b=1}$.



Unit 13

Notions of Generative Capacity

Last time we saw that even though syntax does not define regular string languages due to center embedding (and possibly other constructions we have not considered yet), context-free grammars are powerful enough to handle center embedding. And in an unexpected turn of events we quickly realized that context-free grammars can be viewed as a mechanism for generating strictly 2-local tree languages. So if this is on the right track, syntax does not look all that different from phonology: where a significant fragment of phonology is strictly local over strings, syntax may be strictly local over trees. But as you remember, not all of phonology turned out to be strictly local, so we should not be too eager to claim that syntax is strictly local, either. And as we will see today, this issue is a lot trickier to resolve for syntax.

1 Weak and Strong Generative Capacity

A strictly local tree grammar produces two distinct outputs — a tree language and the corresponding string language. These two notions are not in perfect alignment. Failure to produce the right string language necessarily implies failure to produce the right tree language, but the opposite does not hold. Moreover, a formalism that does not produce the right string language may still be closer to the correct tree language than one that gets the string language right but does so at the expense of ludicrous tree structures. So if we assume that trees are indeed an integral part of syntax — rather than just a convenient device for us to represent certain dependencies over strings — then we have to make sure that strictly local grammars are sufficiently powerful to capture syntax at both the string level and the tree level. In more technical terms, we have to ensure that they have adequate *weak and strong generative capacity*.

Weak Generative Capacity the class of string languages that are generated by the formalism

Strong Generative Capacity the class of tree languages that are generated by the formalism

Weak and strong generative capacity do not exhaust the full spectrum of levels of generative capacity. One could also posit *derivational capacity* as a metric for how succinctly a formalism produces a given tree, or *relational capacity* as a metric for the string-meaning pairings that can be computed. These notions are very little understood, though, so they play no big role at this point in the categorization of formalisms. Even strong generative capacity is still severely understudied, though a lot of progress has been made in recent years. This is partially due to theoretical computational linguistics undergoing a shift in interest from string languages to tree languages. Another driving factor, however, is that more and more application areas build on tree languages. XML, for example, is essentially a standard for dynamically specifying tree languages, which grants it an enormous amount of flexibility and makes it an ideal tool for APIs, among other things. So this is yet another case where theoretical and applied interests converge, leading towards a new focus on understudied concepts — in the case at hand, tree languages.

2 Generative Capacity of Tree-Based Grammar Formalisms

2.1 Strictly Local Tree Grammars

We already proved that $\mathrm{SL}^{\mathrm{T}}_{k-1} \subsetneq \mathrm{SL}^{\mathrm{T}}_{k}$ for all k, so the strong generative capacity of strictly local tree grammars increases with the size of their locality domain. In addition, the tree languages generated by CFGs are a proper subclass of $\mathrm{SL}^{\mathrm{T}}_{2}$ due to CFG's strict distinction between terminal and non-terminal nodes. With respect to strong generative capacity, we thus obtain a strict hierarchy CFG $\subsetneq \mathrm{SL}^{\mathrm{T}}_{2} \subsetneq \mathrm{SL}^{\mathrm{T}}_{3} \subsetneq \cdots$. The strong generative capacity of CFGs and strictly 1-local tree grammars is incomparable. In fact, the two classes have only one tree language in common, and that is the empty language. With the exception of these slightly deviant cases, though, we get a nice proper hierarchy that mirrors exactly the expressive hierarchy for strictly local string languages.

What does a CFG for the empty tree language look like? And what is the corresponding strictly 1-local tree grammar?

The fact that strong generative capacity increases with the size of the locality domain suggests that weak generative capacity does too. After all, that's what we saw with strictly local string languages. Surprisingly, though, this is not the case: weak generative capacity stays the same after $\mathrm{SL}^{\mathrm{T}}_{2}$.

Theorem 13.1. For all $k \geq 2$, $\mathrm{yd}(\mathrm{SL}^{\mathrm{T}}_{k})$ is the class of context-free languages. ⌟

We do not give a proof at this point since the theorem will fall out as a corollary of an even more general result later on. Instead, let us look at some of the surprising repercussions of this fact. A first important observation is that the string languages generated by strictly local tree grammars have very different closure properties compared to their tree languages.

Lemma 13.2. The class of context-free languages is closed under union. ⌟

Proof. It suffices to show that for any two CFGs $G_1 := \langle \Sigma_1, S, R_1 \rangle$ and $G_2 := \langle \Sigma_2, S, R_2 \rangle$ there is a CFG G_3 that generates $L(G_1) \cup L(G_2)$. One can construct G_3 in a manner that is similar to the union of automata:

- for every rewrite rule $A \to A_1 \cdots A_n$ in R_i $(i \in \{1,2\})$, G_3 contains the rewrite rule $A^i \to A^i_1 \cdots A^i_1$,
- for every terminal symbol a of G_i $(i \in \{1,2\})$, G_3 contains the rewrite rule $a^i \to a$,
- the start symbol S of G_3 only occurs in the rewrite rules $S \to S^1$ and $S \to S^2$.

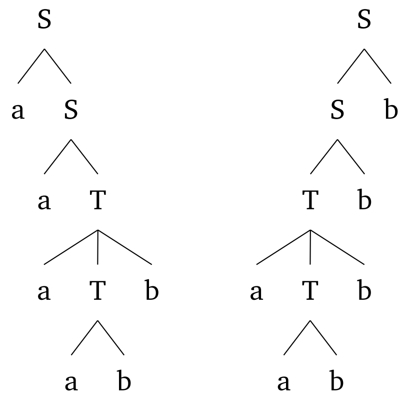
It is easy to see that G_3 generates every string that is a member of $L(G_1)$ or $L(G_2)$, and nothing else. □

Example 13.1 Constructing the Union of Two CFGs

Suppose $m > n \geq 1$. Then $a^m b^n$ and $a^n b^m$ are both context-free languages generated by the respective set of rewrite rules below:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
S & \rightarrow & aS & S & \rightarrow & Sb \\
S & \rightarrow & aT & S & \rightarrow & Tb \\
T & \rightarrow & aTb & T & \rightarrow & aTb \\
T & \rightarrow & ab & T & \rightarrow & ab
\end{array}$$

The trees for *aaaabb* and *aabbbb* are shown below.



The union of $a^m b^n$ and $a^n b^m$ is $a^+ b^+ \setminus a^n b^n$.

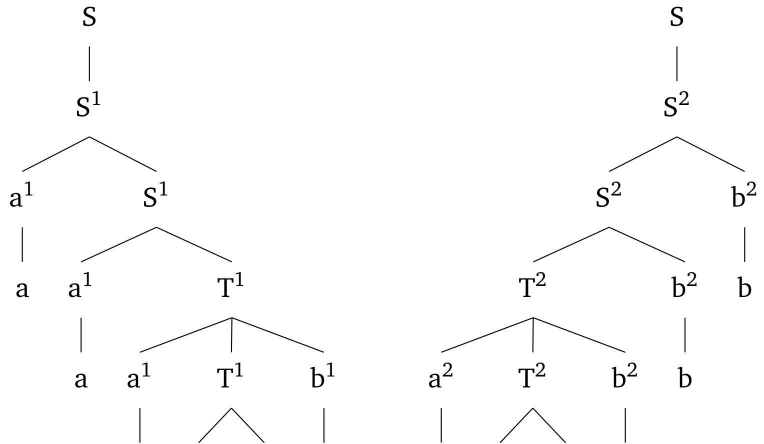
We now build a CFG G_3 for this language, meticulously following the steps in the proof. The first step collects all rewrite rules and adds a superscript to every symbol in the rule.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
S^1 & \rightarrow & a^1 S^1 & S^2 & \rightarrow & S^2 b^2 \\
S^1 & \rightarrow & a^1 T^1 & S^2 & \rightarrow & T^2 b^2 \\
T^1 & \rightarrow & a^1 T^1 b^1 & T^2 & \rightarrow & a^2 T^2 b^2 \\
T^1 & \rightarrow & a^1 b^1 & T^2 & \rightarrow & a^2 b^2
\end{array}$$

Next we have to add a rewrite rule for every non-terminal that we superscripted.

$$\begin{array}{lcl}
a^1 & \rightarrow & \mathrm{a} \\
a^2 & \rightarrow & \mathrm{a} \\
b^1 & \rightarrow & \mathrm{b} \\
b^2 & \rightarrow & \mathrm{b}
\end{array}$$

Finally, we add the rules $S \rightarrow S_1$ and $S \rightarrow S_2$. After the initial rewrite step of S, this grammar behaves almost exactly like G_1 or G_2, depending on what S was rewritten as.



Closure under union is surprising, but even more so is non-closure under intersection.

Lemma 13.3. The class of context-free languages is not closed under intersection. ⌟

Proof. We are not in a position to fully prove this yet. The basic idea is as follows: $a^+ b^n c^n$ and $a^n b^n c^+$ are both context-free languages, but their intersection is $a^n b^n c^n$, which is not context-free. This can be shown via a pumping lemma, which we will encounter later on. □

How is it possible that the strictly local tree languages are closed under intersection but not union, while the very opposite holds for their string yields? Shouldn't closure under intersection for tree languages imply closure under intersection for the string yields, too? The answer is no, because the two intersection operations produce very different outputs. Consider the tree languages L_1 and L_2 that only contain the trees $[_{\mathrm{S}}\ [_{\mathrm{A}}\ \mathrm{a}\ ]]$ and $[_{\mathrm{S}}\ [_{\mathrm{B}}\ \mathrm{a}\ ]]$, respectively. Then $\mathrm{yd}(L_1) = \mathrm{yd}(L_2) = \{a\}$, so the intersection of their string languages is also $\{a\}$. But if we directly intersect L_1 and L_2, we get the empty set instead because they have not a single tree in common. And the string yield of the empty set is also the empty set, which definitely is not the same thing as $\{a\}$. So $\mathrm{yd}(L_1) \cap \mathrm{yd}(L_2) = \{a\} \neq \emptyset = \mathrm{yd}(L_1 \cap L_2)$. It is this misalignment between intersecting tree languages and intersection their string yields that leads to the skewed closure property.

A similar problem arises with union. Union of strictly local tree languages does not necessarily preserve their strict locality because we might be able to build completely new trees from the union of the k-trees. But for union of the string yields, we can freely alter the tree structure, including relabeling interior nodes to keep the k-grams distinct. So once again $\mathrm{yd}(L_1) \cup \mathrm{yd}(L_2)$ is not necessarily the same as $\mathrm{yd}(L_1 \cup L_2)$, and closure under union for string languages cannot be lifted to the level of tree languages.

Keeping all of this mind, you should not be too surprised anymore that context-free languages are also closed under relabelings even though the strictly local tree languages are not.

Lemma 13.4. The class of context-free languages is closed under relabelings. ⌟

Proof. Suppose L is generated by CFG $G := \langle \Sigma, S, R \rangle$ and the relabeling τ is specified as a finite set of pairs $\langle a, b \rangle$ such that $a \in \Sigma_T$. Then the image of L under τ is generated by the grammar G' with $R' := R \cup \{a \to b \mid \langle a, b \rangle \in \tau\}$. □

Theorem 13.5. The class of context-free languages is closed under union and relabelings. It is not closed under intersection and relative complement. ⌟

The difference in closure properties is both a strength and a weakness. On the one hand, it makes things a lot trickier, and one always has to pay attention what kind of objects are being manipulated, tree languages or string languages. On the other hand, it solves a problem we would be facing otherwise: if syntax defines context-free string languages, shouldn't we expect the union of two natural languages to be a natural language? In our discussion of phonology we already pointed out that closure under union is not a property of natural languages because it allows constraints to apply disjunctively. For syntax, the fact that some languages only enforce person agreement between the subject and the verb and some only number agreement would predict via closure under union that there is a language where the subject has to agree in person or number, but not both. This seems unlikely, and it is readily accounted for if we take syntax to generate tree languages, where closure under union does not hold.

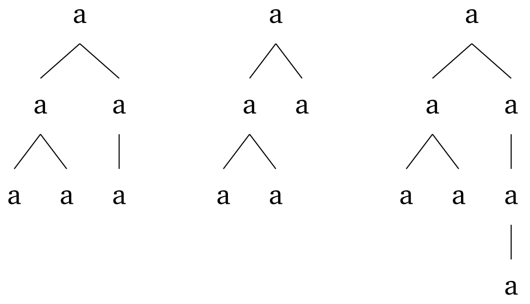
2.2 Refined Strictly Local Tree Grammars

Now that we have a tree analogue for strictly local grammars, it makes sense to define an analogue for refined strictly local grammars and ask how their generative capacity compares to that of standard strictly local tree grammars.

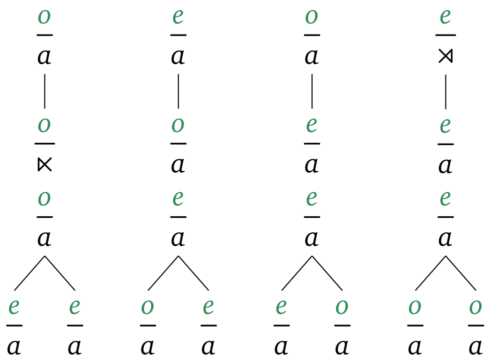
Definition 13.6 (Refined Tree Grammar). A *refined strictly k-local tree grammar G* is a finite set of k-trees over alphabet $\Sigma \times Q$. Such a grammar generates the tree language $L^T(G) := \left\{ t \mid \exists r \in Q^{\hat{t}}, k\text{-trees}(r) \subseteq G \right\}$. Its string language is $L^S(G) = \mathrm{yd}(L^T(G))$. A tree language is refined strictly k-local (in $\mathrm{SL}^{\mathrm{T,R}}$) iff it is generated by a refined strictly k-local tree grammar. The class of all refined strictly k-local tree languages is $\mathrm{SL}^{\mathrm{T,R}} := \bigcup_{k \geq 0} \mathrm{SL}_k^{\mathrm{T,R}}$.

Example 13.2 Two Refined Strictly 2-Local Tree Languages

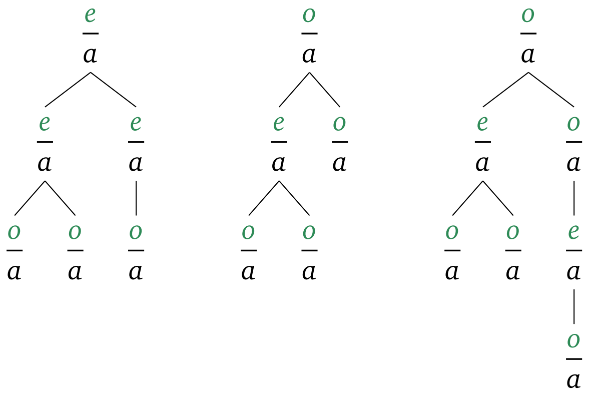
In lecture 12, we saw example of two tree languages that are not strictly local. The first one was the set of unary branching trees over alphabet $\{a\}$ that contain an even number of nodes. This language is refined strictly 2-local, though, as the construction for the string case can be lifted directly to unary branching trees. Let us look at a slight generalization instead, the set of all at most binary branching trees over alphabet $\{a\}$ with an even number of nodes. This language contains the leftmost tree but not the other two.



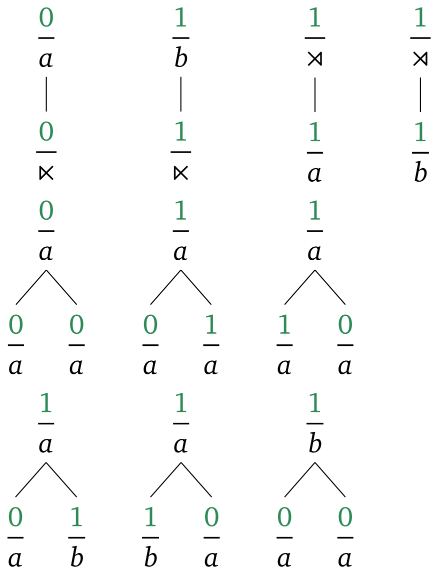
In order to generate this tree language, one needs the refined 2-trees below:



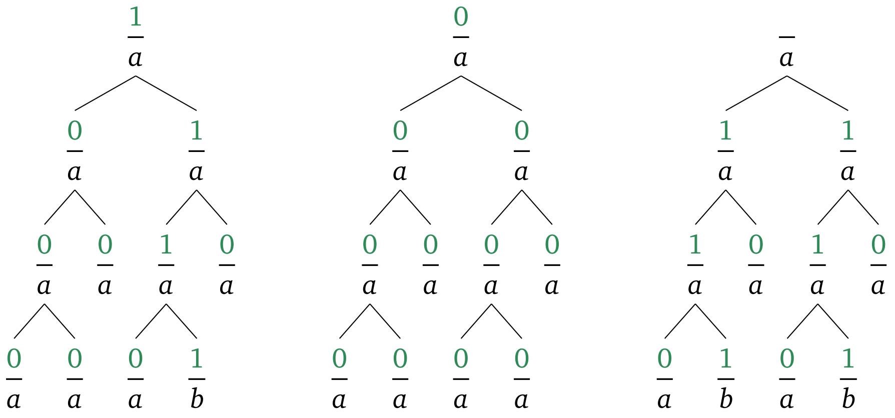
The state assignments for the three trees above now show that only the leftmost is generated by the grammar.



The other language we looked at was $L_{b=1}$, the set of all strictly binary branching trees over $\{a, b\}$ that contain exactly one instance of b. In this case, the states simply keep track of whether a b has been seen yet.



The state assignment once again shows the distinction between well-formed and ill-formed trees.



During our investigation of refined strictly local string languages, we saw that $\mathrm{SL}^R_k = \mathrm{SL}^R_{k+1}$ for all $k \geq 2$, which implies $\mathrm{SL}^R_2 = \mathrm{SL}^R$. The equivalence holds because the extra information that is afforded by the increased locality domain can be encoded in the hidden alphabet layer. For the very same reason, every strictly local language turns out to be refined strictly 2-local. Adapting the constructions used in the proof from strings to trees is straight-forward, and consequently we find the same relations among refined strictly local tree languages.

Theorem 13.7. $\mathrm{SL}^{\mathrm{T}} \subsetneq \mathrm{SL}^{\mathrm{T,R}}_2 = \mathrm{SL}^{\mathrm{T,R}}$ ┘

These relations will allow us to prove rather elegantly that all $\mathrm{SL}^{\mathrm{T}}_k$ have the same weak generative capacity. For this is just a corollary of the fact that the class of context-free languages is exactly $\mathrm{yd}(\mathrm{SL}^{\mathrm{T,R}}_2)$.

Theorem 13.8 (Thatcher 1967). CFGs are weakly equivalent to $\mathrm{SL}^{\mathrm{T,R}}$. ┘

Proof. Since every CFG defines a strictly 2-local tree language and $\mathrm{SL}^{\mathrm{T}} \subsetneq \mathrm{SL}^{\mathrm{T,R}}_2$, every context-free language is included in $\mathrm{yd}(\mathrm{SL}^{\mathrm{T,R}}_2)$. In the other direction, let $L \in \mathrm{SL}^{\mathrm{T,R}}$. Then there is some refined strictly 2-local tree grammar G_2 with $L^T(G_2) = L$. We construct a CFG G_C such that $L^S(G_2) = L^S(G_C)$.

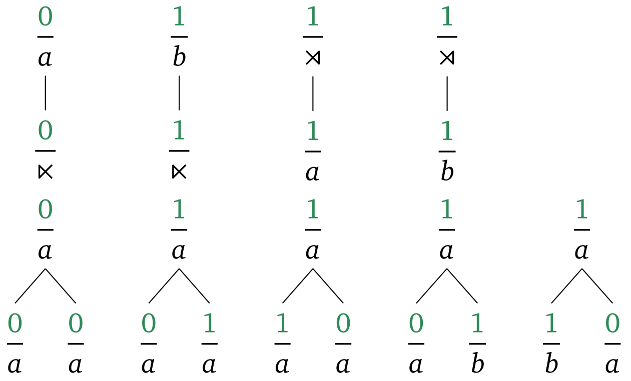
Let $k \in G_2$.

- If k is of the form $[\,\frac{q}{\rtimes}\ \frac{q_1}{A_1}\,]$, then G_C contains the rewrite rule $S \rightarrow \langle A_1, q_1 \rangle$.
- If k is of the form $[\,\frac{q}{A}\ \frac{q_1}{\ltimes}\,]$, then G_C contains the rewrite rule $\langle A, q \rangle \rightarrow A$.
- In all other cases, k is of the form $[\,\frac{q}{A}\ \frac{q_1}{A_1} \cdots \frac{q_n}{A_n}\,]$ $(n \geq 1)$ and G_C contains the rewrite rule $\langle A, q \rangle \rightarrow \langle A_1, q_1 \rangle \cdots \langle A_n, q_n \rangle$.

Let τ be the projection that maps every non-terminal symbol $\langle A, q \rangle$ to A. Furthermore, $t[l \leftarrow l^2]$ is the tree that is obtained from t by replacing every leaf a with the tree $[_a a]$. Note that $\mathrm{yd}(t) = \mathrm{yd}(t[\leftarrow l^2])$. Close inspection of the translation above reveals that $s \in L^T(G_2)$ iff there is a tree $t \in L^T(G_C)$ such that $\tau(t) = [_S s[l \leftarrow l^2]]$. It follows that $\mathrm{yd}(s) = \mathrm{yd}(t)$, and by extension $L^S(G_2) = L^S(G_C)$. □

Example 13.3 Translating a Refined Tree Grammar Into a CFG

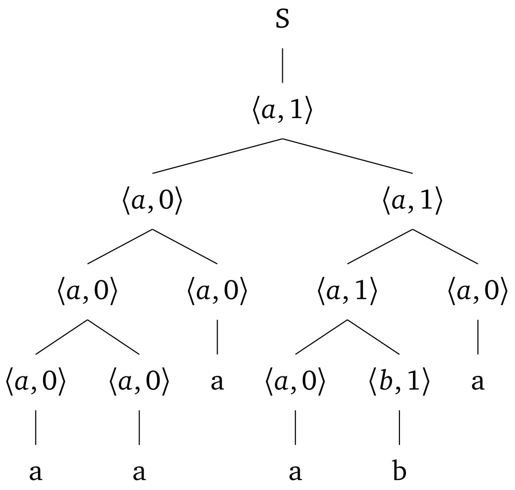
The previous example gave a refined strictly 2-local grammar for $L_{b=1}$, the set of all strictly binary branching trees over $\{a, b\}$ that contain exactly one instance of b. We consider a minor variant where only leafs can be labeled b. To this end we have to remove only one 2-tree from the original grammar.

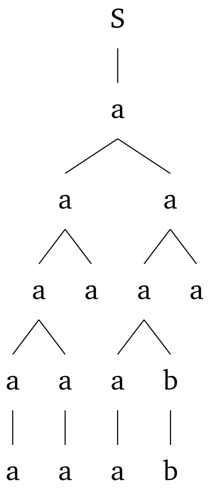

0/a | 0/⋉ | 0/a | 0/a 0/a
1/b | 1/⋉ | 1/a | 0/a 1/a
1/⋊ | 1/a | 1/a | 1/a 0/a
1/⋊ | 1/b | 1/a | 0/a 1/b
1/a | 1/b 0/a

The procedure described in the proof translates these nine k-grams into nine rewrite rules.

$$
\begin{array}{lcl}
\langle a,0\rangle & \rightarrow & a \\
\langle b,1\rangle & \rightarrow & b
\end{array}
\qquad
\begin{array}{lcl}
S & \rightarrow & \langle a,1\rangle \\
S & \rightarrow & \langle b,1\rangle
\end{array}
\qquad
\begin{array}{lcl}
\langle a,0\rangle & \rightarrow & \langle a,0\rangle\ \langle a,0\rangle \\
\langle a,1\rangle & \rightarrow & \langle a,0\rangle\ \langle a,1\rangle \\
\langle a,1\rangle & \rightarrow & \langle a,1\rangle\ \langle a,0\rangle \\
\langle a,1\rangle & \rightarrow & \langle a,0\rangle\ \langle b,1\rangle \\
\langle a,1\rangle & \rightarrow & \langle b,1\rangle\ \langle a,0\rangle
\end{array}
$$

The well-formed tree from the previous example now has the left tree as its correspondent, while the right tree shows the image under τ.





We now have a very intriguing result: CFGs, strictly local tree grammars, and refined strictly local tree grammars all have the same weak generative capacity. The choice between them thus cannot be motivated by well-formedness data since the well-formedness of a sentence in, say, English only tells us that it belongs to the string language defined by English, and all these formalisms define exactly the same string languages. If we want to make an informed choice between these formalisms, it will have to be in terms of the tree languages they can define. That is a much more subtle issue that requires a lot of linguistic finesse. But more on that next time.

Relevant literature for Unit 13

add more info

Thatcher, James W. 1967. Characterizing derivation trees for context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences* 1. 317–322.

Unit 14

Syntactic Dependencies Over Trees

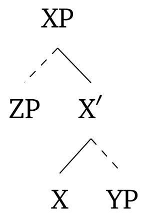
We found out that CFGs, strictly local tree grammars, and refined strictly local tree grammars all have the same weak generative capacity but differ in their strong generative capacity such that $\text{CFG} \subsetneq \text{SL}^{\text{T}}_{2} \subsetneq \text{SL}^{\text{T}}_{3} \subsetneq \cdots \subsetneq \text{SL}^{\text{T,R}}_{2} = \text{SL}^{\text{T,R}}$. This leaves us with the question which one of them is adequately powerful to express syntactic dependencies over trees. Although this issue involves barely any mathematics, it turns out to be a lot harder to answer than anything we have looked at so far.

1 The Complexity of Syntactic Dependencies

1.1 Headedness and Projection

Both CFGs and strictly local tree languages are egalitarian in the sense that all nodes are on equal footing without one enjoying a special status. For example, a rewrite rule like AP → B C is no less natural than AP → A C. This contrasts with one of the important findings of 20th century linguistics: sentences don't just have tree structures, these structures are the result of combining phrases, and each phrase is projected by a head.

These ideas were formalized in terms of X′-theory, where each lexical item X is the head of a phrase XP (Jackendoff 1977) as shown below.



Note that every phrase is at most unary branching. X′-theory has been subjected to extensive criticism over the years (Kornai & Pullum 1990, Chomsky 1993) and is no longer entertained in its canonical form, with current models positing that I) all phrases are strictly binary branching, ruling out unary branching nodes, and II) nodes are labeled in a less elaborate manner. The basic idea that trees are combinations of headed phrases, however, is still upheld across a variety of formalisms.

X′-theory can be defined as a restriction on the shape of rewrite rules, so it does not fall outside the purview of strictly 2-local tree languages.

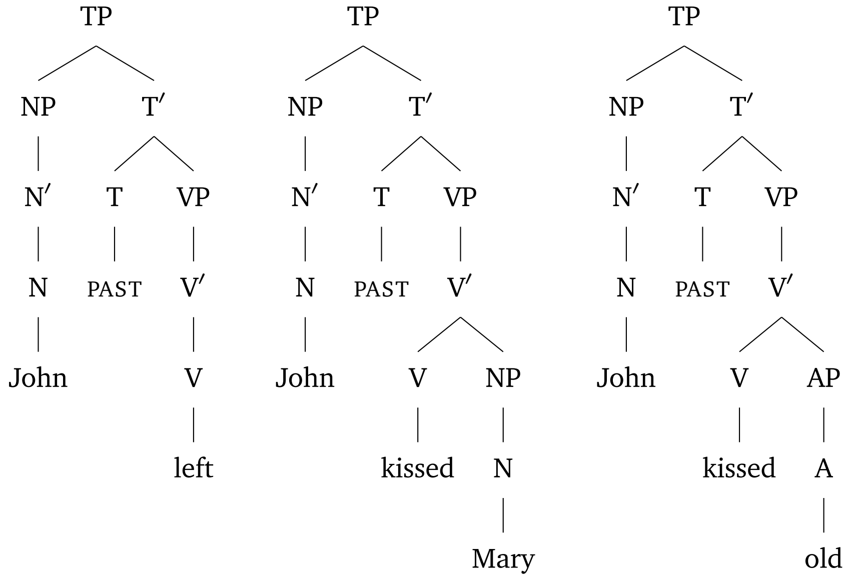
X′-template Every rewrite rule $A \rightarrow A_1 \cdots A_n$ must satisfy the following rules:

- $n \leq 2$,
- if $A = XP$ for some X, then either $A_1 = X'$ and $n = 1$, or $A_2 = X'$ and $A_1 = YP$ for some Y.
- if $A = X'$ for some X, then $A_1 = X$ and either $n = 1$ or $A_2 = YP$ for some Y.
- if $A = X$, then $n = 1$ and A_1 must be a lexical item.

Note that every CFG can be rewritten to comply with the X′-template if one adopts the standard assumption that the empty string can be the head of a phrase. Therefore X′-structure has no effect on weak generative capacity.

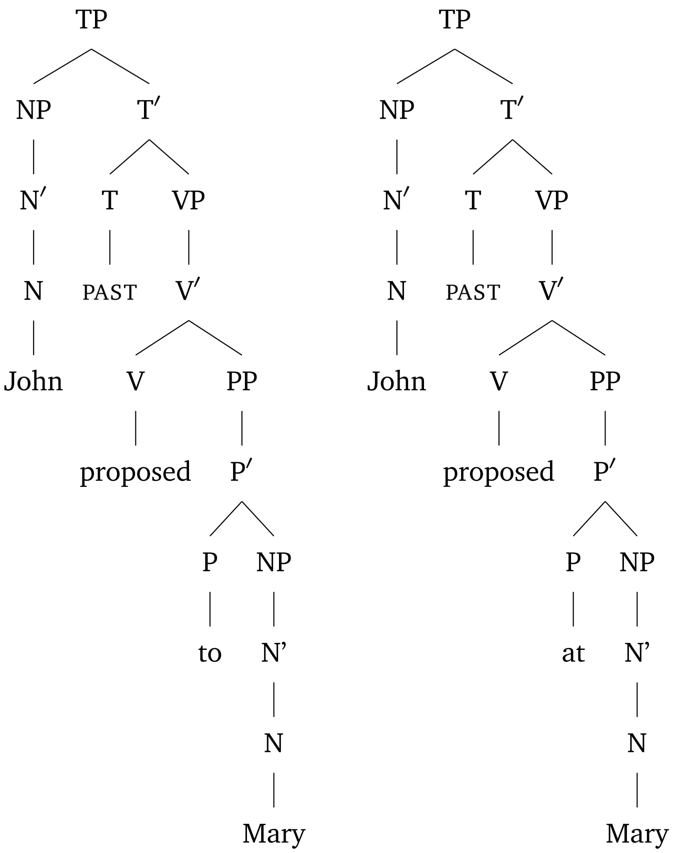
1.2 Subcategorization, Argument Structure, and Lexicalization

Even though X′-theory revolves around the notion of heads and how they regulate structure building, X′-theory by itself is not enough to capture another essential property of language, namely that heads can allow or block the presence of other phrases. More precisely, the head X in an XP controls whether YP and ZP are allowed to be present, and if so, what they must look like. For example, the intransitive verb *left* allows neither ZP nor YP, whereas transitive *kissed* allows YP to be realized as an NP but not as an AP. This property is also known as *subcategorization*, and linguists sometimes speak of a word's *subcategorization frame*.



Sometimes it is not enough to know the general type of YP or ZP and one must also look at the heads of those phrases, as is the case in the following example. Crucially, though, there seem to be no cases in natural language where XP allows a YP depending on what other phrases are contained by the YP.

This is a rather sloppy statement, and under its most literal interpretation it is actually false: long-distance dependencies can indeed prompt behavior where it looks like a head rejects a YP based on what other phrases it contains. A more careful generalization would have to factor out these aspects, which seem to be independent of subcategorization.



With the labeling mechanism of X′-theory, subcategorization is strictly k-local only if the distance between the root of an XP and its head is fixed. This should be the case given everything we have seen so far, where a subtree of depth 3 can always span from XP to the head. A strictly 6-local grammar thus suffices to encode subcategorization — the highest point is XP, the lowest point is the head of YP, and the smallest subtree containing these two nodes has depth 5.

Of course we can also handle subcategorization in CFGs by moving to more elaborate labels. One of the earliest strategies is the slashed category mechanism of *Categorial Grammar* (CG). Instead of simply treating *left* as a verb, it is instead assigned the type *VP* to indicate that it forms a VP all by itself. The transitive verb *kissed*, on the other hand, has the type *VP*/*NP*, which encodes that it must take an NP to its right in order to form a VP. And the past tense head T has the type (*NP**TP*)/*NP*, so that when it is combined with an NP to the right it reduces to *NP**TP*, which returns a TP after it has taken an NP as its left argument.

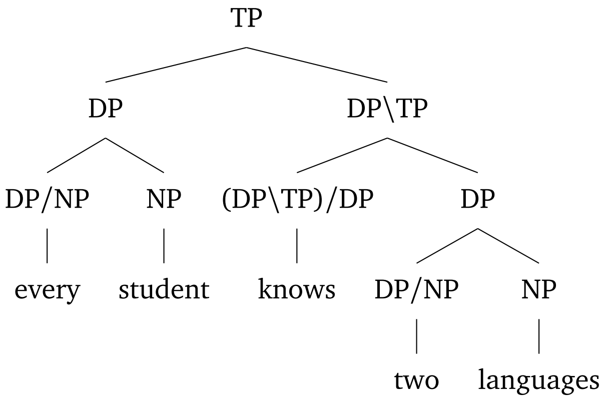
The CG approach has the advantage that it grounds context-free rewriting in an explicit type system. As a result one need not specify a whole rewrite grammar, a general type reduction scheme is enough:

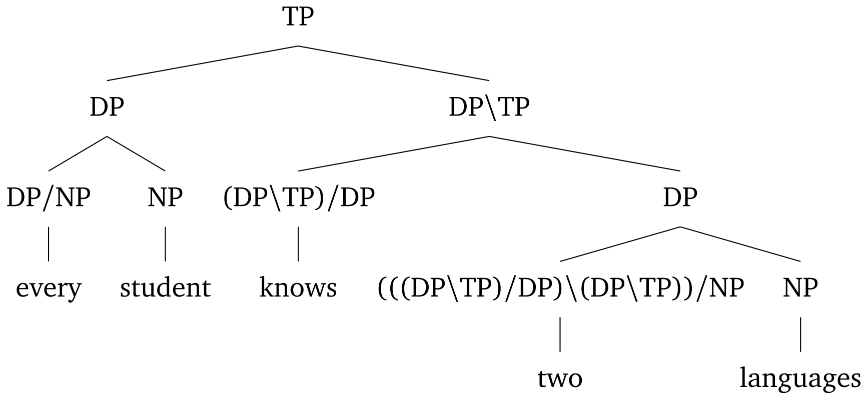
$$(\sigma/\tau) \circ \tau = \tau \circ (\tau\backslash\sigma) = \sigma$$

The language generated by the grammar is thus fully described by the lexicon, which is a list of words and their corresponding types. The CG approach thus *lexicalizes* the grammar. This can be seen as an improvement on X′-theory in the sense that the basic phrasal template is much simpler and the lexicon acts as a centralized storage for all relevant information.

The term *lexicalized grammar* also has a different usage where it refers to a restriction on the distribution of empty heads. We will look at this usage more carefully at a later point.

In addition, the use of a type system makes it easy to read the semantics of a sentence directly off its tree structure. For example, the sentence *Every student speaks two languages* has two readings. In the narrow scope reading, it is the case that every student speaks two languages, although these languages may differ between students. In the wide scope reading, there are two specific languages that are spoken by every student, e.g. Quenya and Klingon. This difference is represented by two trees that share the same geometry but differ in their type assignment.





This higher-order type can either be assigned manually in the lexicon, or the grammar can be equipped with an additional rule to perform type shifts of this kind. As long as the number of types that can be generated this way is finitely bounded, CGs are weakly equivalent to CFGs.

1.3 Features and Agreement

A downside of the CG approach is that the type system must be very fine-grained in order to express the fact that *propose* occurs with PPs headed by *to* but not PPs that are headed by *at*. A natural evolution of the CG system is to modularize the types/non-terminal symbols by turning them into feature bundles, also known as *attribute value matrices* (AVMs).

For example, the lexical item PAST could be associated with a category feature CatV and the subcategorization features Subcat1 and Subcat2, whose respective values are $\langle \text{DP}, r \rangle$ and $\langle \text{DP}, l \rangle$ to indicate that the first argument is a DP to the right of the head and the second one a DP to the left of the head. This information can then be passed around in a local fashion via very general rewrite rules like the ones below (in $M \to M[\alpha]$, M is the result of removing α from the feature matrix):

$$\begin{array}{ll}
M \to l & \text{where } M \text{ is the feature matrix of lexical item } l \\
M \to M[\textsc{Subcat1}\,\langle XP, r\rangle]\; XP & \\
M \to XP\; M[\textsc{Subcat2}\,\langle XP, l\rangle] &
\end{array}$$

Note that these are no longer rewrite rules, but rather meta-rules that encode finite sets of rewrite rules. Only by plugging in the appropriate values for M, l, and XP do we eventually end up with a finite set of context-free rewrite rules. The primary purpose of the meta-rules is to link each feature to syntactic well-formedness conditions.

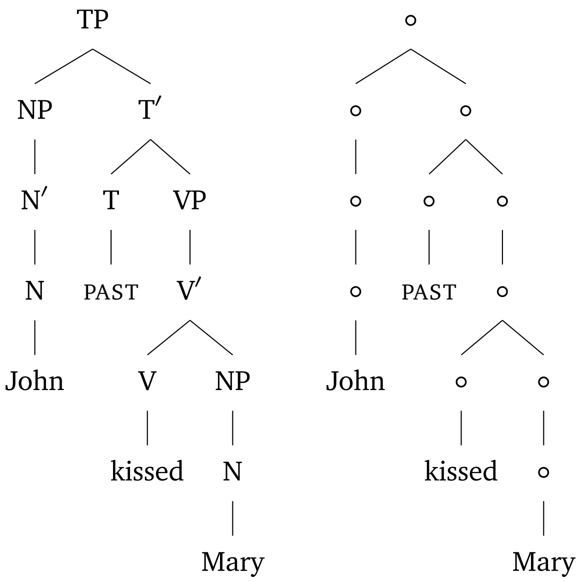
Since features can be added as needed, this system is incredibly flexible and can easily be adapted for various phenomena besides subcategorization. In English, for instance, the finite verb agrees with the subject in person and number. We can capture this by assigning nouns and verbs person and number features whose values must match. This match is enforced by expanding the subcategorization features of the T-head so that they can reference the feature make-up of the arguments. Rather than writing multiple entries for T that differ only in the values of the arguments' person and number features, we may use feature variables.

$$\begin{bmatrix}
\textsc{Phon} & \varepsilon \\
\textsc{Cat} & \mathrm{T} \\
\textsc{Subcat1} & \left\langle \begin{bmatrix} \textsc{Cat} & \mathrm{V} \\ \textsc{Num} & \alpha \\ \textsc{Pers} & \beta \end{bmatrix}, r \right\rangle \\
\textsc{Subcat2} & \left\langle \begin{bmatrix} \textsc{Cat} & \mathrm{N} \\ \textsc{Num} & \alpha \\ \textsc{Pers} & \beta \end{bmatrix}, l \right\rangle
\end{bmatrix}$$

Other common features include tense, gender, animacy, definiteness, finiteness, and TYPE which distinguishes subtypes of common categories, e.g. mass and count nouns. In NLP, it is also useful to add semantic and pragmatic information like instrument, human, that allows for more accurate automatic analysis of sentences like *John hit the man with the black eye* versus *John hit the nail with the hammer*.

In an AVM-based system, the interior nodes of trees are no longer simple labels but complex feature matrices that keep track of all relevant information. This allows them to stay within the boundaries of CFGs, but also introduces a lot of redundancy to the tree structures — for a human, it suffices to know the feature specifications of the heads, this information need not be repeated for every node of the projected phrase. But as already discussed before, the distance between a head and the head of one of its arguments is finitely bounded, so we can omit the AVMs on all interior nodes and simply move from CFGs to some strictly k-local grammar. In fact, all interior node labels are redundant for strictly k-local grammars since every lexical item can be implicitly associated with an AVM (which might involve non-deterministic guessing in case a word has multiple entries in the lexicon), and the AVM fully determines the syntactic behavior of the word.

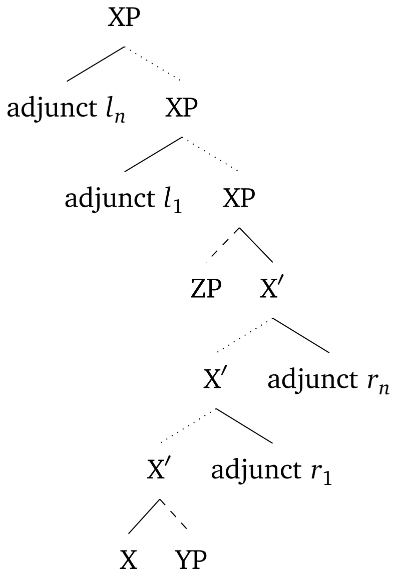
So strictly k-local grammars provide a more minimal labeling algorithm than X$'$-theory (or any of its recent revisions) while doing a lot more work since they also implicitly handle subcategorization and feature matching.



1.4 Adjunction

So far we have seen that three central linguistic properties — headedness, subcategorization, and feature matching — can be enforced by strictly *k*-local grammars without refining the labels (quite the opposite, interior node labels can be completely omitted). But a crucial assumption we made is that these relations are restricted to heads and their arguments, and that the size of a phrase is finitely bounded. At least the former, though, is likely to be wrong due to the unbounded nature of *adjunction*.

Adjuncts are elements that attach to a phrase without being explicitly licensed by the head of the phrase. They are usually optional and iterable. Adjectives are prototypical NP-adjuncts, while adverbs are adjuncts of verbs, adjectives, and other adverbs. In X′-theory, adjuncts occupy positions where their mother and their sibling have the same label.



An alternative analysis, which originated in CG, treats adjuncts as category-preserving heads. Under this view an adjunct is a phrase of category X that selects a phrase of category X. In CG terms, it is of type σ/σ or $\sigma\backslash\sigma$.

For our purposes it doesn't really matter which analysis one adopts, as the problems for strictly local grammars stay the same. Suppose that head h can select an argument XP with head h_a, but not with head h'_a. Then we can add XP-adjuncts to XP until the distance between h and $h_a\backslash h'_a$ exceeds k. Unless the adjuncts provide sufficient information to deduce whether the head of XP is h or h_a, the grammar either allows both or neither. In both cases the grammar no longer generates the intended tree language.

This brief argument already makes it clear that adjuncts pose no challenge if we use elaborate AVMs as interior node labels, or equivalently, if we use a refined strictly local grammar to keep track of non-local information via states. But in the absence of such local coding mechanisms, they invariably render local syntactic dependencies non-local.

1.5 Displacement

Slash feature percolation from GPSG (Gazdar et al. 1985)

Movement dependencies

The fallacious argument for punctuated paths (Abels 2003)

2 Against Tree Language Dogmatism

The choice of tree structures is one of the most contentious issues in syntactic theory. Debates along these lines tend to revolve around two independent factors that unfortunately are often conflated. One pertains to what one might call the macro-constituency of the sentence, i.e. how the major phrases are positioned with respect to each other. This is the domain of constituency tests, locality effects, and so on. The other one focuses on how the macro-constituency is encoded in the tree, and it is dominated by the discussion of branching factors, labels, features, and other highly technical topics with much more abstract evaluation criteria. While syntactic formalisms mostly agree on the former (although tests are sometimes called into doubt and locality effects may be attributed to extra-grammatical factors), there is tremendous variation regarding the latter. The preceding discussion sheds some light on why that is the case.

CFGs, strictly local tree grammars and refined tree grammars define exactly the same tree languages *modulo* relabelings. Node labels can be altered as needed to fit a tree language into the desired level of expressivity. At the same time, there are no syntactic tests to determine the label of a node — whatever test one may concoct, the behavior of formalism A under such a test can also be emulated with formalism B. The observed behavior might appear more natural or plausible under formalism A, but this is not conclusive proof as it may indicate not so much a shortcoming of formalism B as a gap in our understanding of its internals. The same goes for psycholinguistic and neurolinguistic experiments. The alternatives under discussion are so close to being notational variants that a unique winner will arguably never be determined.

This is no reason for despair though — quite the opposite. Rather than trying to discern a unique correct formalism for tree languages, we should take advantage of the fact that it is so easy to translate between the different formats. Just like it causes us little concern that the computational object of a set has multiple algorithmic instantiations in Python in the form of sets, lists, and dictionaries, each one of them serving a specific purpose, we should embrace the diversity of tree languages and continuously switch between these different perspectives to further our goals.

Questions of computational complexity are best addressed by focusing on tree languages without interior node labels, where the differences between local and unbounded dependencies surface most clearly. Strictly 2-local representations with rich interior labels do away with this distinction and thus are best suited for areas where directly handling non-local dependencies is likely to create a lot of extra work with little foreseeable pay-off. Depending on one's goals, parsing is one such area, as we will see next.

Relevant literature for Unit 14

add more info

Abels, Klaus. 2003. *Successive cyclicity, anti-locality, and adposition stranding*. University of Connecticut dissertation.

Chomsky, Noam. 1993. A Minimalist program for linguistic theory. In Kenneth Hale & Samuel Jay Keyser (eds.), *The view from building 20*, 1–52. Cambridge, MA: MIT Press.

Gazdar, Gerald et al. 1985. *Generalized phrase structure grammar*. Oxford: Blackwell.

Jackendoff, Ray. 1977. *X-bar syntax: a study of phrase structure*. Cambridge, MA: MIT Press.

Kornai, Andras & Geoffrey K. Pullum. 1990. The X-bar theory of phrase structure. *Language* 66(1). 24–50.

Unit 15

Syntactic Parsing

In the previous section we concluded that the choice for or against a specific type of tree language cannot be made a vacuum. Since regular tree languages can always be translated into strictly local ones, and *vice versa*, we cannot proclaim one of them to be more cognitively real than the other without putting a very elaborate scaffolding of additional assumptions in place. At this point, we do not know enough about cognition to pick a specific set of assumptions over another, and odds are that in order to distinguish between regular and strictly local tree languages, we would need assumptions that are so specific and fine-grained that they could never be fully backed up via empirical evidence.

A more fruitful strategy is to accept this indeterminacy and make good use of it. This can take a negative form by no longer expanding time and resources on fruitless efforts to settle this issue, but also a positive one where we switch between these types of tree languages depending on which one serves our purposes best. Today we will see how strictly local tree languages provide a simple format for designing parsers.

1 Intersection Parsing

On a computational level (in Marr's sense), a parser is device that infers the tree structures that a grammar assigns to a given string. We can make this more precise via the function yd, which maps every tree to its string yield: a parser is device that takes a grammar G and a string $s \in \Sigma^*$ as input and computes $\text{yd}^{-1}(s)$ with respect to G. Formally, this amounts to computing $\text{yd}^{-1}(\{s\} \cap L^s(G))$.

Conceptualizing parsing in terms of intersection is very elegant as it reduces parsing to basic operations on languages that we are already familiar with. In particular, if one can construct a grammar G' whose string language is just s, and that assigns exactly the same trees to s as G, then $\text{yd}^{-1}(\{s\} \cap L^s(G))$ is just the tree language generated by G'. For CFG, such a G' is guaranteed to exist because singleton languages are regular, and CFLs are closed under intersection with regular languages.

Theorem 15.1. Let L_C and L_R be a context-free and a regular string language, respectively. Then $L_C \cap L_R$ is context-free. ⌟

Proof. Our proof relies on three insights:

- L_C is generated by some CFG $G := \{\Sigma_G, S, R\}$.
- L_R can be represented by a deterministic FSA $A := \{\Sigma_A, Q, q_0, F, \Delta\}$.

- The states of A can be incorporated into the alphabet of G, yielding a grammar G_A that executes both G and A simultaneously.

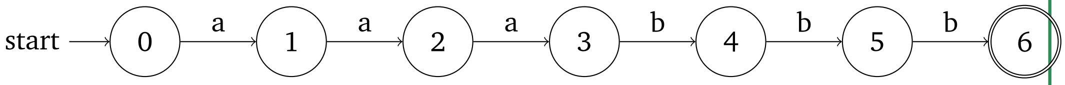
The construction proceeds by prefixing every non-terminal N with the state that A assigns to the leftmost symbol in its substring, and similarly suffixing N by the state of the rightmost symbol. For the sake of simplicity we assume that G is strictly binary branching and ε-free (which does not decrease generative capacity), but the proof is easily adapted to the more general case.

We first add a rule $S \rightarrow {}_{q_0}S_{q_f}$ to G_A for every q_f that is the final state of some accepting run of A over some $s \in L_R$. Given a rewrite rule $X \rightarrow Y\ Z$ and some other rule with ${}_pX_q$ on the righthand side, we add ${}_pX_q \rightarrow {}_pY_u\ {}_uZ_q$ to G_A, where u is some state such that for some run over a string spanned by X that starts with p and ends in q I) the last symbol of the substring spanned by Y ends in u, and II) the first symbol of the substring spanned by Z starts with u. We do the same if B or C are terminal symbols instead, except that they are not prefixed and suffixed with states.

Showing the correctness of the procedure is left as an exercise to the reader. □

Example 15.1

Let G be the CFG with the rewrite rule $S \rightarrow Aa|SB$, $A \rightarrow aa$, $B \rightarrow bbb$ and s the string $aaabbb$, which is recognized by the deterministic automaton A below.



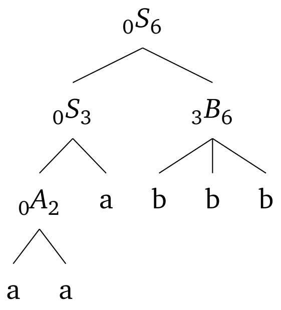
In order to determine what trees G assigns to s, we construct a refined CFG G_A according to the instructions above.

First, we add the rewrite rule $S \rightarrow {}_0S_6$, as 6 is the only final state that A assigns to $aaabbb$ and there are no other strings to consider. Since we now have a refined symbol ${}_0S_6$ we have to take care of all the rewrite rules with S on the left hand side. We add the rule ${}_0S_6 \rightarrow {}_0S_3\ {}_3B_6$ since B spans a substring bbb, which only occurs between the states 3 and 6 in A's run over s. We cannot find any valid state subscripts for A in ${}_0S_6 \rightarrow Aa$ since no string in $L(a)$ allows us to reach state 6 via an a-transition. Consequently, we cannot add a refined rule in this case and move on to the rules for ${}_0S_3$. In this case, the only valid refinement is ${}_0S_3 \rightarrow {}_0A_2a$.

The full set of rewrite rules is given below.

$$\begin{aligned} {}_0S_6 &\rightarrow {}_0S_3\ {}_3B_6 \\ {}_0S_3 &\rightarrow {}_0A_2a \\ {}_0A_2 &\rightarrow aa \\ {}_3B_6 &\rightarrow bbb \end{aligned}$$

It is easy to see that G_A generates only s and assigns it the same tree as G, *modulo* subscripts.



What properties must hold of a CFG so that it only assigns a finite number of trees to a string?

As long as the refined grammar assigns only a finite number of trees to *s* (which can be guaranteed via two simple restrictions), this set can be generated by simply applying all rewrite rules in all possible orders. So our intersection approach has indeed succeeded at reducing parsing to generation with a refined grammar.

The procedure is stated for CFGs, but of course we can extend it to regular tree languages. First we make the hidden alphabet part of the visible alphabet, which turns the refined strictly 2-local tree grammar into a CFG (we might have to change a few labels to establish a clear distinction between terminal and non-terminal symbols). This CFG is then refined, and once the set of trees for the input string has been determined, it is lifted back into a regular tree language via a projection that removes the subscripts and those parts of the label that were originally part of the hidden alphabet. For strictly *k*-local tree languages, we simply construct the corresponding strictly 2-local tree grammar and proceed as before. The fact that we can freely translate between these types of tree languages and use CFGs as a baseline for the refinement construction is what makes intersection parsing applicable across the board.

2 Top-Down Parsing

2.1 Definition

Elegant as intersection parsing may be from a mathematical perspective, it is not a realistic model of human sentence processing. Its major shortcoming is that the full input string must be known before one can even get started on the grammar refinement. Human parsing does not work like this, listeners begin inferring structures as soon as they hear the first word, and there is plenty of evidence that they even build structure before the corresponding part of the string has been encountered. These are two essential properties of human parsing: it is *incremental* and *predictive*.

We need a more procedural model in order to capture these properties. One of the simplest and best-known models is top-down parsing. In fact, it is so intuitive that syntax students, for example, have a tendency to automatically use this algorithm when asked to determine if a grammar generates a given sentence. That's because top-down parsing is still very close to the idea of CFGs as top-down generators.

Suppose you are given the following CFG.

1)	S	→ NP VP	6)	Det	→ a | the
2)	NP	→ PN	7)	N	→ car | truck | anvil
3)	NP	→ Det N	8)	PN	→ Bugs | Daffy
4)	VP	→ Vi	9)	Vi	→ fell over
5)	VP	→ Vt NP	10)	Vt	→ hit

When asked to show that this grammar generates the sentence *The anvil hit Daffy*, you might draw a tree. But a different method is to provide a tabular depiction of the rewrite process.

string	rule
S	start
NP VP	S → NP VP
Det N VP	NP → Det N
the N VP	Det → the
the anvil VP	N → anvil
the anvil Vt NP	VP → Vt NP
the anvil hit NP	Vt → hit
the anvil hit PN	NP → PN
the anvil hit Daffy	PN → Daffy

Of course the rewrite rules could also be applied in other orders.

string	rule
S	start
NP VP	S → NP VP
NP Vt NP	VP → Vt NP
NP Vt PN	NP → PN
NP Vt Daffy	PN → Daffy
NP hit Daffy	Vt → hit
Det N hit Daffy	NP → Det N
Det anvil hit Daffy	N → anvil
the anvil hit Daffy	Det → the

string	rule
S	start
NP VP	S → NP VP
Det N VP	NP → Det N
Det N Vt NP	VP → Vt NP
the N Vt NP	Det → the
the anvil Vt NP	N → anvil
the anvil hit NP	Vt → hit
the anvil hit PN	NP → PN
the anvil hit Daffy	PN → Daffy

In all three cases we proceed top-down: non-terminals are replaced by a string of terminals and/or non-terminals. From the perspective of phrase structure trees, the trees are growing from the root towards the leafs. The difference between the three tables is the order in which non-terminals are rewritten.

- Table 1: depth-first, left-to-right
- Table 2: depth-first, right-to-left
- Table 3: breadth-first, left-to-right

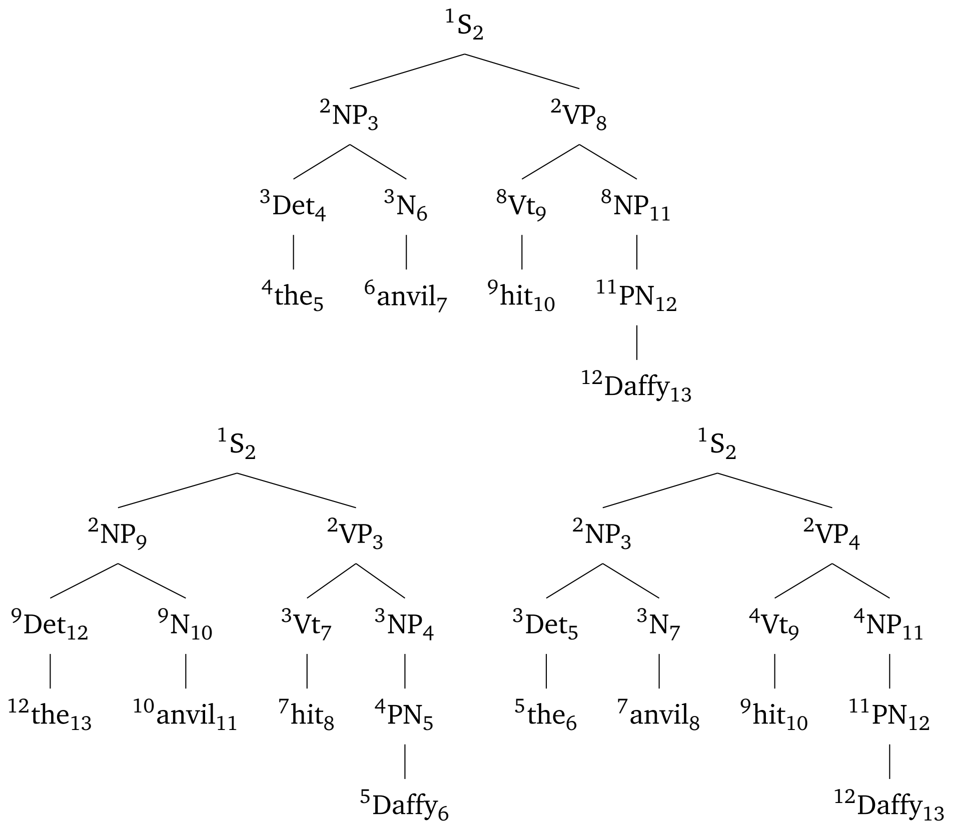
depth-first rewrite some symbol that was introduced during the previous rewriting step

breadth-first before rewriting a symbol introduced during rewriting step j, all symbols that were introduced at rewriting step i must have been rewritten, for every $i < j$

left-to-right if several symbols are eligible to be rewritten, rewrite the leftmost one

right-to-left if several symbols are eligible to be rewritten, rewrite the rightmost one

We can visualize the differences between these strategies by annotating phrase structure trees with indices to indicate when a symbol is first introduced (prefix) and when it is rewritten (suffix). For terminal symbols, which are never rewritten, we stipulate that the suffix is one higher than the prefix.



The prototypical top-down parser operates depth-first and left-to-right, and is also known as a *recursive descent parser*. It's behavior can be stated very succinctly in terms of logical inference rules. This idea was originally called *parsing as deduction* (Pereira & Warren 1983, Shieber, Schabes & Pereira 1995) and was later refined by Sikkel (1997). Parsing as deduction made it possible to abstract away from implementation-heavy concepts like data structures and memory management in order to state the basic parsing procedure as clearly as possible. It is also the foundation of *semiring parsing* (Goodman 1999), which provides an abstract perspective that unifies various parsers similar to how our monoid perspective in chapter 7 unified a variety of scanners (in fact, a semiring is an algebraic object that combines two monoids in a particular fashion).

The logical inference rules take the form of *parse items* $[i, \beta]$, where

- $\beta \in \Sigma^*$ is a string of terminal and/or non-terminal symbols, and
- i is a position in the string.

The parser always starts with the *axiom* $[0, S]$, which represents the assumption that the input string can be obtained by rewriting the start symbol S. The parser accepts a string iff it can use its inference rules from the axiom to the *goal* item $[n,]$, where n is the end of the input string. The inference rules are as follows:

$$\textbf{Scan} \qquad \frac{[i, a\beta]}{[i+1, \beta]}\, a = w_i$$

$$\textbf{Predict} \qquad \frac{[i, N\beta]}{[i, \gamma\beta]}\, N \rightarrow \gamma \in R$$

The scan rule removes a terminal symbol from a parse item if it matches the symbol in the input at the corresponding position. The predict rule expands non-terminal nodes according to the grammar.

Example 15.2 Top-down parse of *The anvil hit Daffy*

Let G be the grammar at the beginning of the handout. Then a depth-first, left-to-right parse of *The anvil hit Daffy* will proceed as follows, where predict(n) denotes a prediction step using rule n.

parse item	**inference rule**
[0,S]	axiom
[0,NP VP]	predict(1)
[0,Det N VP]	predict(3)
[0,the N VP]	predict(6)
[1,N VP]	scan
[1,truck VP]	predict(7)
[2,VP]	scan
[2,Vt NP]	predict(5)
[2,hit NP]	predict(10)
[3,NP]	scan
[3,PN]	predict(2)
[3,Daffy]	predict(8)
[4,]	scan

Note that the table above only shows the prediction steps leading to a successful parse. The parser, on the other hand, applies every possible prediction step. So from [0,NP VP,4], for instance, the parser not only predicts [0,Det N VP,4] via rule 3 but also [0,PN VP,4] via rule 2 since both can be applied to NP.

Without some kind of additional memory, the parser does not actually keep track of the tree structure and merely acts as a recognizer. But it is fairly simple to assemble the correct tree from the table above as all the needed information is contained in the application of the predict steps.

2.2 Psycholinguistic Predictions

The recursive descent parser is clearly incremental, as it can start parsing before we have seen a single input symbol. This also shows that it is predictive. In fact, there is no such thing as an non-predictive top-down parser seeing how the parser must build a subtree before it can even start scanning the leafs of the subtree. But with a few ancillary assumptions, the recursive descent parser can also explain certain shortcomings of human sentence processing such as garden-path effects and center-embedding. Let us take a closer look at the former.

Garden path effects were one of the first phenomena to be discussed in the psycholinguistic literature. They arise with sentences that are grammatically well-formed but nonetheless difficult to parse (Frazier 1979, Frazier & Rayner 1982). More precisely, a garden path sentence is a sentence w that up to some w_i has a strongly preferred analysis that must be discarded at w_{i+1}. Here's a list of examples:

(1) *Structural ambiguity*
   a. The horse raced past the barn fell.
   b. The raft floated down the river sank.
   c. The player tossed a Frisbee smiled.
   d. The doctor sent for the patient arrived.
   e. The cotton clothing is made of grows in Mississippi.
   f. Fat people eat accumulates.
   g. I convinced her children are noisy.

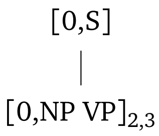
(2) *Lexical ambiguity*
   a. The old train the young.
   b. The old man the boat.
   c. Until the police arrest the drug dealers control the street.
   d. The dog that I had really loved bones.
   e. The man who hunts ducks out on weekends.

Frazier (1979) proposes to treat garden path effects as a result of reanalysis: the parser is forced to abandon its current set of hypotheses, backtrack to an earlier point, and build a new structure from there. If for some reason this process proves too difficult, the parser gets stuck and assigns no structure at all. Let's try to make Frazier's account precise.

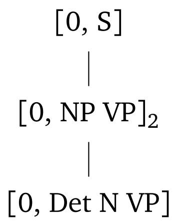
First, we will use prefix trees to represent the parse history, i.e. a record of how the parser built a parse table, rejected it at some point, and moved on to the next one.

Example 15.3 Parse Tables as Trees

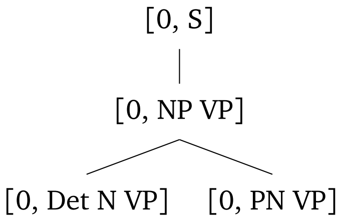
Suppose we are once more using the grammar above to parse *the anvil hit Daffy* with a recursive descent parser. Our parse table starts with [0,S] as usual, from which we can only predict [0,NP VP]. This can be represented as the tree below.



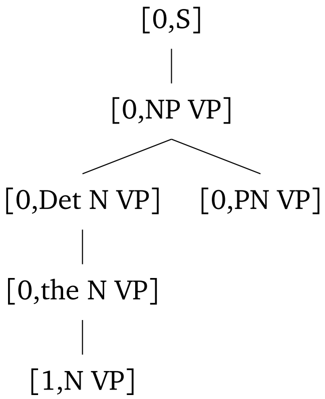
The node [0, NP VP] has subscripts 2 and 3 to indicate that we can use rules 2 and 3 of our CFG to predict new parse items. Suppose the parser uses predict(3) to create the item [0, Det N VP]. Then this item is added as a daughter of [0, NP VP] to the previous tree, and the subscript 3 is also removed from [0, NP VP]. We do not need to add a subscript to [0, Det N VP] since it is a leaf.



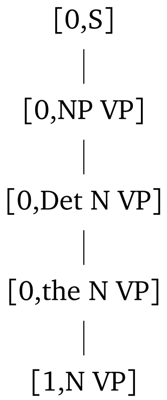
The next step of the parser now depends on how distinct parses are prioritized. We can either expand the table that ends in [0, Det N VP], or go back to the one ending in [0, NP VP] and apply predict(2). Suppose we do the latter, so that [0, PN VP] is added as the second daughter of [0, NP VP], which thus loses its last subscript.



Now let's expand [0, Det N VP] again to obtain [0, the N VP] and then apply a scan step, yielding [1, N VP].



If our parser is really smart, it will be able to infer at this point that the scanned word *the* can never be obtained from [0,PN VP] (technically this is achieved by associating every parse item p with a regular expression that describes the possible left edges of the strings that can be derived from p). So the parser can remove [0,PN VP] from the tree, which is tantamount to discarding the parse table where NP was rewritten as PN.



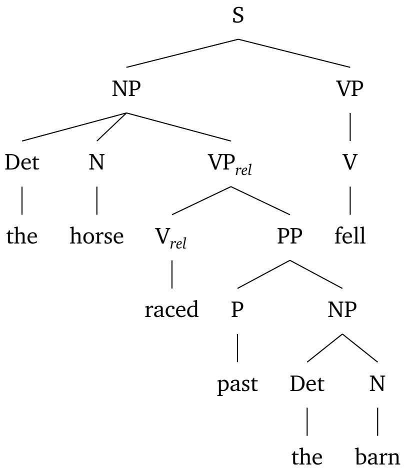
What makes the prefix tree representation of parse tables appealing is that the construction and prioritization of parse tables can be reduced to strategies for tree building. The kind of serial parsing envisioned by Frazier corresponds to a depth-first strategy where the parser always builds a single complete parse history rather than multiple partial ones. If the parse history cannot be expanded anymore, the parser either stops (successful parse) or backtracks to the last choice point in the parse history and tries a different choice instead. With such a strategy, garden path effects are readily explained by the amount of attempts it takes to find a successful parse.

Example 15.4 Backtracking in *the horse raced past the barn fell*

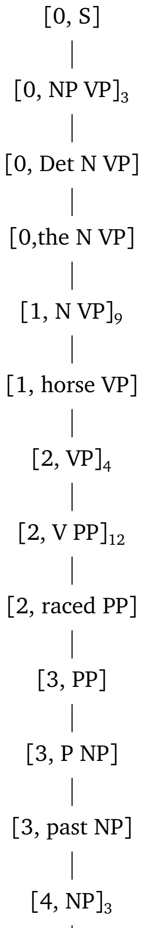
Assume we have the following (massively simplified) grammar:

1)	S	→ NP VP	8)	Det	→ the
2)	NP	→ Det N	9)	N	→ barn
3)	NP	→ Det N VP_{rel}	10)	N	→ horse
4)	VP	→ V	11)	P	→ past
5)	VP	→ V PP	12)	V	→ fell
6)	VP_{rel}	→ V_{rel} PP	13)	V	→ raced
7)	PP	→ P NP	14)	V_{rel}	→ raced

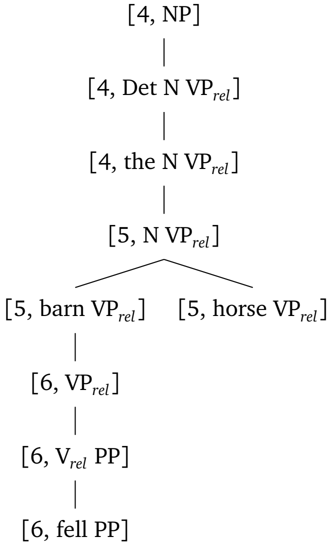
Here's the resulting phrase structure tree for our garden path sentence.



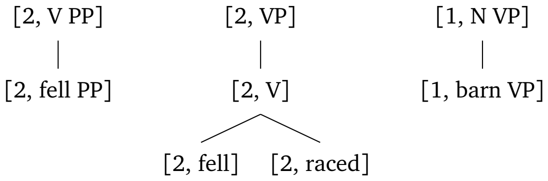
If the parser prefers NP → Det N over NP → Det N VP_{rel} and operates in a recursive descent fashion, the construction of the first parse table results in the prefix tree below.



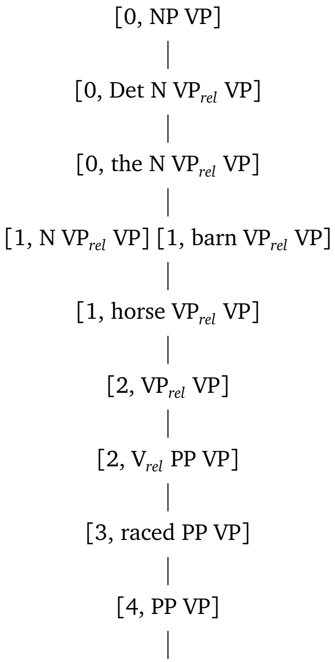
Since this parse does not succeed, the parser needs to backtrack. The closest choice point is [5,N]$_{10}$, which obviously does not fix the problem of integrating *fell* into the structure, as can be verified after a single scan step. The next choice point is [4,NP]$_3$. Here the parser still has the option of replacing NP by Det N CP, which won't help much either, but it takes quite a while to realize this because the tree for the parse table obtained by following this option involves a choice point, too.


[4, NP]
[4, Det N VP_{rel}]
[4, the N VP_{rel}]
[5, N VP_{rel}]
[5, barn VP_{rel}]
[5, horse VP_{rel}]
[6, VP_{rel}]
[6, V_{rel} PP]
[6, fell PP]

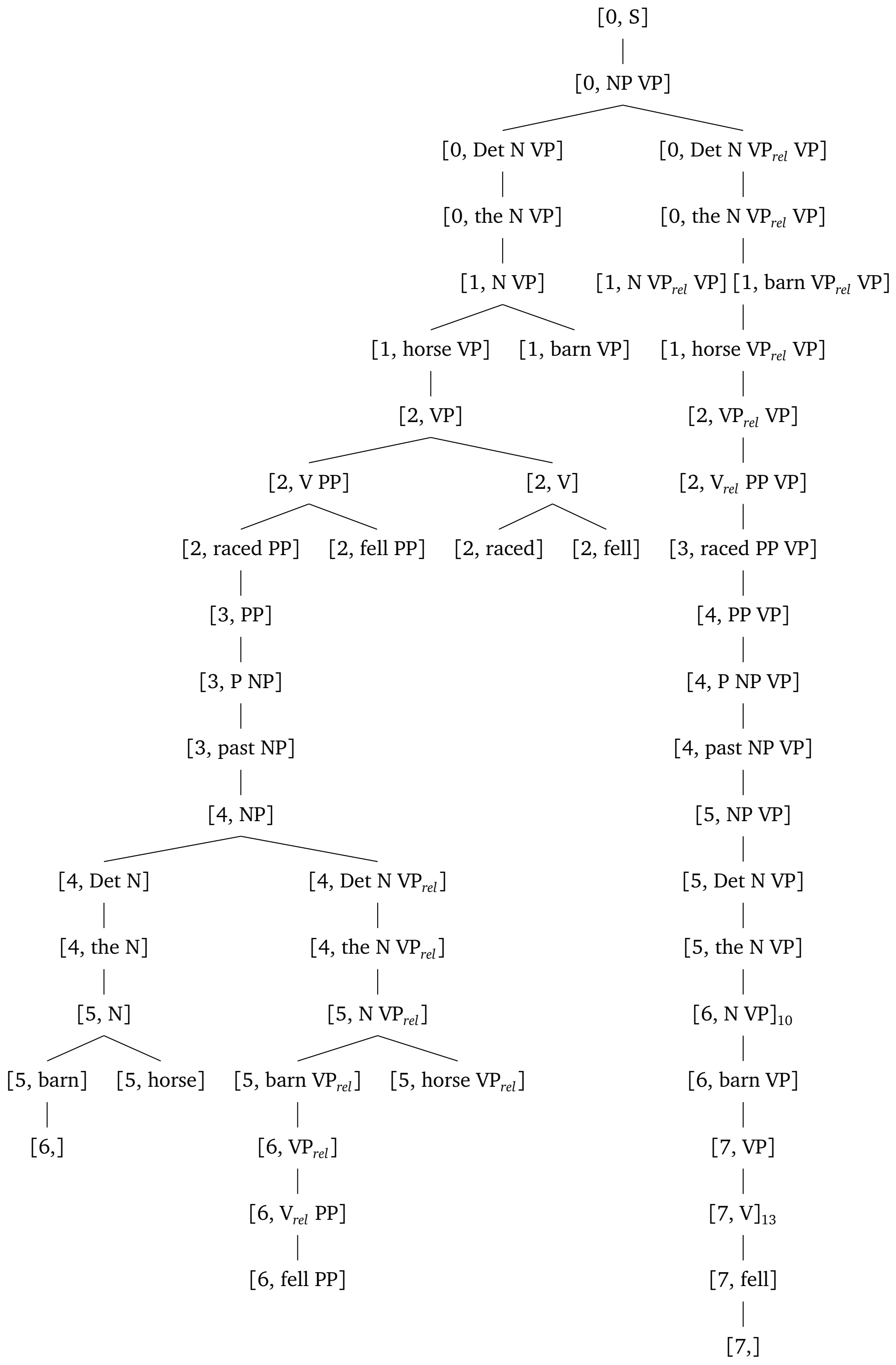
Since this venue didn't yield a successful parse either, the parse backtracks to [2,V PP]$_{12}$, and after this fails, to [2,VP]$_4$. Once again it is not successful, and the same holds once it expands [1,N VP].



Only if the parser backtracks all the way to [0, NP VP]$_3$, essentially undoing all its work so far, can it find a working parse.


[0, NP VP]
[0, Det N VP_{rel} VP]
[0, the N VP_{rel} VP]
[1, N VP_{rel} VP] [1, barn VP_{rel} VP]
[1, horse VP_{rel} VP]
[2, VP_{rel} VP]
[2, V_{rel} PP VP]
[3, raced PP VP]
[4, PP VP]

The prefix tree for all the parse histories built by the parser before it encounters a successful parse it much bigger than the simple phrase structure tree for the sentence.



Relevant literature for Unit 15

add more info

Frazier, Lyn. 1979. *On comprehending sentences: syntactic parsing strategies*. University of Connecticut dissertation.

Frazier, Lyn & Keith Rayner. 1982. Making and correcting errors during sentence comprehension: eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology* 14. 178–210.

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics* 25. 573–605. http://www.aclweb.org/anthology/J99-4004.

Pereira, Fernando C. N. & David Warren. 1983. Parsing as deduction. In *21st annual meeting of the association for computational linguistics*, 137–144.

Shieber, Stuart M., Yves Schabes & Fernando C. Pereira. 1995. Principles and implementations of deductive parsing. *Journal of Logic Programming* 24. 3–36.

Sikkel, Klaas. 1997. *Parsing schemata* (Texts in Theoretical Computer Science). Berlin: Springer.

Unit 16

Syntax as First-Order Logic

We are now operating under the assumption that syntax can be adequately described in terms of regular tree languages. They are sufficiently expressive to capture advanced linguistic concepts like headedness, projection, subcategorization, agreement, adjunction, and instances of displacement like wh-movement and topicalization in English. At the same time, they can be automatically converted into CFGs, which makes them suitable for a variety of well-understood parsing techniques. Today we will look once more at what can and cannot be done with regular tree languages, and we will see that, puzzlingly, syntax is both within and outside this class, depending on what kind of object we take syntax to define.

1 Syntactic Constraints

1.1 A General Template for Constraints

The syntactic literature is full of an enormous number of constraints. Some examples include:

- the Empty Category Principle (ECP), which restricts the possible landing sites of movement,
- Binding theory, which restricts the distribution of pronouns (*him*, *her*) and reflexives (*himself*, *herself*),
- the Person Case Constraint (PCC), which blocks certain combinations of pronouns in a sentence,
- the Shortest Derivation Principle (SDP), which favors shorter derivations over longer ones.

This is obviously not a complete list, and linguists keep coming up with new constraints or modify old ones to better fit the data. How could we possibly say that all constraints in natural language syntax fall within the class of regular tree languages?

As with any other empirical science, there is of course the possibility that a new discovery completely contradicts our current theories and requires new, more powerful machinery. But with any new piece of data we discover that matches current proposals, this scenario becomes less and less likely. After 50 years of generative linguistics with many competing formalisms, an abstract template for the formulation of constraints has crystallized. Constraints that are stated with respect to a single tree usually involves four components:

- a finite number of nodes between which the dependency holds (e.g. the target site and the source of a moving phrase),
- a tree-geometric relation that picks out these nodes (e.g. c-command)
- a locality domain within which the dependency must be satisfied,
- a logical control mechanism that triggers the dependency (e.g. "if X is in position Y, then Z must be satisfied")

Comparative constraints like the SDP, where the well-formedness of a tree cannot be decided without looking at other trees, do not obviously fit this template, and we will not discuss them here. However, Graf (2013) shows that they, too, can be broken down into templates of this form (using ideas we encountered in the proof that certain variants of OT generate regular string languages).

The existence of such a template for constraints is crucial because if all four factors stay within the realm of regular tree languages, then all constraints that obey this template are regular, too. In the early 90s it was realized that constraints obeying the template can be easily expressed as statements of a formal description language that can be automatically translated into refined strictly 2-local tree grammars, thereby establishing their regularity. This insight forms the backbone of what is now known as *model-theoretic syntax* (Blackburn, Gardent & Meyer-Viol 1993, Backofen, Rogers & Vijay-Shanker 1995, Kracht 1997, Rogers 1998, Potts & Pullum 2002, Pullum 2007).

1.2 Constraints, Logics, and Model Theory

If one abstracts away from all matters of implementation, a constraint c over trees is simply a method for defining the largest set of trees that satisfy c, or equivalently, do not violate c. So we can equate every constraint with a (possibly infinite) set of trees.

Something remarkably similar can be found in mathematical logic. There are many different logics, e.g. propositional logic, first-order logic, or modal logic. What makes each one of them a logic is that they are formal systems with a well-defined syntax, which defines what strings are well-formed formulas of the logic, and a semantics that assigns a specific interpretation to each formula.

For first-order logic, the syntax can be defined in a way that's very close to a context-free grammar. First, we have to fix a vocabulary, also called a *signature*. It includes a finite number of relational symbols R_i^n of arity n, and a set O of logical operands:

$\wedge$	and
$\vee$	or
$\rightarrow$	implies
$\leftrightarrow$	if and only if
$\neg$	not
$\forall$	for every
$\exists$	there exists

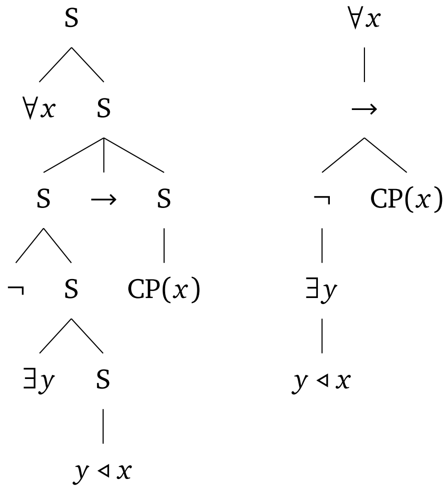
In addition, there is an infinite set of variables $x_1, \ldots, x_n, \ldots$, and the bracketing symbols "(" and ")". The set of well-formed formulas is given by the following rules:

$$\begin{array}{lcl}
S & \rightarrow & (S \land S) \\
S & \rightarrow & (S \lor S) \\
S & \rightarrow & (S \rightarrow S) \\
S & \rightarrow & (S \leftrightarrow S) \\
S & \rightarrow & (\neg S) \\
S & \rightarrow & (\forall x_i \; S) \\
S & \rightarrow & (\exists x_i \; S) \\
S & \rightarrow & R^n_i(x_1, \ldots, x_n)
\end{array}$$

Assuming that our only relational symbols are binary $\lhd$ and unary CP, the rules above tell us that the following is a well-formed formula:

$$(\forall x((\neg(\exists y(\lhd(y,x)))) \rightarrow \mathrm{CP}(x))$$

Assuming certain standard conventions about operator scope switching to infix notation for binary relations, we can drop some of the brackets and simplify the formula to $\forall x(\neg\exists y(y \lhd x) \rightarrow \mathrm{CP}(x))$, and we can also represent it in terms of a tree (the left format shows the phrase structure, the right one the dependency structure).



Defining the set of well-formed formulas gives us only one half of a logic, the much more important half is to endow each well-formed formula with a specific meaning. This is achieved by the notion of a *model*. A mathematical structure M is a model of formula ϕ (written $M \models \phi$) iff ϕ holds in M under a fixed interpretation. This interpretation is defined in a piece-wise fashion, where $\phi[x \leftarrow u]$ is the result of replacing every free instance of x (= not c-commanded/dominated by $\forall x$ or $\exists x$ in the phrase structure/dependency tree) in ϕ by u.

$$\begin{array}{ll}
M \models \phi \land \psi & \text{iff both } M \models \phi \text{ and } M \models \psi \\
M \models \phi \lor \psi & \text{iff } M \models \phi \text{ or } M \models \psi \\
M \models \phi \rightarrow \psi & \text{iff } M \models \phi \text{ implies } M \models \psi \\
M \models \phi \leftrightarrow \psi & \text{iff } M \models \phi \text{ implies } M \models \psi \text{ and the other way round} \\
M \models \neg\phi & \text{iff } M \models \phi \text{ does not hold} \\
M \models \forall x \phi & \text{iff } M \models \phi[x \leftarrow u] \text{ for all } u \text{ of } M \\
M \models \exists x \phi & \text{iff } M \models \phi[x \leftarrow u] \text{ for some } u \text{ of } M
\end{array}$$

In addition, one must also define an interpretation for all relational symbols. For example, if we assume that our class of models is the set of all trees, we may take $\lhd$ to denote the mother-of relation, whereas CP(x) means that node x has the label CP. In that case, the formula above holds of all trees whose root is labeled CP. Or the other way round, every tree whose root is labeled CP is a model for this formula.

Just like every constraint can be identified with the set of trees that satisfy it, every formula can be identified with its set of models. But this means that we can actually think of constraints in terms of logical formulas: a constraint c is a logical formula ϕ such that the set of structures obeying c is exactly the set of models of ϕ.

Example 16.1 A Formula for Trace Licensing

The ECP states that every trace must be c-commanded by the moving phrase that left behind this trace. We will formalize a simplified variant that ensures the presence of a c-commanding phrase that could be the original mover. To this end, we use first-order logic with a signature that contains $\triangleleft^+$ for proper dominance, $\approx$ for equivalence, and a unary predicate for every label in our intended tree alphabet. Note that we do not have to add the mother-of relation to the signature because it can be defined in terms of proper dominance:

$$x \triangleleft y \iff x \triangleleft^+ y \wedge \neg\exists z[x \triangleleft^+ z \wedge z \triangleleft^+ y]$$

The same is true of c-command:

$$\text{c-com}(x, y) \iff \neg(x \triangleleft^* y) \wedge \neg(y \triangleleft^* x) \wedge \forall z[z \triangleleft^+ x \rightarrow z \triangleleft^+ y]$$

Here $\triangleleft^*$ denotes reflexive dominance, which is also definable from proper dominance and equivalence.

$$x \triangleleft^* y \iff x \approx y \vee x \triangleleft^+ y$$

So now we have all the predicates we need to talk about the tree geometric configuration between traces and movers, but we still need a way to identify potential movers.

Clearly a phrase is a mover if it occupies a position where it cannot have been selected as an argument. According to the standard theory of movement, movers can only occur in specifiers, which are usually reserved for the second argument of a head. So it suffices to look at the head of the phrase and check whether it selects a second argument (for the sake of simplicity, we will assume that no head has an optional second argument, that phrases obey a strict X′-template where every phrase has at most one specifier, and we also ignore adjuncts, which would be identified as potential movers with this approach).

First we need to define a predicate that identifies whether a node is a specifier. A phrase is a specifier iff it is the sibling of a node labeled X′. Rather than assuming that the trees include this information in their interior node labels, we define a predicate for identifying the X′-node projected by a lexical item, which is then referenced in the definition of the predicate for specifiers.

$$\begin{aligned}
\text{bar}(x,y) &\iff \exists z[x \triangleleft z \wedge z \triangleleft y \wedge \bigvee_{l \text{ a lexical item}} l(y)] \\
\text{sibling}(x,y) &\iff \exists z[z \triangleleft x \wedge z \triangleleft y] \\
\text{spec}(x,y) &\iff \exists z[\text{sibling}(x,z) \wedge \text{bar}(z,y)]
\end{aligned}$$

Now we can finally define a predicate to identify moving phrases as specifiers of a head that takes no second argument.

$$\text{mover}(x) \iff \exists y[\text{spec}(x,y) \wedge \bigvee_{l \text{ takes at most one argument}} l(y)]$$

In the last step we require every trace to be c-commanded by a phrase.

$$\forall x[t(x) \rightarrow \exists y[\text{mover}(y) \wedge \text{c-com}(y,x)]]$$

Note that this is an actual formula, whereas all the previous formulas are just definitions that define predicates as a shorthand for specific subformulas that we use in definition of the ECP constraint.

Our implementation of the ECP leaves a lot to be desired, of course (a more adequate implementation is given in Rogers 1998, a logical formalization of GB that takes up over 150 pages). One glaring omission is that a mover cannot have left a specific trace behind if its category feature does not fit the position in which the trace occurs, for in that case the mover cannot have originated there. This can be added to the formula above by defining a predicate that checks what type of category is required for the position occupied by a trace and checks that the mover has this category — a rather simple exercise. A more interesting aspect is that movement is usually assumed to be locally bounded, which means that the licensor of a trace must occur within a specific domain, e.g. the smallest CP containing the trace. Such domain restrictions are very common across linguistic constraints, but fortunately they pose no challenge to first-order logic.

$$\begin{aligned}
\text{smallest}(x,y,ZP) &\iff ZP(y) \wedge y \triangleleft^+ x \wedge \neg\exists z[ZP(z) \wedge y \triangleleft^+ z \wedge z \triangleleft^+ x] \\
\text{local}(x,y,ZP) &\iff \exists z[\text{smallest}(x,z,ZP) \wedge \text{smallest}(y,z,ZP)]
\end{aligned}$$

$$\forall x[t(x) \rightarrow \exists y[\text{mover}(y) \wedge \text{c-com}(y,x) \wedge \text{local}(x,y,\text{CP})]]$$

While the example above implements a specific constraint, it is easy to see that the tricks and techniques work for virtually all linguistic constraints. Lexicalization and projection means that all local information can be inferred from the head of a phrase, and we can keep track of this information via newly defined predicates. More complex tree geometric notions like c-command or m-command are easily defined in terms of dominance, and the same holds for the notions of closeness and locality domain that are ubiquitous in linguistics. With a little bit of ingenuity, pretty much all syntactic constraints can be expressed in first-order logic.

First-order logic is a fragment of monadic second-order logic (MSO), which enriches first-order logic with the option to quantify over sets of nodes. It turns out that MSO is exactly as powerful as refined strictly 2-local tree grammars, so one can freely translate between grammars and MSO formulas (Büchi 1960). The construction is fairly involved (see Morawietz 2003 for details), and the computational complexity of the translation is unbounded — it doesn't get any less tractable than that. Nonetheless the conversion is reasonably fast in practice, and the fact that linguistic constraints can be stated without set quantification reduces the complexity a lot. Overall, then, it is a viable strategy to formalize linguistic theories in terms of first-order formulas, which are then automatically translated into refined tree grammars or possibly even CFGs.

Conjecture 16.1 (FO-Syntax). *All aspects of natural language syntax can be expressed in terms of first-order formulas over trees without interior node labels.*

2 Pushing the Limits

2.1 Polarity Items

There are a few phenomena that push the limits of what can be expressed in first-order logic. One of them is the distribution of so-called *polarity items*, or at least their predicted distribution according to standard generalizations. Polarity items can only occur in a context of a specific polarity. Negative polarity items (NPIs) are restricted to sentences where they appear in the scope of some negative element, though it is difficult to characterize what counts as such a negative element, as the examples below illustrate for the idiomatic NPI *lift a finger*:

(1) a. Nobody did so much as lift a finger.
b. * Everybody did so much as lift a finger.
c. You didn't so much as lift a finger.
d. * You did so much as lift a finger.
e. You never did so much as lift a finger.
f. You hardly did so much as lift a finger.
g. * You always did so much as lift a finger.

Negation and negative quantifiers license this NPI, but so do adverbials like *never* and *hardly*. The licensing element can also occur in a higher clause, in which case it may be a verb like *doubt*.

(2) a. Nobody believes you did so much as lift a finger.
b. * I believe you did so much as lift a finger.
c. I doubt you did so much as lift a finger.

The standard explanation is in term of semantic entailment patterns, with NPIs like *lift a finger* and *ever* restricted to downward entailing patterns, while positive polarity items (PPIs) like *somewhat* are restricted to contexts that are not downward entailing. The specifics of this proposal are not all that relevant to us except for one crucial property: negation switches downward entailing contexts into non-downward entailing ones, and *vice versa*. So if an NPI is licensed in configuration C, it should not be licensed in $\neg C$, but be licensed again in $\neg\neg C$. If this is indeed the correct analysis, then the distribution of NPIs and PPIs cannot be defined in first-order logic. That's because we are dealing with a *modulo* counting pattern, similar to what we saw for the string language $(aa)^+$, and these patterns require the full power of regular languages. Since first-order logic is a fragment of MSO, it does not have access to this additional power and thus cannot express the semantic generalization.

But whenever one is faced with generalizations that need more power than expected, the first step of action is to verify that I) the generalization is empirically correct, and II) there isn't a structurally simpler generalization that covers the same data. In the case of NPI and PPI licensing, the predicted pattern of alternating licensing contexts simply does not obtain.

(3) a. I doubt you did so much as lift a finger. (predicted grammatical)
b. ? I doubt you didn't do so much as lift a finger. (predicted ungrammatical)
c. ? Nobody doubts you did so much as lift a finger. (predicted ungrammatical)
d. ?? Nobody doubts you didn't do so much as lift a finger. (predicted grammatical)

In addition, NPIs are frequently accepted in contexts that look downward entailing but actually aren't.

(4) Every politican that no news paper reported on in great detail has ever made it past the first round.

These effects are usually attributed to processing errors, but in combination with the other facts they cast further doubt on the canonical analysis. Perhaps the distribution of NPIs and PPIs is subject to a first-order definable constraint after all, a constraint that looks quite a bit different from the semantic generalization simply due to the fact that first-order logic is incapable of expressing such a generalization.

2.2 Binding Theory

Binding theory regulates the distribution of elements whose reference depends on some other element in the sentence. These usually take the form of reflexives (*himself*, *herself*, *itself*), pronouns (*her*, *him*, *it*), and reciprocals (*each other*). The involvement of reference and meaning turns binding theory into a very wide and vaguely circumscribed phenomenon that spans syntax, semantics, and pragmatics. How exactly it can and should be factored across these components is difficult to decide, and as we will see next, the FO-syntax conjecture hinges on what kind of factorization one assumes.

The canonical view of binding theory is that DPs and referentially dependent elements are assigned referential indices as in the examples below.

(5) a. John$_i$ hit himself$_i$.
b. John$_i$ threatened Bill$_j$ to hit him$_j$.
c. * John$_i$ hit him$_i$.
d. * Mary$_i$ hit himself$_j$.

Sentences are then filtered out according to whether these indixations satisfy certain principles. The exact nature of said principles vary across languages, but we can identify to macro-parameters (cf. Kiparsky 2002, 2012).

± **s-bound** Does the referentially dependent element need a *syntactic* antecedent within a certain locality domain? (rather than just a previously established discourse referent)?

± **obviative** Must the index of the referentially dependent element be unique within a certain syntactic locality domain?

English *himself* for instance, must be s-bound within the smallest TP containing a subject, but it is not obviative. English *him*, on the other hand, is more ambiguous.

In general, *him* needs no syntactic antecedent, but it does presuppose a previously established discourse referent. This requirement is lifted when the pronoun is used *deictically*, e.g. by pointing at somebody in the room. Obviation is a little trickier to determine. The sentence *Gatsby*$_i$ *is much older than him*$_i$ seems perfectly fine when the speaker, mistaken about Gatsby's appearance, is unknowingly pointing at Gatsby. At the same, it is unclear that this sentence actually involves two identical indices, rather than two distinct indices (= distinct people in the speaker's mind) that happen to map to the same person in the real world. Excluding these special cases, though, *him* is obviative with the smallest TP containing a subject, which is witnessed by the fact that *John*$_i$ *hit him*$_i$ cannot mean the same thing as *John hit himself*.

See Heim (1998) for a detailed discussion of such sentences.

Other languages allow for different combinations. The Icelandic reflexive *sig*, for example, mirrors *himself* in that it is not obviative, but at the same time its locality domain for s-binding is much larger. Swedish *sig* and Marathi *aapan* have a similar s-binding requirement within a large locality domain but in addition they are obviative within the smallest TP containing a subject.

Given such a tremendous amount of cross-linguistic variation, one might be rightfully worried about the feasibility of a first-order formalization of binding theory. These worries are justified, but the culprit isn't variation; it is indexation. By assumption, languages are infinite and the length of sentences is unbounded. Consequently, there is no upper limit on how many referents may be mentioned in a sentence, which in turn implies that there is no finite bound on the number of indices we need — for every sentence with one more DP we may have to add one more index. This is obviously a problem given our assumption that alphabets must be finite, but there are ways to encode indices without putting them in the alphabet — for example, if all indices are natural numbers then *John*$_n$ could be represented as a node *John* whose sister is the root of a unary branching tree of depth n. But Rogers (1998) shows that no matter how indices are represented, the free-indexation system at the core of binding theory increases the power of the system so much that it becomes *undecidable*. That is to say, it is no longer possible to determine for arbitrary, freely indexed trees whether they are well-formed or not. So binding theory as commonly envisioned isn't just a problem for first-order logic, it is an insurmountable task for all computational machinery.

Of course this formal result seems at odds with our intuition that speakers do not have a hard time determining referents, so either all real-world problems simply happen to be the most trivial instances of an enormously complex system, or free indexation isn't the right way to think about binding theory — or possibly both. Index-free theories of binding theory have been developed in the meantime (Bonato 2005, Kobele 2006), defusing the problem to some extent. This is no guarantee, though, that binding theory is first-order definable. In order to show that, we have to go one step further.

As mentioned at the outset of this section, binding theory combines three different mechanisms:

- a syntactic mechanism that regulates whether specific lexical items may occur in a given context,
- a semantic mechanism that links the interpretation of referentially dependent elements to antecedents in the structure or the discourse,
- a discourse mechanism that keeps track of referents and uses pragmatic principles to help in discourse resolution.

The FO-conjecture is about the complexity of syntax, not semantics or pragmatics. So only the syntactic part of binding theory can furnish a licit counterexample. This syntactic part does not have to pay attention to specific meanings, since those are established by semantics. Instead, it only has to ensure that the distributional requirements are satisfied, which is the case as long as there is at least one possible reading. In other words, syntax only has to rule out sentences where the referentially dependent elements are distributed in such a fashion that it is impossible to assign a meaning according to the laws of binding theory.

With this shift in perspective, binding theory becomes a lot simpler at the syntactic level. Determining that s-binding is satisfied isn't all that different from our constraint earlier on that every trace needs to be c-commanded by a mover. The definition of locality differs quite a bit, and what counts as a valid antecedent is more restricted (*Mary* cannot be an antecedent of *himself* due to gender mismatch), still it all fits comfortably within the bounds of first-order logic. Obviation is a lot harder. Due to the unbounded nature of various constructions, it is theoretically possible to have an unbounded number of obviative elements within the same obviation domain. Now it necessarily holds that if there are n obviative elements within the same obviation domain, then there must be at least n possible antecedents outside the obviation domain. You might remember that this kind of unbounded counting cannot be achieved with finite-state methods, which are equivalent to MSO, which subsumes first-order logic. So first-order logic cannot even handle syntactic binding theory unless the following additional restriction holds in every natural language:

This is a necessary condition, not a sufficient one. We ignore the extra conditions here for the sake of exposition. See Graf & Abner (2012) for a full discussion.

Limited Obviation There is an upper bound k such that if an obviation domain contains $n > k$ obviative elements, k antecedents suffice to furnish a grammatical reading.

Limited Obviation thus posits that some obviative elements within the same obviation domain can have the same referent, and that only a fixed number of antecedents are ever needed thanks to this potential overlap.

Limited Obviation is not the kind of condition linguists would come up with, it follows strictly from our search for a restriction that renders binding theory first-order definable. This makes it even more surprising that Limited Obviation seems to be an actual language universal, as is argued by Graf & Abner (2012) based on a careful examination of a variety of unbounded constructions. So even though binding theory is a tight fit, it does stay within the bounds of first-order logic.

2.3 Copy Movement

3 Towards a Context-Free Solution

Relevant literature for Unit 16

add more info

Backofen, Rolf, James Rogers & K. Vijay-Shanker. 1995. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language and Information* 4. 5–39. https://doi.org/10.1007/BF01048403.

Blackburn, Patrick, Claire Gardent & Wilfried Meyer-Viol. 1993. Talking about trees. In *Proceedings of the sixth conference of the European chapter of the Association for Computational Linguistics*, 30–36. https://doi.org/10.3115/976744.976748.

Bonato, Roberto. 2005. Towards a computational treatment of binding theory. In *Logical aspects of computational linguistics, 5th international conference, LACL 2005, bordeaux, france, april 28-30, 2005*.

Büchi, J. Richard. 1960. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 6. 66–92. https://doi.org/10.1002/malq.19600060105.

Graf, Thomas. 2013. *Local and transderivational constraints in syntax and semantics*. UCLA dissertation. http://thomasgraf.net/doc/papers/Graf13Thesis.pdf.

Graf, Thomas & Natasha Abner. 2012. Is syntactic binding rational? In *Proceedings of the 11th international workshop on Tree Adjoining Grammars and related formalisms (TAG+11)*, 189–197. http://thomasgraf.net/doc/papers/GrafAbner12TAG.pdf.

Heim, Irene. 1998. Anaphora and semantic interpretation: A reinterpretation of Reinhart's approach. In Uli Sauerland & O. Percus (eds.), *The interpretive tract*, vol. 25 (MIT Working Papers in Linguistics), 205–246. Cambridge, MA: MIT Press.

Kiparsky, Paul. 2002. Disjoint reference and the typology of pronouns. In Ingrid Kaufmann & Barbara Stiebels (eds.), *More than words*, vol. 53 (Studia Grammatica), 179–226. Berlin: Akademie Verlag.

Kiparsky, Paul. 2012. Greek anaphora in cross-linguistic perspective. *Journal of Greek Linguistics* 12. 84–117.

Kobele, Gregory M. 2006. *Generating copies: An investigation into structural identity in language and grammar*. UCLA dissertation. http://home.uchicago.edu/~gkobele/files/Kobele06GeneratingCopies.pdf.

Kracht, Marcus. 1997. Inessential features. In Alain Lecomte, F. Lamarche & G. Perrier (eds.), *Logical aspects of computational linguistics*. Berlin: Springer.

Morawietz, Frank. 2003. *Two-step approaches to natural language formalisms*. Berlin: Walter de Gruyter. https://doi.org/10.1515/9783110197259.

Potts, Christopher & Geoffrey K. Pullum. 2002. Model theory and the content of OT constraints. *Phonology* 19(4). 361–393.

Pullum, Geoffrey K. 2007. The evolution of model-theoretic frameworks in linguistics. In James Rogers & Stephan Kepser (eds.), *Model-theoretic syntax @ 10*, 1–10.

Rogers, James. 1998. *A descriptive approach to language-theoretic complexity*. Stanford: CSLI.

Unit 17

Derivations as the Primary Data Structure

Our formal understanding of syntax has matured quite a bit over the last few weeks, but it seems that we have reached an impasse. On the one hand, the hypothesis that syntax is definable in first-order logic seems to be on the right track when it comes to expressing syntax dependencies: virtually all constraints in the syntactic literature fit into this class, and where constraints in their canonical formulation would require more powerful computational machinery — as is the case with NPI-licensing and binding theory — a closer look at the data actually supports a more restricted formulation that does not exceed the limits of first-order logic. When formal predictions line up with empirical data so nicely, the odds are good that one has identified a relevant property.

On the other hand, there are weak-generative capacity arguments that first-order logic is not enough. Since the first-order definable tree languages are just another dot in the hierarchy of tree languages between CFGs an refined strictly local, a first-order definable set of phrase structure trees can only generate context-free string languages. But constructions like reduplication in Yoruba (Kobele 2006) produce string languages of the form a^{2^n}, which is not context-free. As you might recall, weak generative capacity arguments are the strongest argument one can make for establishing a lower bound; at the very least, a grammar formalism must be able to generate the correct string language. So unless we want to doubt the Yoruba data (and several other constructions in completely unrelated languages), it seems that we have unassailable proof that syntax is not first-order definable.

Remember that the fact that a language contains a proper subset of complexity c does not imply that the whole language has at least complexity c. After all, the regular string language $(aa)^+$ is a subset of the strictly 0-local Σ^*. But Kobele (2006) uses the right proof technique to show that Yoruba as a whole is not context-free.

Today we will see that this conclusion is not quite right, and that a more refined view can accommodate both the relative weakness of syntactic constraints over trees and the much greater expressivity over strings. This approach will also solve a variety of other problems, such as how to accommodate copy movement as well as string patterns that do not involve copying yet are nonetheless not context-free.

1 The Role of Syntax

1.1 Structural Descriptions

In our discussion so far, we have treated syntax as a mechanism that generates tree languages. Our original motivation for this step was that tree languages make it very easy to generate non-regular string languages. We could have used a different strategy.

For instance, a finite-state automaton can be turned into a *push-down automaton* (PDA) by adding a stack that can memorize an unbounded number of symbols but may only read or manipulate the top-most symbol of the stack (see e.g. Sipser 2005). The transition rules of a PDA are expanded FSA transitions that also take the top symbol in the stack into account. One can show that the stack symbols correlate in an abstract sense with the non-terminal nodes of a tree, but that does not mean that a PDA generates tree languages of any kind. Its output is string languages, and the class of string languages recognized by a PDA is exactly the class of context-free languages. So trees were not an inevitable choice for our treatment of syntax, instead of enriching our data structures from strings to trees we could have adopted a more sophisticated memory architecture.

But we opted for trees because they line up with how linguists think of trees. There is plenty of evidence that sentences have an internal tree-like structure. Some arguments pertain to how children generalize from input when acquiring their language, some hinge on prosody, others on semantic scope. Syntactic constraints also seem to pay attention to structural factors rather than just string properties. And formally, of course, the usage of trees revealed a nice parallel between phonology and syntax such that the technical machinery we employed for the former could readily be applied to the latter. So a tree-based view of syntax is ultimately a lot more appealing.

Syntax, then, is a mechanism for generating tree languages such that each tree is a structural description of some sentence. Every linguist's natural inclination at this point would be to equate structural description with phrase structure trees. But nothing what we have said so far supports this conclusion. One can imagine many different kinds of tree-like structural descriptions. A dependency tree, for instance, encodes the head-argument relation rather than phrase structure, which must be deduced from the structure (whereas the head-argument relation must be deduced in a phrase structure tree with movement). An example of the two tree structures is given in Fig. 17.1. The phrase structure tree dwarfs the dependency tree in size, yet one can freely convert between the two formats given a fixed theory of what phrase structure trees look like.

The phrase structure tree ties the grammatical relations like subject, direct object and indirect object to specific positions in the tree and then displaces various subtree in order to obtain the desired word order. The labels and projections only matter to the extent that they identify the relevant positions and regulate what may be displaced. The dependency tree explicitly represents the grammatical relations and completely ignores word order, which must be computed through a separate mechanism, e.g. a mapping from dependency trees to derivation trees. Given this kind of malleability, it is a lot less clear what kind of structural description syntax operates over.

1.2 Interfaces

Another argument for phrase structure trees is that they (supposedly) serve as the input to what's called the *interfaces*, PF and LF. LF stands for *logical form* and is the interface to semantics, i.e. where the meaning of sentences is computed. PF stands for *phonetic/phonological/physical form* and covers all kinds of aspects related to uttering the tree, e.g. computing its string yield. The interplay between syntax, PF and LF is usually depicted via the (inverted) T-model.

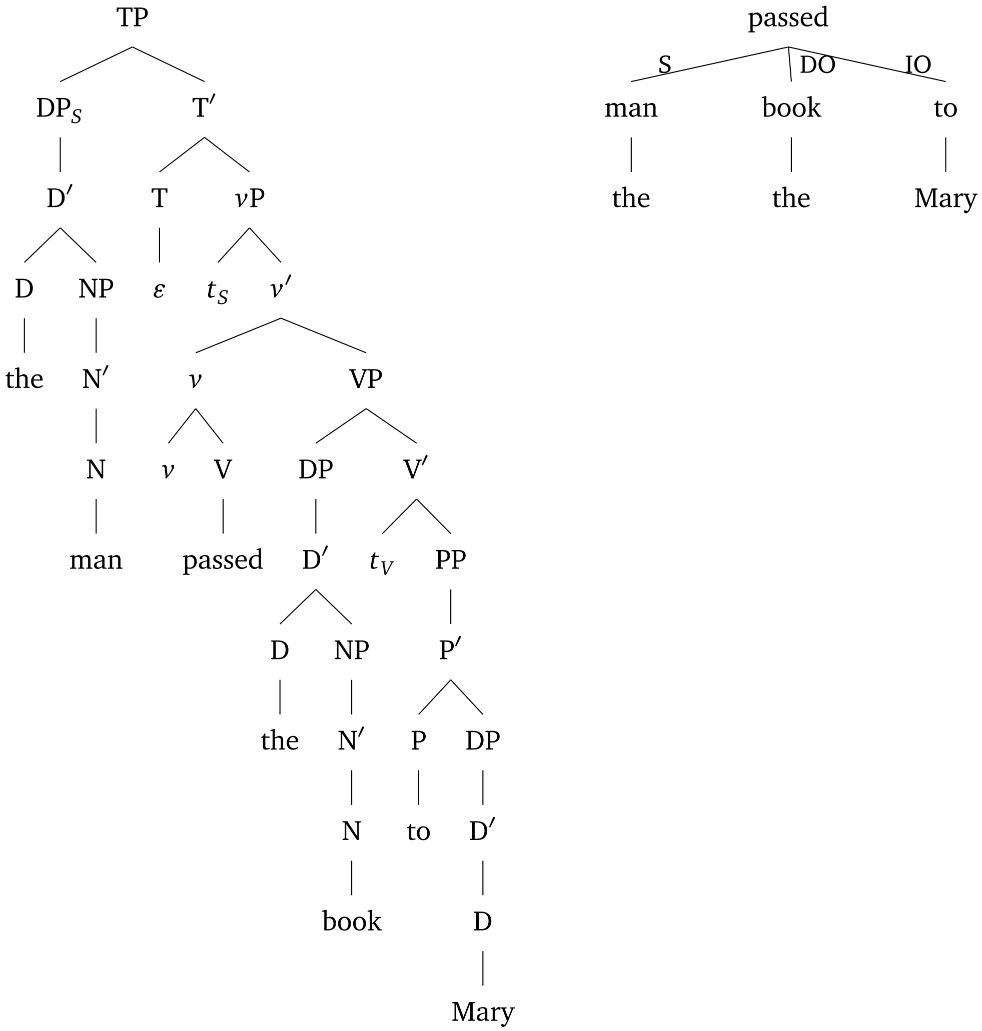


Figure 17.1: Phrase structure tree and dependency tree for *The man passed the book to Mary*.

$$\begin{tikzcd}
& \text{Lexicon} \arrow[d, "\text{Syntax}"] & \\
& \text{Spell-Out} \arrow[dl, "\text{PF}"'] \arrow[dr, "\text{LF}"] & \\
\pi & & \lambda
\end{tikzcd}$$

The idea is that syntax combines a collection of lexical items into a phrase structure tree. Once this assembly process is finished, the operation Spell-Out passes on the tree to PF and LF, which apply further modifications to yield an output string π and some kind of logical formula λ.

Before we determine whether the T-model favors a particular kind of tree structure, we should establish first that this is a reasonable model of language. After all, the T-model has been extensively criticized by various linguists on the grounds that it, supposedly, puts too much emphasis on syntax and rules out any kind of PF and LF interaction. A common complain is also how syntax can pick the right lexical items and build the desired tree for a given meaning. This criticism is confused and stems from a common mistake among linguists: criticizing a formalism for the intuition that is used to make it more approachable. The T-model has nothing to say about the actual building process, it is not a model of generation. What is describes is a collection of languages and the functional relations between them. The lexicon is a finite collection of lexical items, and the set T of syntactically well-formed phrase structure trees is the image of the lexicon under a function we call syntax. Similarly, the sets of strings and meanings of the language, respectively are π, the image of T under the PF mapping, and λ, the image of T under the LF mapping. All the T-model establishes is sets and mappings between these sets.

This also means that arguments about derivational versus representational syntax, and one-level versus multiple levels are all rather misguided. They might aid the intuitions of researchers and lead them to posit interesting conjectures or seek out new data, but on a mathematical level there is no sense in which mappings are clearly derivational or representational.

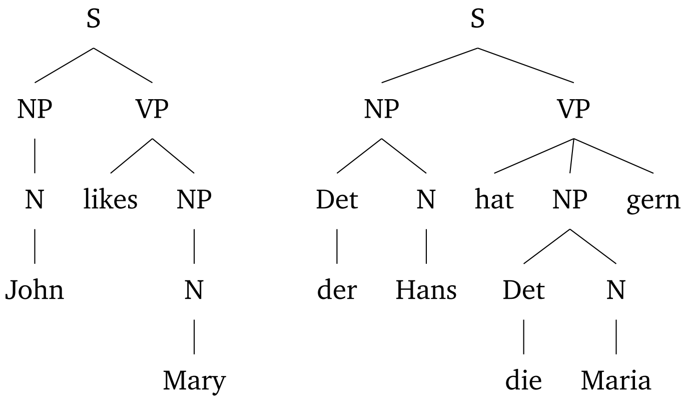
Mappings have no inherent directionality, for given a relation R one can always take its inverse R^{-1}. So producing a string to express meaning ϕ is tantamount to computing $\mathrm{PF}(\mathrm{LF}^{-1}(\phi))$. Figuring out the meaning of string s, on the other hand, means computing $\mathrm{LF}(\mathrm{PF}^{-1}(s))$. All the T-model does, then, is to establish syntax as a mediator between these two kinds of objects, physical utterances and logical formulas: the process of mapping one to the other is analyzed as first constructing a fitting tree structure which is then fed into the correct mapping. Yet at no point does syntax "come first", and the arrows in the T-model only indicate the direction in which the individual mappings are defined, not the direction in which the mapping is run.

The T-model does not even make a bold ontological claim that one could take issue with. Whenever one has a relation R between two sets S and T, this relation can be decomposed into a *bimorphism*, which consists of two mappings R_1 and R_2 and an *interpolant* L such that S is the image of L under R_1, T is the image of L under R_2, and $R = R_1^{-1} \circ R_2$ (Arnold & Dauchet 1982). That is exactly the scenario described by the T-model: S is the set of output strings, T the set of logical formulas, L is the set of structural descriptions, R_1 is PF, and R_2 is LF. If one takes syntax to be a finite description of the interpolant L based on a language's lexicon, the place of syntax in the T-model becomes an unavoidable mathematical truth (and any interaction between PF and LF happens in syntax by definition).

The flip side is that syntax may be just a mathematical truth, with only π and λ enjoying any degree of cognitive reality. That would not be particularly shocking, though, since one of the main advantages of abstraction is that objects can be posited irrespective of whether they are real — monoids do not exist in any verifiable sense, but they are a useful abstraction and unification of a variety of distinct objects. Even if syntax isn't real, it plays an important part in modeling the behavior of a real cognitive ability.

Another point supporting the generality of the T-model is that the very same perspective is becoming increasingly more important in machine translation. In machine translation, one has to translate freely between two languages, e.g. English and Chinese. The longest time, tree-based approaches took the form of *synchronous grammars*, which generate two trees in parallel. For example, the following synchronous CFG translates *John likes Mary* into Bavarian German *Der Hans hat die Maria gern*, and the other way round.

$$
\begin{array}{rcl}
\text{S} & \rightarrow & \langle \text{NP VP, NP VP} \rangle \\
\text{NP} & \rightarrow & \langle \text{N, Det N} \rangle \\
\text{Det} & \rightarrow & \langle \text{, der} \rangle \mid \langle \text{, die} \rangle \\
\text{N} & \rightarrow & \langle \text{John, Hans} \rangle \mid \langle \text{Mary, Maria} \rangle \\
\text{VP} & \rightarrow & \langle \text{likes NP, hat NP gern} \rangle
\end{array}
$$



Somewhat surprisingly, synchronous grammars are not particularly well-behaved, with minor variants yielding vastly different formal properties. But this variability can be studied more insightfully from the bimorphism perspective. In our example, one would posit an interpolant that represents the structural similarities between the two trees, and two mappings that turn the interpolant into the English and German tree, respectively. The formal differences between variants of synchronous CFGs are then revealed to give rise to mappings of greatly differing complexity. These differences are very hard to state as properties of the direct mapping between languages, but are highly transparent from the bimorphism perspective (see e.g. Shieber 2004).

Overall, the T-model represents a very general idea of decomposing mappings. In fact, since the basic task of any native speaker is to translate sentences into thoughts and the other way round, it isn't too surprising that the T-model looks exactly like a modern machine translation template. The big question at this point is whether this tells us anything about what the interpolant should look like. In other words, what sort of information needs to be present in the structural descriptions furnished by syntax?

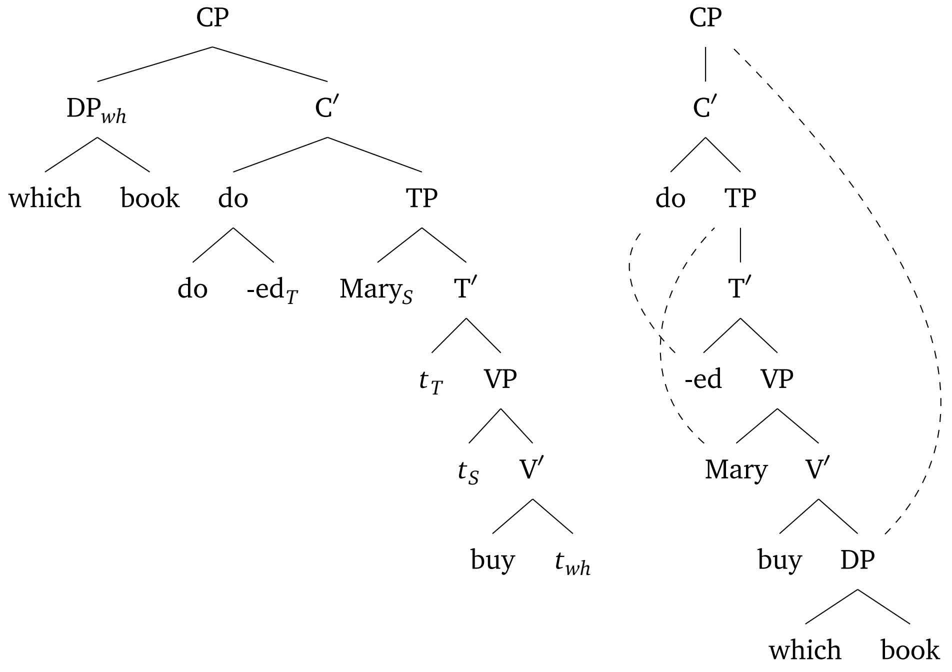
2 Interpolation via Derivation Trees

2.1 Movement without Displacement

We are still committed to the idea that syntax is first-order definable, but we have now refined our conjecture so that it applies to the structural descriptions that are regulated by syntax. These are not necessarily phrase structure tree, but some interpolant that provides all the necessary information so that PF and LF can produce the desired outputs π and λ. Seeing how phrase structure trees are still used in semantics and research on prosody, we do not want to stray too far to them. If a minor variant of phrase structure trees is enough to give us a first-order definable interpolant, this is sufficient support for our conjecture.

What exactly is it, then, that makes phrase structure trees the most commonly used data structure in linguistics? One advantage is that they still provide an implicit encoding of argument structure, even if it is partially obscured by displacement. Argument structure is obviously essential for determining basic thematic relations in a sentence, e.g. who is punching whom. Phrase structure trees also encode semantic scope, e.g. that the universal has scope over the existential in *Every student that took some history course wound up bored*. But the LF mapping is not entirely straight-forward since semanticists often undo displacements or add new instances thereof in order to get the correct scope. This contrasts with the PF mapping, which only has to read the leafs from left to right in order to get the output sentence (we ignore prosody and other phonological factors here for the sake of exposition). Right now, then, the phrase structure trees fully encode movement to keep PF simple, even though it leads to complications on the LF side. There is no *a priori* reason why the workload should be distributed this way. We could just as well leave displacement to the PF mapping so that LF no longer has to undo movement steps — and as we will see next, that is exactly the answer to our problem.

Suppose that instead of moving phrases in our syntactic trees, we simply indicate what is supposed to move where, e.g. via a dashed branch. That way, LF still gets all the information it needs to compute relative scope, while PF can carry out the actual displacement. Compare the two trees for *which book did Mary buy* below (we do not use an X′ template here in order to keep the trees as simple as possible).



Obviously PF can convert the tree to right into the one to the left by making each mover the daughter of the highest node it is related to via a dashed branch. In fact, we can even define this mapping in first-order logic.

2.2 The PF Mapping

So far we have only used logic to define constraints, but it is actually fairly easy to write logical formulas that define mappings between trees. The idea is that we can write formulas that express instructions of the form "if this relation holds between nodes x and y in the input tree, then the following relation must hold between them in the output tree".

Example 17.1 A First-Order Definable PF Mapping

Suppose that we already have a predicate $\mathrm{occ}(x, y)$ that holds between x and y iff x is a landing site (= occurrence) for y, i.e. x immediately dominates y via a dashed branch. Then we first define a predicate to pick out the final occurrence of y, which is where it will be pronounced in the string. Due to how movement works, the final occurrence of y will always be the highest node occurrence of y.

$$\text{f-occ}(x, y) \iff \mathrm{occ}(x, y) \wedge \neg\exists z[z \triangleleft^+ x \wedge \mathrm{occ}(z, y)]$$

Now we define a predicate $\blacktriangleleft$ that represents immediate dominance in the output tree. We specify that $x \blacktriangleleft y$ iff either x is the mother of y and y does not move, or x is the final landing site of y.

$$x \blacktriangleleft y \iff (x \triangleleft y \wedge \neg\exists z[\mathrm{occ}(z, y)]) \vee \text{f-occ}(x, y)$$

A similar strategy can be used to define the left-sibling predicate such that movers are the left sibling of the daughter of their final occurrence and all other nodes inherit the sibling specification from the input tree (if we assume that syntactic trees are unordered, this information must be inferred from some other relation like c-command). With the left-sibling relation, it is very easy to compute the string yield of the output tree, which is exactly the utterance represented by the input tree.

The current PF mapping gives us non-copying movement, but it is very simple to add in copy movement. In fact, copy movement is much easier to handle because it corresponds to a graph-theoretic operation known as *unfolding*, which turns a graph into a tree: if a node n has i mothers $m_1, \ldots, m_i$, make i copies of the subtree rooted in n and make each copy a daughter of exactly one of n's mothers. Unfoldings are not definable in first-order logic, but we can combine them with the previous transduction. First, we assume that each mover has a feature that tells us whether is undergoing normal movement or copy movement. Then we refine $\blacktriangleleft$ so that normal movers are still attached to only their final occurrence, whereas copy movers are attached to all their occurrences. We then compose this mapping with an unfolding to get a PF mapping that can handle both copying and non-copying movement.

Notice that the existence of copying movement is no longer an obvious threat to our conjecture that syntax is first-order definable. While the PF output structures may not fit into this class, the structural descriptions of syntax might since they do not contain multiple copies of a tree. But it might still be the case that these kind of augmented trees with dashed branches are not first-order definable for some other reason. As we will see next, that is not the case either given an empirically supported constraint on displacement.

2.3 Doing Away With Movement Branches

The question whether augmented trees with dashed movement branches are first-order definable intersects with another issue, which is whether the branches are needed at all. Remember that we want syntax to still use tree structures since those are well-understood objects and can automatically be converted into CFGs, which makes them amenable to a variety of standard techniques (e.g. from the parsing literature). So we want to get rid of the branches without losing track of what is moving where.

Let us assume for the sake of exposition that every phrase moves at most once and that every phrase is a target for at most one mover. So TP, for instance, can only be targeted by the subject, and the subject moves directly to TP without going any farther or stopping somewhere else on the way. We can represent this fact by adding a feature TARGET with value *nom* to the AVM of TP's head, and the corresponding DISPLACE with value *nom* to the head of the subject. As long as there is no other head with a *nom*-valued feature, the features make it clear that the subject moves to SpecTP, so we do not need a dashed branch to indicate this information — it can be inferred from the feature calculus.

Example 17.2 A Feature-Based Definition of Occurrences

The feature-based specification of movement allows us to define the predicate occ(x,y) from the previous example without making reference to dashed branches. Instead, the occurrence of a phrase (remember that we assume every phrase moves at most once, so it cannot have more than one occurrence) is the closest dominating node whose head has a target feature matching the displace feature of the head of moving phrase. To make the formula we readable, it is preferable to first define the notion of a potential occurrence via feature matching and then define occurrence as the closest potential occurrence.

$$\text{p-occ}(x,y) \iff \exists t \exists d[\text{head}(t,x) \wedge \text{head}(d,y) \wedge \bigvee_{l \text{ has target feature } f} l(t) \wedge \bigvee_{l \text{ has displace feature } f} l(d)]$$

$$\text{occ}(x,y) \iff \text{p-occ}(x,y) \wedge \neg \exists z[\text{p-occ}(z,y) \wedge x \triangleleft^+ z]$$

The assumption that every feature occurs on at most one head in a tree is highly unrealistic. For example, a sentence with multiple embeddings, e.g. *John said that Mary believes that Peter likes her*, involves multiple subjects that all move to their respective SpecTP, and we do not want to posit different features for these different instances of subject movement. Fortunately this isn't necessary. In the example above, we defined *occurrence* — our logical analogue to dashed branches — as the closest node with matching target feature. In a sentence with multiple embeddings, the closest potential occurrence for each subject is the TP in the same clause. So we can allow multiple heads to have the same displace feature as long as they each have distinct occurrences. We stipulate that all trees in which this is not the case are ungrammatical, which can even be enforced via a first-order constraint:

Determinism $\forall x \exists t \Big[\big(\text{head}(x,t) \wedge \bigvee_{l \text{ has target feature } f} l(t) \big) \rightarrow \exists ! y [\text{occ}(x,y)] \Big]$

At first sight determinism does not look empirically tenable since there are plenty of cases where two phrases move to the same target due to the same feature, e.g.

in multiple wh-movement. But I) these cases have competing analysis where the wh-phrases form a cluster that moves once, rather than all phrases moving at the same time, and II) it is unclear whether multiple wh-movement is unbounded. As long as there is a finite bound on the number of parallel movers, the data can be brought in line with Determinism. The crude but simple option is to distinguish multiple features of the same type, e.g. wh_1, wh_2, …instead of just wh. A more elegant solution is to provide a first-order definable locality mechanism that decides the relative movement order between phrases, thus preserving the intuition of Determinism that we can correctly infer dashed branches just from feature specifications.

2.4 Derivation Trees and Minimalist Grammars

We now have an incredibly flexible and powerful yet first-order definable theory of syntax.

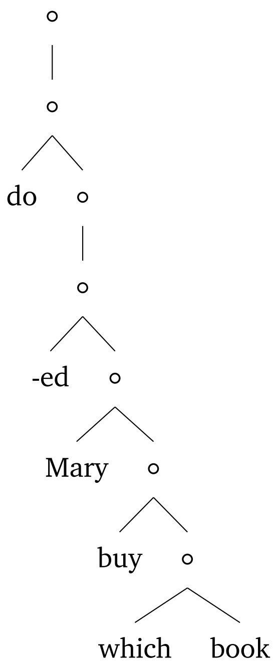
1. Lexical items are richly annotated AVMs whose features determine subcategorization and movement dependencies (the latter via TARGET and DISPLACE).

2. Syntactic trees are label-free and represent movement only indirectly through the features.

3. Movement is regulated by the first-order definable occurrence predicate and the first-order constraint Determinism.

4. The PF mapping turns syntactic trees into phrase structure trees. The non-copying part of the mapping is first-order definable.

Notice that since trees are label-free, our tree for *which book did Mary buy* should actually look like the one below (with AVMs omitted).



Unary branching nodes indicate where movement takes place. In the jargon of the dominant theory of syntax, *Minimalism*, we can identify them with instances of the

Move operation. The binary branching nodes on the other hand correspond to items being combined via subcategorization, which is handled by Merge. So the structural descriptions that we have converged on due to our desire to maintain first-order definability are the derivation trees of the leading syntactic framework. In a certain sense we have come full circle, seeing how originally we viewed phrase structure trees as the derivation trees of CFGs.

The moral of the story is that even though CFGs are not powerful enough to generate the right string languages for syntax, that shortcoming is due to their simple PF mapping: the string yield is obtained by simply reading the leaves from left to right. In our new system, we still have very simple tree structures, but the PF mapping is more complex since it now also has to take movement into account. But once the complexity of computing the string yield is factored out, the remainder is a first-order definable tree language that can accommodate all syntactic dependencies.

This is the core insight of recent work on *Minimalist grammars* (MGs), a formalization of Minimalism by Stabler (1997). In the last 10 years, MGs have generalized the ideas above to allow for new movement types, provide a corresponding LF mapping, and much more. For an extensive overview, see Graf (2013). At this point, we only note that the first few years of MG research were rather messy and it was difficult to estimate how the formalism might behave when new operations or constraints are introduced. The factorization of MGs into a first-order definable derivation tree language and a mapping from derivations to output trees (or simply strings) has made many of these questions downright trivial. This shows that the full scope of a system often proves difficult to grasp and progress is best achieved by isolating its components and how they interact.

This insight is not restricted to MGs or linguistics, of course. All science proceeds by breaking down the complex into simpler subparts, and we see the same in mathematics and computer science. For example, tree transducers (the tree analog of FSTs) come in dozens of varieties, and the common strategy for bringing out their commonalities is to decompose them into cascades of simpler transducers. That way, a top-down tree transducer with regular look ahead turns out to be the composition of a cylindrification and a top-down tree transducer. See Comon et al. (2008) and Gécseg & Steinby (1984, 1997) for more examples.

Some important linguistic insights have come out of MG research, including:

1. The ECP, which requires that a mover c-commands its trace, is too strict as MGs with the ECP generate only context-free languages. We already saw that ECP-obeying movement can be encoded via slash-feature percolation over phrase structure trees and thus can be handled by CFGs, so it is not surprising that ECP movement gives only context-free languages. But the result comes with a complementary finding, which is discussed next.

2. MGs with non-ECP obeying movement can generate languages that aren't context-free even if the movement does not involve copying. That's because such instances of movement are very complex in the phrase structure tree — the trace of a mover can now occur at various positions that the mover does not c-command. This relaxation makes the process of matching movers and their traces so difficult that it can no longer be accomplished with first-order logic (nor monadic second-order logic) over phrase structure trees. In the derived tree, though, ECP-obeying movement and non-ECP-obeying movement have exactly the same complexity since phrases remain in their base position in the derivation tree anyways.

3. As a nice corollary, it follows that blocking non-ECP-obeying movement requires a rather unnatural constraint over derivation trees. So the derivational perspective suggests that a formalism with movement should have both ECP-obeying and non-ECP-obeying movement, which is indeed the received opinion in Minimalist syntax nowadays.

4. MGs with non-ECP-obeying movement weakly generate exactly the class of *multiple context-free languages* (MCFL). This class does not contain languages like a^{2^n}, but it does include languages like $a^n b^n c^n$, which are the formal counterpart of cross-serial dependencies.

5. The copy language $\{ww \mid w \in \Sigma^*\}$ is also an MCFL. In fact, $\left\{w^i \mid w \in \Sigma^*\right\}$ is an MCFL for every $i \geq 0$. So one does not need copy movement to create multiple copies. Copy movement is only required when the output of copying is part of another copying operation.

6. That said, creating a copy language without copy movement requires highly convoluted and unintuitive grammars. This is a case where one might want to move to a more powerful operation not because it is mandated by weak generative capacity, but because the weaker variant cannot express important generalizations as elegantly.

7. Adding other non-copying movement types like sidewards movement or lowering to the formalism does not increase its weak generative capacity beyond MCFLs, but strong generative capacity does increase.

8. Every movement type can be decomposed into an instance of standard upward movement that is possibly followed by an instance of downward movement.

9. All common parsing strategies for CFGs can be extended to MGs because the derivation trees are context-free and contain all necessary information. Adapting a parser from CFGs to MGs boils down to switching out the simple yield-mapping used with CFGs for a more complex one that recognizes the fact that linear order in MG derivations does not reflect linear order in the string.

10. The parsing facts immediately imply that MGs can be parsed top-down (see Stabler 2013), so despite claims to the contrary in the psycholinguistic literature it is not the case that Minimalist syntax only allows for (empirically inadequate) bottom-up processing.

11. Various debates about the shape of phrase structure trees, such as whether they are trees or multidominance trees or what kind of label is projected become immaterial because the central data structure is derivation trees, which can be label-free trees and still be first-order definable.

12. Similarly, the ongoing debate between derivational (≈ process-based) and representational (≈ constraint-based) frameworks makes little sense from this perspective. While we interpret the underlying trees as derivation trees, that is just one of many possible interpretations. The only thing that is for sure is that they are trees that produce the intended outputs under specific mappings — nothing about them is inherently derivational. And although the tree language and the mappings are defined declaratively via first-order logic, which is close to the spirit of representational approaches, we could just as well have used a tree automaton and a tree transducer instead, which are operational and thus derivational. In other words, the MG perspective is neither inherently derivational nor inherently representational, and even the machinery used in the specification can be freely picked from either camp.

This list is not exhaustive, of course, but it shows how a computationally informed perspective can complement and enhance standard linguistic research. And this is ultimately what this journey of ours has been all about: to look at language from a computational lens, to give formal characterizations and identify classes that capture specific properties, to make out important similarities, distinctions and limitations of linguistic submodules, and to use this knowledge in an effort to study and explain specific processes, phenomena and universals.

Relevant literature for Unit 17

add more info

Arnold, A. & M. Dauchet. 1982. Morphismes and bimorphismes d'arbres. *Theoretical Computer Science* 20. 33–93.

Comon, H. et al. 2008. *Tree Automata: Techniques and Applications*. Available online: http://www.grappa.univ-lille3.fr/tata. release from November 18, 2008.

Gécseg, Ferenc & Magnus Steinby. 1984. *Tree automata*. Budapest: Academei Kaido. http://arxiv.org/abs/1509.06233.

Gécseg, Ferenc & Magnus Steinby. 1997. Tree languages. In Gregorz Rozenberg & Arto Salomaa (eds.), *Handbook of formal languages*, vol. 3, 1–68. New York: Springer.

Graf, Thomas. 2013. *Local and transderivational constraints in syntax and semantics*. UCLA dissertation. http://thomasgraf.net/doc/papers/Graf13Thesis.pdf.

Kobele, Gregory M. 2006. *Generating copies: An investigation into structural identity in language and grammar*. UCLA dissertation. http://home.uchicago.edu/~gkobele/files/Kobele06GeneratingCopies.pdf.

Shieber, Stuart M. 2004. Synchronous grammars as tree transducers. In *TAG+7: seventh international workshop on Tree Adjoining Grammar and related formalisms*, 88–95.

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

Stabler, Edward P. 1997. Derivational Minimalism. In Christian Retoré (ed.), *Logical aspects of computational linguistics*, vol. 1328 (Lecture Notes in Computer Science), 68–95. Berlin: Springer. https://doi.org/10.1007/BFb0052152. https://doi.org/10.1007/BFb0052152.

Stabler, Edward P. 2013. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5. 611–633. https://doi.org/10.1111/tops.12031. https://dx.doi.org/10.1111/tops.12031.

Bibliography

Abels, Klaus. 2003. *Successive cyclicity, anti-locality, and adposition stranding*. University of Connecticut dissertation.

Arnold, A. & M. Dauchet. 1982. Morphismes and bimorphismes d'arbres. *Theoretical Computer Science* 20. 33–93.

Ashok, Vikas Ganjigunte, Song Feng & Yejin Choi. 2013. Success with style: using writing style to predict the success of novels. In *Emnlp 2013*, 1753–1764.

Backofen, Rolf, James Rogers & K. Vijay-Shanker. 1995. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language and Information* 4. 5–39. https://doi.org/10.1007/BF01048403.

Blackburn, Patrick, Claire Gardent & Wilfried Meyer-Viol. 1993. Talking about trees. In *Proceedings of the sixth conference of the European chapter of the Association for Computational Linguistics*, 30–36. https://doi.org/10.3115/976744.976748.

Bonato, Roberto. 2005. Towards a computational treatment of binding theory. In *Logical aspects of computational linguistics, 5th international conference, LACL 2005, bordeaux, france, april 28-30, 2005*.

Büchi, J. Richard. 1960. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 6. 66–92. https://doi.org/10.1002/malq.19600060105.

Chandlee, Jane. 2014. *Strictly local phonological processes*. University of Delaware dissertation. http://udspace.udel.edu/handle/19716/13374.

Chomsky, Noam. 1957. *Syntactic structures*. The Hague: Mouton.

Chomsky, Noam. 1959. On certain formal properties of grammars. *Information and Control* 2. 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6. https://doi.org/10.1016/S0019-9958(59)90362-6.

Chomsky, Noam. 1965. *Aspects of the theory of syntax*. Cambridge, MA: MIT Press.

Chomsky, Noam. 1990. On formalization and formal linguistics. *Natural Language and Linguistic Theory* 8. 143–147.

Chomsky, Noam. 1993. A Minimalist program for linguistic theory. In Kenneth Hale & Samuel Jay Keyser (eds.), *The view from building 20*, 1–52. Cambridge, MA: MIT Press.

Chomsky, Noam & Morris Halle. 1968. *The sound pattern of English*. New York: Evanston.

Chomsky, Noam & M. P. Schützenberger. 1963. The algebraic theory of context-free languages. In P. Braffort & D. Hirschberg (eds.), *Computer programming and formal systems* (Studies in Logic and the Foundations of Mathematics), 118–161. Amsterdam: North-Holland.

Comon, H. et al. 2008. *Tree Automata: Techniques and Applications*. Available online: http://www.grappa.univ-lille3.fr/tata. release from November 18, 2008.

Crocker, Matthew. 2010. Computational psycholinguistics. In Alex Clark, Chris Fox & Shalom Lappin (eds.), *Handbook of computational linguistics and natural language processing*, 482–514. London: Blackwell.

Dasgupta, Sajoy, Christos Papadimitriou & Umesh Vazirani. 2006. *Algorithms*. New York, NY: McGraw Hill.

Frank, Robert & Giorgio Satta. 1998. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics* 24. 307–315. http://www.aclweb.org/anthology/J98-2006.

Frazier, Lyn. 1979. *On comprehending sentences: syntactic parsing strategies*. University of Connecticut dissertation.

Frazier, Lyn & Keith Rayner. 1982. Making and correcting errors during sentence comprehension: eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology* 14. 178–210.

Gazdar, Gerald et al. 1985. *Generalized phrase structure grammar*. Oxford: Blackwell.

Gécseg, Ferenc & Magnus Steinby. 1984. *Tree automata*. Budapest: Academei Kaido. http://arxiv.org/abs/1509.06233.

Gécseg, Ferenc & Magnus Steinby. 1997. Tree languages. In Gregorz Rozenberg & Arto Salomaa (eds.), *Handbook of formal languages*, vol. 3, 1–68. New York: Springer.

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics* 25. 573–605. http://www.aclweb.org/anthology/J99-4004.

Gorn, Saul. 1967. Explicit definitions and linguistic dominoes. In *Systems and computer science, proceedings of the conference held at university of western ontario, 1965*. Toronto: University of Toronto Press.

Graf, Thomas. 2013. *Local and transderivational constraints in syntax and semantics*. UCLA dissertation. http://thomasgraf.net/doc/papers/Graf13Thesis.pdf.

Graf, Thomas & Natasha Abner. 2012. Is syntactic binding rational? In *Proceedings of the 11th international workshop on Tree Adjoining Grammars and related formalisms (TAG+11)*, 189–197. http://thomasgraf.net/doc/papers/GrafAbner12TAG.pdf.

Hale, John. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI.

Hayes, Bruce. 1995. *Metrical stress theory*. Chicago: Chicago University Press.

Heim, Irene. 1998. Anaphora and semantic interpretation: A reinterpretation of Reinhart's approach. In Uli Sauerland & O. Percus (eds.), *The interpretive tract*, vol. 25 (MIT Working Papers in Linguistics), 205–246. Cambridge, MA: MIT Press.

Heinz, Jeffrey. 2010. Learning long-distance phonotactics. *Linguistic Inquiry* 41. 623–661. https://doi.org/10.1162/LING_a_00015. http://dx.doi.org/10.1162/LING_a_00015.

Heinz, Jeffrey. 2014. Culminativity times harmony equals unbounded stress. In Harry van der Hulst (ed.), *Word stress: theoretical and typological issues*, 255–275. Cambridge, UK: Cambridge University Press.

Heinz, Jeffrey & William Idsardi. 2011. Sentence and word complexity. *Science* 333(6040). 295–297.

Hodges, Andrew. 1983. *Alan turing: the enigma*. Simon & Schuster.

Hofstader, Douglas. 1979. *Gödel, Escher, Bach: an eteneral golden braid*. 1999 Anniversay Edition. Basic Books.

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

Jackendoff, Ray. 1977. *X-bar syntax: a study of phrase structure*. Cambridge, MA: MIT Press.

Jäger, Gerhard. 2002. Gradient constraints in finite state OT: the unidirectional and the bidirectional case. In I. Kaufmann & B. Stiebels (eds.), *More than words. A festschrift for Dieter Wunderlich*, 299–325. Berlin: Akademie Verlag.

Johnson, C. Douglas. 1972. *Formal aspects of phonological description*. The Hague: Mouton.

Joshi, Aravind. 1985. Tree-adjoining grammars: How much context sensitivity is required to provide reasonable structural descriptions? In David Dowty, Lauri Karttunen & Arnold Zwicky (eds.), *Natural language parsing*, 206–250. Cambridge: Cambridge University Press.

Kaplan, Ronald M. & Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20(3). 331–378. http://www.aclweb.org/anthology/J94-3001.pdf.

Karttunen, Lauri. 1998. The proper treatment of optimality in computational phonology. In *Proceedings of the international workshop on finite state methods in natural language processing*, 1–12. Stroudsburg, PA: Association for Computational Linguistics. http://www.aclweb.org/anthology/W98-1301.

Keenan, Edward & Larry Moss. 2012. Mathematical structures in language. Ms., UCLA and Indiana University.

Kiparsky, Paul. 2002. Disjoint reference and the typology of pronouns. In Ingrid Kaufmann & Barbara Stiebels (eds.), *More than words*, vol. 53 (Studia Grammatica), 179–226. Berlin: Akademie Verlag.

Kiparsky, Paul. 2012. Greek anaphora in cross-linguistic perspective. *Journal of Greek Linguistics* 12. 84–117.

Kobele, Gregory M. 2006. *Generating copies: An investigation into structural identity in language and grammar*. UCLA dissertation. http://home.uchicago.edu/~gkobele/files/Kobele06GeneratingCopies.pdf.

Kornai, Andras. 2007. *Mathematical linguistics*. New York: Springer.

Kornai, Andras & Geoffrey K. Pullum. 1990. The X-bar theory of phrase structure. *Language* 66(1). 24–50.

Koskenniemi, Kimmo. 1983. *Two-level Morphology: A General Computational Model For Word-Form Recognition and Production*. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

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

Kracht, Marcus. 1997. Inessential features. In Alain Lecomte, F. Lamarche & G. Perrier (eds.), *Logical aspects of computational linguistics*. Berlin: Springer.

Krahmer, Emiel. 2010. What computational linguists can learn from psychologists (and vice versa). *Computational Linguistics* 36. 285–294. https://doi.org/10.1162/coli.2010.36.2.36201.

Marr, David. 1982. *Vision: a computational investigation into the human representation and processing of visual information*. Reprinted in 2010 by MIT Press. New York: Freeman. https://doi.org/10.7551/mitpress/9780262514620.001.0001.

Marr, David & Tomaso Poggio. 1976. *From Understanding Computation to Understanding Neural Circuitry*. Tech. rep. Artifical Intelligence Laboratory, MIT, AIM-357.

Martin, Andrew Thomas. 2005. *The effects of distance on lexical bias: sibilant harmony in navajo compounds*. UCLA MA thesis.

Matsui, Hiroshi, Kengo Sato & Yasubumi Sakakibara. 2005. Pair stochastic tree adjoining grammars for aligning and predicting pseudoknot RNA structures. *Bioinformatics* 21. 2611–2617.

Morawietz, Frank. 2003. *Two-step approaches to natural language formalisms*. Berlin: Walter de Gruyter. https://doi.org/10.1515/9783110197259.

Penn, Gerald. 2006. Symbolic computational linguistics: overview. In Keith Brown (ed.), *Encyclopedia of language & linguistics*, 338–352. Amsterdam: Elsevier. https://doi.org/10.1016/B0-08-044854-2/00928-7.

Pereira, Fernando C. N. & David Warren. 1983. Parsing as deduction. In *21st annual meeting of the association for computational linguistics*, 137–144.

Potts, Christopher & Geoffrey K. Pullum. 2002. Model theory and the content of OT constraints. *Phonology* 19(4). 361–393.

Prince, Alan & Paul Smolensky. 2004. *Optimality theory: constraint interaction in generative grammar*. Oxford: Blackwell.

Pullum, Geoffrey K. 2007. The evolution of model-theoretic frameworks in linguistics. In James Rogers & Stephan Kepser (eds.), *Model-theoretic syntax @ 10*, 1–10.

Pullum, Geoffrey K. & András Kornai. 2003. Mathematical linguistics. In *Oxford international encyclopedia of linguistics*, 2nd edn., 17–20. Oxford: Oxford University Press.

Riggle, Jason. 2004. *Generation, recognition, and learning in finite-state optimality theory*. University of California, Los Angeles dissertation.

Rogers, James. 1997. Strict LT_2 :Regular::Local:Recognizable. In Christian Retoré (ed.), *Logical aspects of computational linguistics: first international conference, LACL '96 (selected papers)*, vol. 1328 (Lectures Notes in Computer Science/Lectures Notes in Artificial Intelligence), 366–385. Springer.

Rogers, James. 1998. *A descriptive approach to language-theoretic complexity*. Stanford: CSLI.

Rogers, James. 2007. *Formal Description of Syntax*. Lecture Notes. http://www.cs.earlham.edu/~jrogers/files/reader.pdf.

Rounds, William C. 1970. Mappings on grammars and trees. *Mathematical Systems Theory* 4. 257–287.

Schwartz, Martin. 2008. The importance of stupidity in scientific research. *Journal of Cell Science* 121. 1771–1771. https://doi.org/0.1242/jcs.033340.

Shieber, Stuart M. 2004. Synchronous grammars as tree transducers. In *TAG+7: seventh international workshop on Tree Adjoining Grammar and related formalisms*, 88–95.

Shieber, Stuart M., Yves Schabes & Fernando C. Pereira. 1995. Principles and implementations of deductive parsing. *Journal of Logic Programming* 24. 3–36.

Sikkel, Klaas. 1997. *Parsing schemata* (Texts in Theoretical Computer Science). Berlin: Springer.

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

Stabler, Edward P. 1997. Derivational Minimalism. In Christian Retoré (ed.), *Logical aspects of computational linguistics*, vol. 1328 (Lecture Notes in Computer Science), 68–95. Berlin: Springer. https://doi.org/10.1007/BFb0052152. https://doi.org/10.1007/BFb0052152.

Stabler, Edward P. 2013. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5. 611–633. https://doi.org/10.1111/tops.12031. https://dx.doi.org/10.1111/tops.12031.

Thatcher, James W. 1967. Characterizing derivation trees for context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences* 1. 317–322.

Turing, Alan M. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42. 230–265. https://doi.org/10.1112/plms/s2-42.1.230.

Turing, Alan M. 1938. On computable numbers, with an application to the Entscheidungsproblem: a correction. *Proceedings of the London Mathematical Society* 43. 544–546. https://doi.org/10.1112/plms/s2-43.6.544.

Uemura, Yasuo et al. 1999. Tree adjoining grammars for RNA structure prediction. *Theoretical Computer Science* 210. 277–303.

Whitehill, Jacob. 2013. Understanding ACT-R — an outsider's perspective. Ms. UCSD. http://arxiv.org/abs/1306.0125.

Wilks, Yorick. 2006. Computational linguistics: history. In Keith Brown (ed.), *Encyclopedia of language & linguistics*, 761–769. Amsterdam: Elsevier. https://doi.org/10.1016/B0-08-044854-2/00928-7.