V-NYI #9: Introduction to Mathematical Linguistics

Vinny Czarnecki & Scott Nelson

6/27/2024 – 7/12/2024

Contents

Opening Remarks

Hello and welcome to Introduction to Mathematical Linguistics at V-NYI # 9. This course is part of the M/W/F block 3 and meets from 12:00–1:20 pm NY time (7:00–8:20 pm St. P/Kyiv time). Since this is an introductory course, we designed the lessons under the assumption that most students would have more background in linguistics than in math (beyond what is required in grade school). Because of this, we start off with two lessons that focus primarily on math. Then we shift gears and show how we can use various types of mathematical objects to describe various types of linguistic knowledge.

A rough outline for how we will work through these course notes is shown in the table below.

Lecture	Date	Lesson(s)
1	6/28	1
2	7/1	2
3	7/3	3
4	7/5	3
5	7/8	4
6	7/10	4

Within the text itself you will find material set off from the main text in two different color boxes. An exercise block contains exercises that we may or may not do together during the course meetings. An information block contains additional material that we think is interesting/important, but beyond the scope of the main course. At the end of the course notes is an appendix with additional exercises. Students are encouraged to attempt these exercises as homework assignments, but doing so is not required.

In preparing our lessons, we drew heavily on the notes of our colleagues who have previously taught similar material in various capacities: Siddharth Bhaskar, Jane Chandlee, Thomas Graf, Jeffrey Heinz, Adam Jardine, Jon Rawski, and Bruce Tesar. While they have certainly influenced our thinking, all mistakes and non-technical claims about the material are solely our own.

We look forward to working through this material together with you!

Lesson 1

Overview + Mathematical Background: Sets, Relations, and Functions

1.1 Topics and Goals

In this first lecture we will provide a brief overview of what mathematical linguistics is and why we think it is a useful way to study natural language. Afterwards, we will introduce some formal mathematical objects that will be used throughout the course. By the end of this lecture you should be able to describe mathematical linguistics in your own words and have a basic intuition about some of the formal tools used by mathematical linguists.

1.2 What/Why Mathematical Linguistics?

Studying linguistic phenomena with mathematics is the most simple definition one can give for mathematical linguistics, but mathematics is a large field and not all subfields are represented equally in mathematical linguistics. In this course, we will focus primarily on how set theory, logic, and automata are used to model linguistic cognition.

The following two quotes explain why being mathematically explicit is useful for linguistic theorizing. We would like to stress that we view mathematical formalization as part of the theoretical process and not a separate task left for specialists. Our goal in this course is to provide you the tools that you can use in your theorizing toolbox moving forward.

> "The aim of formalization is to cast initial ideas using mathematical expressions (again, of any kind, not just quantitative) so that one ends up with a [formalization] f – or at least a sketch of f. Once this is achieved, follow-up questions can be asked: Does f capture ones initial intuitions? Is f well defined (no informal notions are left undefined)? Does f have all the requisite properties and no undesirable properties (e.g., inconsistencies)? If inconsistencies are uncovered between intuitions and formalization, theorists must ask themselves if they are to change their intuitions, the formalization, or both...In

> practice, it always takes several iterations to arrive at a complete, unambiguously formalized f given the initial sketch" – van Rooij and Baggio (2021, p. 686).
>
> "The quality of formalization depends both on the degree of faithfulness to the original ideas and on the mathematical elegance of the resulting system. Because the proper choice of formal apparatus is often a complex matter, linguists, even those as evidently mathematical-minded as Chomsky, rarely describe their models with full formal rigor, preferring to leave the job to the mathematicians, computer scientists, and engineers who wish to work with their theories. Choosing the right formalism for linguistic rules is often very hard. There is hardly any doubt that linguistic behavior is governed by rather abstract rules or constraints that go well beyond what systems limited to memorizing previously encountered examples could explain...The only way to shed light on such issues is to develop alternative formalizations and compare their mathematical properties. – Kornai (2007, pp. 4-5).

1.2.1 Starting Assumptions

This course assumes a **Nativist** approach to linguistics, meaning that the internal, mental computations and representations that give rise to the human language faculty are part of an **internal** grammar, which is the result of a genetic endowment.

Language is a biological fact about humans in a similar way that hair follicles or extensor muscles are. Like hair follicles and extensor muscles, while communication systems among animals are undoubtedly prevalent and quite diverse, Language is a uniquely human trait.

We've evolved to have an **innate** capacity for producing and perceiving Language, and the processes and structures that give rise to this are internal to the mind.

They are also **intensional**, meaning that they make infinite use of finite means.

The majority of modern linguistics, in this tradition, aims to answer the question:

What is the nature of linguistic computations and representations?

Mathematical linguistics uses various tools from within mathematics to aid in answering precisely this question.

1.2.2 Motivating Ideas

Formal Language Theory is concerned with different classes of stringsets, treesets, or mappings thereof, and the relative differences in their complexity. This will be discussed at length in the course, but as a motivating example let's think about the following two patterns, which display the sort of abstraction we will be dealing with.

Let $\Sigma = \{a, b\}$ be an **alphabet** of symbols used to create words.

We will define a **language** L as a set of strings.

Consider the following languages:

- $L_1 = a^n b = \{ab, aab, aaab, aaaab, \dots\}$
- $L_2 = a^n b^n = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$

On the surface these look incredibly similar, but they are drastically different in terms of their complexity. The sort of machine you need to generate L_1 is less formally expressive than the one you need to generate L_2. Why is that?

Imagine I have two buttons. When I hit the first button I receive an a, and when I hit the second button I receive a b– and any time I hit either button, the character gets tacked on to a string.

Generating L_1:

I can hit Button 1 as many times as I want without keeping track, as long as I hit Button 2 only once. Then and only then is my string in the language. Any other combination of hitting these buttons will not be in the language.

Generating L_2:

I can hit Button 1 as many times as I want; however, how many times I hit Button 2 *necessarily* depends on how many times I hit Button 1. If I hit Button 1 thirty times, I need to hit Button 2 thirty times for my string to be in the language.

The difference here is that generating L_2 requires *keeping a memory of arbitrarily many characters*. If you've seen n a's, you have to know that you've seen n a's to get the right amount of b's. In generating L_1, you don't need this kind of memory.

L_1 is a **Regular** language which means it can be recognized by a Finite State Acceptor, whereas L_2 is a **Context-Free** language which means it can be recognized by a Pushdown Automaton (meaning that L_2 is of the same sort of compelxity as the early theories of syntax).

Throughout the course we will see other ways of expressing these complexities using logic and automata.

This is just one example of the difference in complexities of formal languages.

The **Chomsky-Schutzenberger Hierarchy** is a hierarchy of language classes that differ in complexity.

Regular < Context-Free < Context-Sensitive < Recursively Enumerable

Any language that is recognized by a:

- Finite State Acceptor is Regular.
- (non-deterministic) Pushdown Automaton is Context-Free.
- Linear Bounded Automaton is Context-Sensitive.
- Turing Machine is Recursively Enumerable.

Throughout the course, we will go into much greater detail about what these things mean, but for now the main point is that languages (and mappings) differ in terms of their complexity, and the expressive power required to generate or recognize them differs.

1.2.3 The Subregular Hierarchy

Early research relating Formal Language Theory to linguistic theory placed a lower complexity bound at the Regular languages; however, more recently, research in phonology and morphology has resulted in a more nuanced understanding of less complex, *subregular* classes (Chandlee, 2017; Heinz, 2018).

Phonological generalizations were previously thought to be Regular, meaning phonotactic well-formedness can be recognized by Finite State Acceptors and phonological processes can be carried out using Finite State Transducers. But is this a strong enough hypothesis? The Subregular Hypothesis conjectures that phonological generalizations exist within classes and mappings that exist lower wihtin this hierarchy than Regular.

Below, **The Subregular Hierarchy**, a refinement of classes beneath the Regular languages is shown:

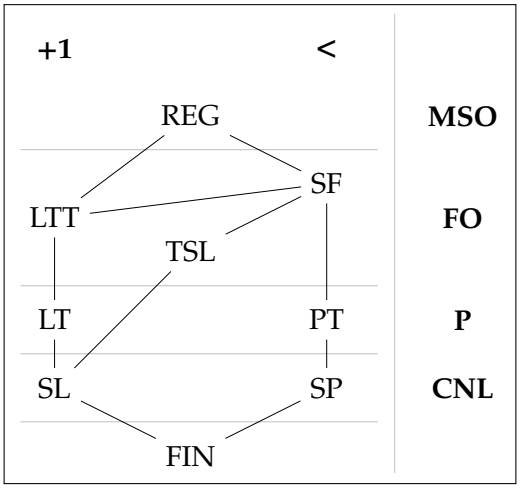


Figure 1.1: The Subregular Hierarchy; +1 refers to constraints defined using immediate successor, whereas < referes to constraints defined using general precedence; MSO refers to Monadic Second Order Logic, FO refers to First Order Logic, P refers to Propositional Logic, and CNL refers to Conjunction of Negative Literals; Combinations of logic and representation result in different language classes (explained in more detail in Lecture 3).

Expressing linguistic generalizations through the lens of stringsets within this hierarchy has lead to a great deal of research within computational phonology, morphology, and more recently, computational syntax.

In this course, we will see the tools that lead to understanding the impact of this research, and how to contribute to it!

1.3 Sets

A **set** is a well-defined collection of objects. Objects in this set are called *members* or *elements*.

Sets can have *intensional* or *extensional* descriptions.

Consider the following examples:

	Intensional	Extensional
Odd_{pos}	$\{2x+1 \mid x \in \mathbb{N}\}$	$\{1, 3, 5, 7, \dots\}$
L_1	$\{a^n b \mid n \in \mathbb{N}\}$	$\{b, ab, aab, aaab, aaaab, \dots\}$
NYI_{fac}	$\{x \mid x$ has been an NYI faculty member$\}$	{John Bailyn, Jerry Fodor, David Pesetsky,…}

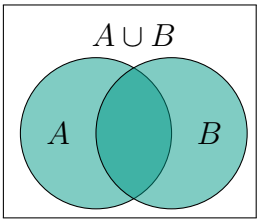
Table 1.1: Examples of Sets

To say that a is an element of the set A, we say $a \in A$ (a is in A) and to say that it is not an element, we say $a \notin A$ (a is not in A).

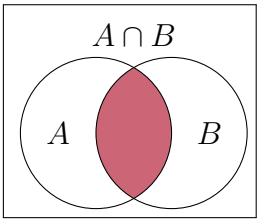
Thinking About Set Membership

1. What are some statements that hold for these sets?
2. What are some statements that don't hold?

The **union** of two sets A and B, denoted $A \cup B$, is the set containing all elements that are either in A or in B.



The **intersection** of two sets A and B, denoted $A \cap B$, is the set containing all elements that are only in both A and in B.



Vowel Inventories - Union and Intersection

Consider the following vowel inventories:

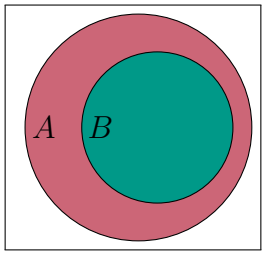
West Greenlandic (Hagerup, 2011): {i, a, u}
Malagasy (Howe, 2021): {i, e, a, o, u}
Swedish (Riad, 2014): {i, y, e, ø, ɛ, ɑ, o, u, ʉ}

Answer the following questions:

1. What is the union of West Greenlandic and Swedish?
2. What is the intersection of Malagasy and Swedish?
3. What is the intersection of West Greenlandic and Malagasy?

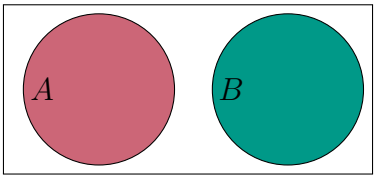
A set B is called a **subset** of a set A if every element of B is also an element of A. This is denoted by $B \subseteq A$. If every element of B is also an element of A and $B \neq A$ then we say B is a **proper subset** of A. This is denoted by $B \subset A$.

A set A is called a **superset** of a set B if every element of B is also an element of A. This is denoted by $A \supseteq B$. If every element of B is also an element of A and $B \neq A$ then we say A is a **proper superset** of B. This is denoted by $A \supset B$.

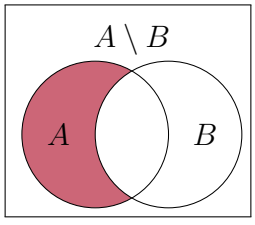


The set with no elements $\varnothing$ is called the **empty set**.

Two sets A and B are called **disjoint** if they have no elements in common. Namely, if it is the case that $A \cap B = \varnothing$.



The **Set Difference** of two sets A and B is denoted by $A \setminus B$ (sometimes $A - B$) and represents the set of all elements that are in A but not in B, shown below:



Vowel Inventories - Subset/Superset/Difference

Consider the same vowel inventories from above:
West Greenlandic (Hagerup, 2011): {i, a, u}
Malagasy (Howe, 2021): {i, e, a, o, u}
Swedish (Riad, 2014): {i, y, e, ø, ɛ, ɑ, o, u, ʉ}

Answer the following questions:
1. Which vowel inventories stand in a subset/superset relationship?
2. Are any of the vowel inventories disjoint with another?
3. What is the set difference of *West Greenlandic* and *Swedish*? And vice versa?

The **Cartesian Product** of two sets A, B is

$A \times B = \{(a, b) \mid a \in A,\ b \in B\}$, which denotes the set of all ordered pairs where the first element is in A and the second element is in B.

More generally, the **n-fold Cartesian Product** of n sets is

$X_1 \times X_2 \times \cdots \times X_n = \{(x_1, x_2 \ldots, x_n) \mid x_1 \in X_1,\ x_2 \in X_2, \ldots,\ x_n \in X_n\}$, which denotes the set of all ordered tuples where the n-th element is in X_n.

The **Power Set** of a set A is

$\mathcal{P}(A) = \{B \mid B \subseteq A\}$, which denotes the set containing all of the subsets of A (including the empty set).

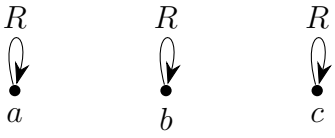
1.4 Relations

We define a **Relation** with arity n as a subset of the Cartesian Product $X_1 \times \cdots \times X_n$. A **Binary Relation** R between sets X and Y is therefore an element of $\mathcal{P}(X \times Y)$. We denote this as $R \subseteq X \times Y$ and will write either xRy or $R(x, y)$ to mean $(x, y) \in R$.

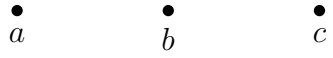
There are certain important properties that a relation R may hold over a set X:

- **Reflexive** - for all $x \in X$, xRx
- **Irreflexive** - for all $x \in X$, it is not the case that xRx
- **Symmetric** - for all $x, y \in X$, if xRy then yRx
- **Antisymmetric** - for all $x, y \in X$, if xRy and yRx then $x = y$
- **Asymmetric** - for all $x, y \in X$, if xRy then it is not the case that yRx
- **Transitive** - for all $x, y, z \in X$, if xRy and yRz then xRz
- **Connected** - for all $x, y \in X$, if $x \neq y$ then xRy or yRx

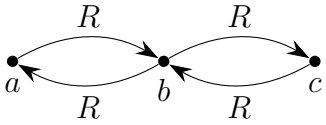
Some schematizations of these different properties above are shown below for an arbitrary set $\{a, b, c\}$.



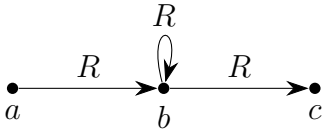
Reflexive



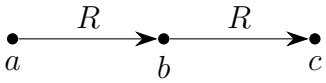
Irreflexive



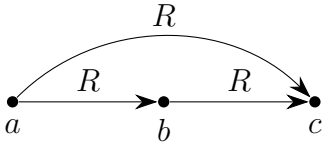
Symmetric



Antisymmetric



Asymmetric



Transitive

Thinking about Relations

Can you think of any examples of relations that meet each of these properties, linguistic or non-linguistic?

Note: Important Combinations of Properties

We are also interested in relations that satisfy certain combinations of these properties: **Equivalence Relation** - reflexive, symmetric, transitive; **Partial Order** - reflexive, antisymmetic, transitive; **Strict Partial Order** - irreflexive, asymmetric, transitive; **Total Order** - reflexive, antisymmetric, transitive, connected; **Strict Total Order** - irreflexive, asymmetric, transitive, connected

1.5 Functions

Formally, a **function** is a special type of relation $f \subseteq X \times Y$ such that:

$$\forall x, y, y'.((x, y) \in f \wedge (x, y') \in f) \rightarrow y = y'$$

More simply, a function is a relation where one element in a set X can be related to *at most one other element* in a set Y. Note, the converse does not need to hold.

We say that a function f is **undefined** for an element $x \in X$ if there is no pair $(x', y) \in f$ such that $x = x'$, otherwise f is *defined* for x.

When f is defined for x we write $f(x) = y$ or sometimes $x \mapsto_f y$. Equivalently, we may refer to function f as a *map* and say f maps x to y.

If $\forall x \in X.(x, y) \in f$ then we say that f is a **total function**, otherwise f is a **partial function**.

The set X is called the **domain** and the set Y is called the **co-domain** (or range) of the function $f : X \to Y$. Relatedly, the **image** of f is the set $\{f(x) \in Y \mid x \in X \wedge f(x) \text{ is defined}\}$ and the **pre-image** of f is the set $\{x \in X \mid f(x) \text{ is defined}\}$. In other words, the pre-image is the subset of the domain where the function is defined and the image is the subset of the co-domain where the function is defined. This is schematized below in Figure 1.2.

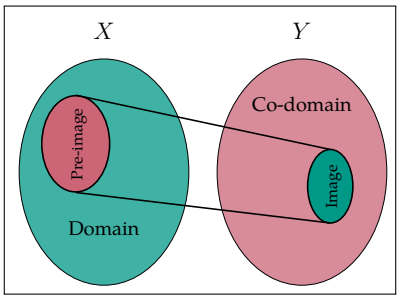


Figure 1.2: A function from X to Y

A function $f : X \to Y$ is **injective** or *one-to-one* if for all $y \in Y$, the pre-image $f^{-1}(y)$ has at most one element. In other words, f maps at most one x to any given y.

A function $f : X \to Y$ is **surjective** or *onto* if for all $y \in Y$, the pre-image $f^{-1}(y)$ is non-empty. In other words, f maps onto every element of Y.

A function $f : X \to Y$ **bijective** it is both injective and surjective. In other words, every element in Y is in correspondence with a unique element in X.

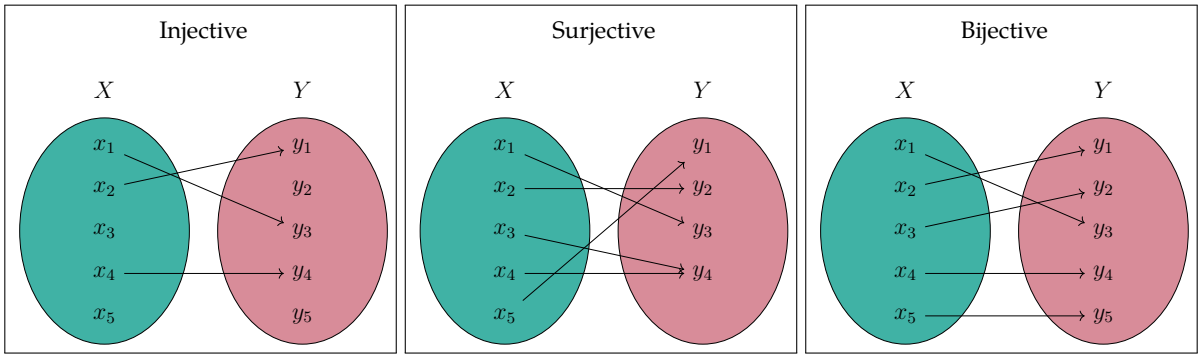


Figure 1.3: Examples of Injectivity, Surjectivity, Bijectivity

Phonological Neutralization and Injectivity

Neutralization occurs when a phonological process eliminates a distinction between two morphemes that was held at a previous level. Consider these data from German **final devoicing** (Dinnsen and Garcia-Zamor, 1971):

1. (a) /bad+en/ → [baden] 'to bathe'
   (b) /bad/ → [bat] 'bath'
   (c) /bat+en/ → [baten] 'asked'
   (d) /bat/ → [bat] 'ask'

Notice how the morphemes for 'bath' and 'ask' contrast at the underlying level but are the same at the surface level? This is neutralization. If we think of phonology as a function, explain why the phonology function is not injective (and therefore not bijective). Hint: this has to do with neutralization!

Given a function $f : X \to Y$, where X, Y are both uniform sets such that each member of each individual set is the same type of thing, we say that the function f has **type** X to Y.

Typed Functions and The λ-Calculus

The idea of typed functions stems from the λ-calculus (Church, 1932, 1933), which is a universal abstract computing device that is equivalent to a Turing machine (Turing, 1936).

In linguistics, the λ-calculus plays a central role in many theories of formal semantics. When you have a predicate in English like *runs*, this can be represented by a function using the λ-calculus as:

$$[\![\text{runs}]\!] = \lambda x.[x \text{ runs}]$$

But we would also say this has the type $\langle e, t\rangle$, indicating that it is a function from individuals to truth values. Equivalently, we can write it as $\texttt{runs} : e \to t$.

Given two functions $f : X \to Y$ and $g : Y \to Z$ such that the domain of g is the co-domain of f, we can define the **composition** of f and g as a single function $g \circ f : X \to Z$. this is equivalent to first applying f to an input x and then applying the output of f to g, or $g(f(x))$.

Function composition is *associative*: $h \circ g \circ f = h \circ (g \circ f) = (h \circ g) \circ f$, meaning that for three functions it can apply to any two followed by the third.

Commutativity and Function Composition: Derivational Morphology

1. We know that function composition is always associative (convince yourself of this!), but is it always commutative? Namely, is it always the case that $f \circ g = g \circ f$? Why or why not?

2. Consider the sets:
   $V = \{\text{lock, interpret, rely, } \dots\}$,
   $Adj = \{\text{lockable, interpretable, reliable, } \dots\}$,
   $Noun = \{\text{lockability, interpretability, reliability, } \dots\}$

   and the two functions:
   -able : $\texttt{string} \to \texttt{string}$ -ity : $\texttt{string} \to \texttt{string}$

   Given that these two functions add 'able' or 'ity' to the end of a string, is it the case that -able ∘ -ity = -ity ∘ -able?

3. **Bonus**: If we change the function types to match the parts of speech, what do you think might go wrong with the composition? (hint: types)

1.6 Closing Thoughts and Further Reading

Closing Thoughts

In this lecture, we gave an overview of our ideas about mathematical linguistics and how it can be viewed as a complementary tool in the theoretical linguists toolbox, and then introduced some of the basic tools commonly employed by mathematical linguists. We familiarized ourselves with sets, relations, and functions and showed how to use them to reason about simple linguistic data .

Further Reading

For approachable (non-linguistic) resources on some of the fundamentals of sets, relations, functions, logic, basic proof techniques, etc. see Balakrishnan (2012); Eccles (2013); Velleman (2019). For those interested, these also contain many other useful concepts that we won't cover for time reasons. More in-depth resources on the specific tools we use throughout the course will be given in further secitons.

For more on the underlying linguistic assumptions we make regarding what exactly it is that we're interested in formalizing, as well as some basic exposure to some of the formal tools we've seen, and how it all relates to cognitive science more generally, see Isac and Reiss (2013).

Lesson 2

Mathematical Background: Logic, Recursive Data Structures, Automata

2.1 Topics and Goals

In this second lecture we will continue to introduce the formal mathematical objects that will be used throughout the rest of the course. Today's topics expose us to two main pieces of machinery used in current linguistic theorizing: logic and automata. These topics are intimately connected but provide us with two different ways to understand various types of linguistic structure. By the end of this lecture you should be comfortable reasoning about the various mathematical concepts discussed and be ready to start applying them to novel linguistic data.

2.2 Logic

Here, we will briefly introduce Propositional Logic as a building block for using logic more generally later in the course.

Simply put, a **proposition** is a statement that is either true or false, but cannot be both.

Propositions

Of the following statements below, which are propositions? Of those that are propositions, which are true and which are false?

1. John is tall.
2. x is tall.
3. 't' is a voiced segment.
4. x is a voiced segment.

Something cannot be a proposition if it has a value that has not been specified.

Logical languages are built from two main components: a *syntax* and a *semantics*. The syntax defines the form that terms of the language take, and the semantics defines how those forms are interpreted logically. In the syntax of a logical language, there are atoms which build up to make larger elements called *well formed formulas* (or wffs).

In propositional logic, these atoms are propositions $P, Q, R, \ldots$ and a well formed formula is a well-defined string (defined below) of these atoms combined with the following logical connectives.

$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$
not	and	or	implies	iff

Table 2.1: Logical Connectives

Also note that the connectives $\vee, \rightarrow$, and $\leftrightarrow$ can all be defined using $\neg$ and $\wedge$, thus only these two are included in the definition of a wff below. The truth tables below show how each of the connectives above are interpreted, where the values 0 and 1 correspond to false true, respectively.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Table 2.2: Truth Tables for the Logical Connectives Mentioned Above

Suppose ϕ, ψ are two atomic formulas, then:

- ϕ is a wff
- $\neg\phi$ is a wff
- $(\phi \wedge \psi)$ is a wff

A few examples of wffs in propositional logic are shown below:

- P
- $(P \wedge \neg Q)$
- $\neg((P \wedge \neg Q) \wedge R)$

Ill Formed Formulas

What are some examples of strings that are not generated by our syntax? Namely, what are some examples of *ill*-formed formulas?

Latin exclusive and inclusive 'or'

Given two propositions P, Q and an arbitrary binary operator $\bullet$, there are 16 logically possible ways to interpret $P \bullet Q$. The connectives mentioned above are a subset of these 16 that are commonly employed in natural language, but some languages do make use of others.

The $\vee$ is what is called 'inclusive or', but there is a variant $\oplus$ called 'XOR' which stands for 'exclusive or', defined by the truth table below (either P or Q but not both):

P	Q	$P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

Latin is one natural language which had this distinction encoded lexically with *vel* standing for inclusive or and *aut* standing for exclusive or.

Defining Connectives with Connectives

1. Use the two connectives $\neg$ and $\wedge$ to define the remaining connectives $\vee, \rightarrow, \leftrightarrow$
2. Can all the other connectives be defined using just $\neg$ and $\vee$ as well?
3. **Bonus**: There is a single connective $\dagger$ which is sometimes called 'NAND' and is defined with the following truth table:

P	Q	$P \dagger Q$
0	0	1
0	1	1
1	0	1
1	1	0

Can you define *all* of the connectives above using *only* $\dagger$?

As of right now, these examples are just meaningless strings. These strings are given meaning through an **assignment** or **interpretation** function.

Let $\mathbb{P}$ be the set of all propositions and let $i : \mathbb{P} \rightarrow \{0, 1\}$ be a function from propositions to truth values that assigns a truth value to a given proposition.

So if $P \in \mathbb{P}$ is a proposition, then $i(P) = 1$ means P is true and $i(P) = 0$ means P is false. These are called truth valuations, since they are statements about the truth of a proposition.

Propositional Logic: Interpretation

Observe the following propositions given the elements {k,t,æ}:

$$\mathbb{P} = \left\{ \begin{array}{ll} P_1 & = \text{k is a voiceless segment,} \\ P_2 & = \text{æ is a voiced segment,} \\ P_3 & = \text{t is a voiceless segment,} \\ P_4 & = \text{k is a voiced segment,} \\ P_5 & = \text{k comes before æ,} \\ P_6 & = \text{æ comes before t,} \\ P_7 & = \text{t comes before æ,} \\ P_8 & = \text{æ comes before k,} \\ & \vdots \\ P_n & = \ldots \end{array} \right\}$$

1. Define the truth conditions for $P_1 - P_4$ given the standard interpretation of these IPA symbols.
2. Define the truth conditions for $P_5 - P_8$ supposing we wanted to model the English word 'cat'.
3. How would the previous truth table differ if we were modeling the English word 'tack' instead.
4. **Bonus**: Suppose we wanted to model the English word 'act'. How would you interpret P_6? (hint: does the precise definition of 'comes before' affect your answer?)

A binary **satisfaction** relation $\vDash$ between truth valuations and formulas specifies when a formula defined by our syntax is true or false.

$i \vDash \phi$ means that ϕ is true under i, or i models/satisfies ϕ.

$i \nvDash \phi$ means that ϕ is false under i, or i does not model/satisfy ϕ.

As we will see, there are various properties that we can add or remove from propositional logic that will change its expressive power. For example, by adding quantification over variables, we get a stronger logic called **First-Order Logic** (also called Predicate Logic). Instead, if we limit the connectives to only conjunction ($\wedge$) and restrict negation ($\neg$) to only atomic elements (propositions), then we get a weaker logic called **Conjunction of Negative Literals**.

Later in the course, we will see how propositional logic and and other logics (both more and less powerful) can be used to define certain language classes (and mappings) that are very useful for linguistic theorizing.

Propositional Logic: Satisfaction

Recall the propositions from the previous examples:

$$\mathbb{P} = \left\{ \begin{array}{ll} P_1 & = \text{k is a voiceless segment}, \\ P_2 & = \text{æ is a voiced segment}, \\ P_3 & = \text{t is a voiceless segment}, \\ P_4 & = \text{k is a voiced segment}, \\ P_5 & = \text{k comes before æ}, \\ P_6 & = \text{æ comes before t}, \\ P_7 & = \text{t comes before æ}, \\ P_8 & = \text{æ comes before k}, \\ & \vdots \\ P_n & = \ldots \end{array} \right\}$$

1. Given the i function you defined in the first two questions of the previous example (modeling 'cat'), does i model the following formulas ($i \vDash \phi$)?

   - $\phi_1 := P_1 \wedge P_5$
   - $\phi_2 := (P_7 \vee P_8) \rightarrow P_1$
   - $\phi_3 := \neg P_2$
   - $\phi_4 := (\neg P_7 \wedge P_6) \rightarrow P_3$

2. Given the i function you defined in the first and third questions of the previous example (modeling 'tack'), does i model the same formulas?

3. **Bonus**: Define a new i function grounded in any reality you choose (be creative) and come up with some wff's that it both models and does not model.

2.3 Recursive Data Structures

A **recursive data structure** is a type of data that contains other data of the same type. We will focus on two types of structures: strings and trees.

2.3.1 Strings

A **string** is a sequence of symbols. Symbols can be anything: IPA letters, parts of speech, morphemes, words, and so on. Symbols are drawn from a set called the *alphabet* which is represented as Σ. To define strings recursively, we rely on the constructor $(\cdot)$ and the empty string λ. The recursive definition of a string is as follows:

Base Case: λ is a string
Inductive Case: If $\sigma \in \Sigma$ and w is a string, then $\sigma \cdot (w)$ is a string

As an example, suppose we have an alphabet $\Sigma = \{a, b, c\}$. We construct the string bac in the following way:

$$b \cdot (a \cdot (c \cdot (\lambda)))$$

We then can recursively check whether this is a valid string using the following steps:

1. Inductive Case: $\sigma = b \in \Sigma$; $w = (a \cdot (c \cdot (\lambda)))$
2. Inductive Case: $\sigma = a \in \Sigma$; $w = (c \cdot (\lambda))$
3. Inductive Case: $\sigma = c \in \Sigma$; $w = (\lambda)$
4. Base Case: λ is a string

The validity of the base case percolates backwards to validate the inductive cases. In other words bac is a valid string because ac is a valid string, ac is a valid string because c is a valid string, and c is a valid string because the empty string is a valid string.

Representing strings as recursive data structures allows us to define sub-properties of strings recursively as well. For example, the length of a string w, written $|w|$ is defined as follows:

Base Case: If $w = \lambda$, then $|w| = 0$
Inductive Case: Otherwise, $w = \sigma \cdot (x)$ and $|w| = 1 + |x|$

Returning to our string bac, we recursively calculate its length the following way:

1. Inductive case: $w = b \cdot (ac)$; $|bac| = 1 + |ac|$
2. Inductive case: $w = a \cdot (c)$; $|ac| = 1 + |c|$
3. Inductive case: $w = c \cdot (\lambda)$; $|c| = 1 + |\lambda|$
4. Base case: $w = \lambda$; $|w| = 0$

The length of bac is 1 + the length of ac, the length of ac is 1 + the length of c, the length of c is 1 + the length of the empty string which is 0. The length is ulitmatley determined by the recursive application of the function which essentially counts the number of symbols and terminates once it reaches the empty string.

2.3.2 Trees

A **tree** extends the idea of a string from one dimension to two. Recall that a string is a sequence of symbols, so in two dimensions this becomes a sequence of a sequence of symbols. The first dimension is linear order, the second dimension is dominance. In other words, our structures now encode *siblinghood* and *parenthood*.

As was the case with strings, we once again will assume an alphabet of symbols Σ. Whereas we had the empty string λ, there is no equivalent empty tree. This is because each tree must have a root. The closest we have is a **leaf** which is a tree with no daughters. We represent a leaf as $\sigma[\lambda]$. At times we will represent leaves simply as $\sigma[\,]$.

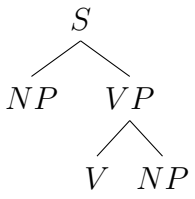
The recursive definition of a tree is as follows:

Base Case: If $\sigma \in \Sigma$ then $\sigma[\lambda]$ is a tree
Inductive Case: If $\sigma \in \Sigma$ and w is a string of trees, then $\sigma[w]$ is a tree

Notice this definition relies on our previous definition of strings. Suppose we have an alphabet $\Sigma = \{S, NP, VP, V\}$. We construct the tree $S[NP[\lambda]\ VP[V[\lambda]\ NP[\lambda]]]$ in the following way:

$$S[NP[\lambda] \cdot (VP[V[\lambda] \cdot (NP[\lambda] \cdot (\lambda))] \cdot (\lambda))]$$

A more recognizable, 2D visualization of this tree is shown below (note that the λ need not be shown since it is implied by the lack of a child):



Note that our inductive case has two conditions: is the root symbol in our alphabet and is w a string of trees. Checking the first condition is easy, but checking the second condition is where the recursion really comes into play. We check whether the above is a valid tree using the following steps:

1. Inductive Case: $\sigma = S \in \Sigma$; $w = NP[\lambda] \cdot (VP[V[\lambda] \cdot (NP[\lambda] \cdot (\lambda))] \cdot (\lambda))$

   The next step requires us to check each element in our string

2. (a) Base Case: $NP \in \Sigma$; $NP[\lambda]$ is a tree

   (b) Inductive Case: $\sigma = VP \in \Sigma$; $w = V[\lambda] \cdot (NP[\lambda] \cdot (\lambda))$

   Once again, we check every element in the string

3. (a) Base Case: $V \in \Sigma$; $V[\lambda]$ is a tree

   (b) Base Case: $NP \in \Sigma$; $NP[\lambda]$ is a tree

Essentially, we recurse through the tree looking for the leaves. Since leaves are valid trees, they validate the higher level structures as long as the root of each tree is drawn from the alphabet. Specifically in this case, because $V[\lambda]$ and $NP[\lambda]$ are valid trees and $VP \in \Sigma$, then $VP[V[\lambda]\ NP[\lambda]]$ is a valid tree. Since this is a valid tree, $NP[\lambda]$ is a valid tree, and $S \in \Sigma$, then $S[NP[\lambda]\ VP[V[\lambda]\ VP[\lambda]]]$ is a valid tree.

We can use our recursive definition of trees to also define the set of all trees of depth n. This is expressed with the following definition:

$$T_0 = \{a[\lambda] \mid a \in \Sigma\}$$
$$T_{n+1} = \{a[w] \mid a \in \Sigma, w \in T_n^*\} \cup T_n$$

Similarly, we denote the set of all logically possible trees as $\Sigma^T = \bigcup_{i \in \mathbb{N}} T_i$.

Recursive Tree Functions

Above, we showed how to recursively compute the length of a string. This was possible due to the recursive nature of the data structure. We can do similar computations over trees with what are called divide-and-conquer algorithms.

1. The **yield** of a tree is the concatenation of all of its leaves. Write a recursive function `yield`(T) that computes the yield of a tree T.
2. The **depth** of a tree is the maximum number of levels between the root and lowest leaf. Write a recursive function `depth`(T) that computes the depth of a tree T.
3. The **size** of a tree is the total number of all nodes within it. Write a recursive function `size`(T) that computes the size of a tree T.
4. **Bonus**: You may have noticed some similarities between these functions. Using this intuition, how might you go about defining a `single` function `info`(T) that combines all of this information with only a single traversal?

2.4 Automata

Automata are abstract computing machines. There are two general types of problems that can be solved with an automaton:

1. The Membership Problem
2. The Transformation Problem

The Membership Problem: assume a set X of strings (or trees); given an input string (or tree) x, is $x \in X$?

The Transformation Problem: assume a string to string (or tree to tree) function $f : X \to Y$; given an input string (or tree) x, what is $f(x)$?

In this course we will focus primarily on *finite state machines* which are automata with a fixed number of states and no additional memory device. A **finite state acceptor** (FSA) solves the membership problem while a **finite state transducer** (FST) solves the transformation problem.

Finite state machines can have many additional properties such as deterministic vs. nondeterministic, 1-way vs. 2-way application (for strings), and bottom-up vs. top-down application (for trees).

2.4.1 String Acceptors

We will begin by looking at a 1-way, deterministic, finite state string acceptor (1DFSA). Formally, a 1DFSA is a mathematical object represented as a five-tuple $\langle \Sigma, Q, q_o, \delta, F \rangle$ where:

$$\begin{aligned} \Sigma &\text{ is the alphabet/set of symbols} \\ Q &\text{ is a set of states} \\ q_0 \in Q &\text{ is a single initial state that is a member of } Q \\ \delta &\text{ is a transition function: } \Sigma \times Q \rightarrow Q \\ F \subseteq Q &\text{ is a set of final states that is a subset of all the states} \end{aligned}$$

δ^* is a recursive extension of the transition function with domain $\Sigma^* \times Q$:[1]

$$\begin{aligned} \delta^*(\lambda, q) &= q \\ \delta^*(\sigma w, q) &= \delta^*(w, \delta(\sigma, q)) \end{aligned}$$

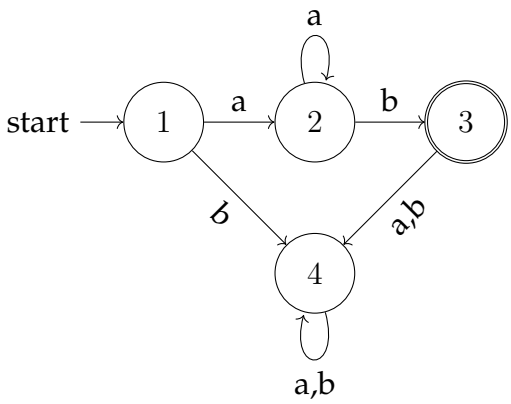
Given a 1DFSA A and a string $s \in \Sigma^*$, if $\delta^*(s, q_0) \in F$ then we say that A accepts w (alternatively: recognizes,generates). $L(A) = \{s \in \Sigma^* \mid \delta^*(s, q_0) \in F\}$ is the **language** (stringset) accepted/recognized/generated by A.

Let's return to the formal language $a^n b$ discussed above. We define the 1DFSA for this language as follows:

$$\begin{aligned} \Sigma &= \{a, b\} \\ Q &= \{1, 2, 3, 4\} \\ q_0 &= 1 \\ \delta &= \{((a,1),2), ((a,2),2), ((a,3),4), ((a,4),4), ((b,1),4), ((b,2),3), ((b,3),4), ((b,4),4)\} \\ F &= \{3\} \end{aligned}$$

While this is the formal definition of the automaton that accepts the $a^n b$ language, it is incredibly opaque. In general, we will represent automata visually as directed graphs. Each node in the graph represents a unique state in Q. The transition function δ is encoded with labeled edges. The initial state q_0 is shown with an ingoing arrow and final states F are circled twice.

[1] Recall from above that a string s can be decomposed into its initial symbol σ and the remaining material w.



Suppose we wanted to check if the string $aaab$ was in the language. In order to do so, we see if $\delta^*(aaab, 1) = 3$:

$$\begin{aligned}
\delta^*(aaab, 1) &= \delta^*(aab, \delta(a, 1)) \\
&= \delta^*(aab, 2) \\
&= \delta^*(ab, \delta(a, 2)) \\
&= \delta^*(ab, 2) \\
&= \delta^*(b, \delta(a, 2)) \\
&= \delta^*(b, 2) \\
&= \delta^*(\lambda, \delta(b, 2)) \\
&= \delta^*(\lambda, 3) \\
&= 3
\end{aligned}$$

Now, if we were to do the same check for the string $abab$, we will see that we end up in state 4 which is not a final/accepting state:

$$\begin{aligned}
\delta^*(abab, 1) &= \delta^*(bab, \delta(a, 1)) \\
&= \delta^*(bab, 2) \\
&= \delta^*(ab, \delta(b, 2)) \\
&= \delta^*(ab, 3) \\
&= \delta^*(b, \delta(a, 3)) \\
&= \delta^*(b, 4) \\
&= \delta^*(\lambda, \delta(b, 4)) \\
&= \delta^*(\lambda, 4) \\
&= 4
\end{aligned}$$

The processing of $abab$ lands us in state 4 which is not an accepting state. State 4 in this automaton is what is called a *sink state*. This is a non-accepting state that, once reached,

cannot be left. Therefore, we can interpret any edge to a sink-state as the moment that a string becomes unacceptable. For example, in A above reading a b as the initial symbol takes us to the sink state since only strings that start with a are in the language. Additionally, reading any symbol after reading an initial string of a's and one b sends us to the sink state since we want to end with exactly one b.

2.4.2 Tree Acceptors

Let's move now from trees to strings. We will start by looking at a bottom-up, deterministic, finite state tree acceptor (BDFTA). Formally, a BDFTA is a mathematical object represented as a four-tuple $\langle \Sigma_r, Q, \delta, F \rangle$ where

$$\begin{aligned} \Sigma_r &\text{ is a ranked alphabet} \\ Q &\text{ is a set of states} \\ \delta &\text{ is a transition function: } \Sigma \times Q^* \to Q \\ F \subseteq Q &\text{ is a set of final states that is a subset of all the states} \end{aligned}$$

As was the case with string acceptors, we recursively extend the transition function to $\delta^*$with domain $\Sigma^T \times Q$:[2]

$$\begin{aligned} \delta^*(a[\lambda]) &= \delta(a, \lambda) \\ \delta^*(a[t_1 \cdots t_n]) &= \delta(a, \delta^*(t_1) \cdots \delta^*(t_n)) \end{aligned}$$

This time, let's return to the other formal language $a^n b^n$ discussed above. There is no finite state *string acceptor* that will accept this *string language*. Instead, we will focus on a tree language with trees that yield this string language. We define the BDFTA for this language as follows:

$$\begin{aligned} \Sigma &= \{a, b, S\} \\ Q &= \{q_a, q_b, q_S\} \\ \delta &= \{((a, \lambda), q_a), ((b, \lambda), q_b), ((S, q_a q_b), q_S), ((S, q_a q_S q_b), qS)\} \\ F &= \{q_S\} \end{aligned}$$

A ranked alphabet is simply an alphabet where each symbol has an *arity* n, indicating that it must take n arguments (for trees, have n children). The symbols a and b both take 0 arguments since they have no children whereas our S symbol takes either 2 or 3 arguments because it can either have 2 children a, b or 3 children a, S, b.

[2]Recall from above that Σ^T refers to the set of all finite trees.

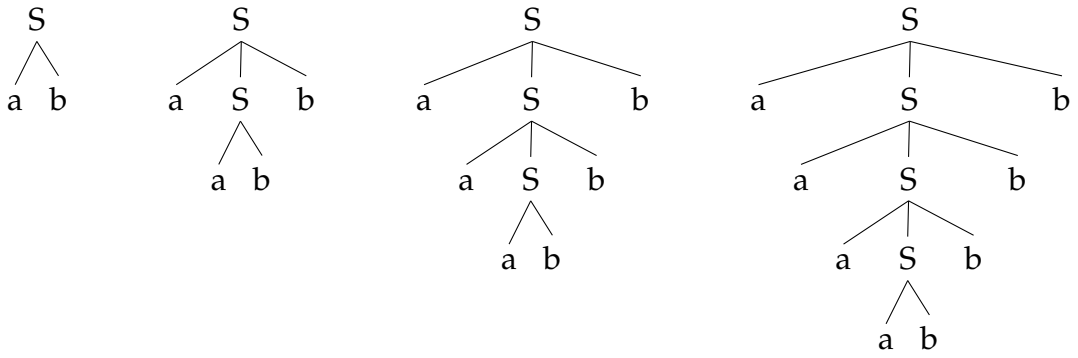
Arity: Functions vs. Relations

Standardly, a ranked alphabet Σ_r uses a function $\texttt{Arity} : \Sigma \to \mathbb{N}$ to determine the arity $\texttt{Arity}(\sigma) = n$ of each symbol $\sigma \in \Sigma$.

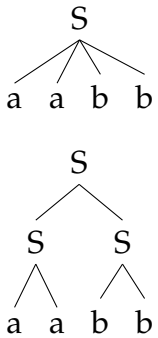
Elements with arity 0 are constants, with arity 1 are unary, with arity 2 are binary, and so on. In the example above, we see that the element S takes either 2 or 3 arguments. What changes about the alphabet if we consider Arity as a function or relation?

Unlike string acceptors, graph representations of tree acceptors do not help with interpretability (though see Lambert (2022, chapter 7) for a recent proposal). Instead, we will rely primarily on the processing function δ^* to analyse the acceptability of a given tree.

Some trees from the tree language accepted by our BDFTA are below:



The *yield* of our trees all come from the string language $a^n b^n$, but we can derive similar string yields with different tree structures. For example, the following trees S[a a b b] and S[S[a a] S[b b]] which are accepted by very different acceptors also yield $aabb$:



Now, let's look at what the process function shows for the trees S[a [S[a b]] b] and S[a a b b]. We'll start with the valid tree.

$$\begin{aligned}
\delta^*(S[a[S[ab]]b]) &= \delta(S, \delta^*(a[\lambda])\ \delta^*(S[a\ b])\ \delta^*(b[\lambda])) \\
&= \delta(S, \boldsymbol{\delta(a, \lambda)}\ \delta^*(S[a\ b])\ \delta^*(b[\lambda])) \\
&= \delta(S, \boldsymbol{q_a}\ \delta^*(S[a\ b])\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \boldsymbol{\delta(S, \delta^*(a[\lambda])\ \delta^*(b[\lambda]))}\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \delta(S, \boldsymbol{\delta(a, \lambda)}\ \delta^*(b[\lambda]))\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \delta(S, \boldsymbol{q_a}\ \delta^*(b[\lambda]))\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \delta(S, q_a\ \boldsymbol{\delta(b, \lambda)})\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \delta(S, q_a\ \boldsymbol{q_b})\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \boldsymbol{q_S}\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ q_S\ \boldsymbol{\delta(b, \lambda)}) \\
&= \delta(S, q_a\ q_S\ \boldsymbol{q_b}) \\
&= \boldsymbol{q_S}
\end{aligned}$$

Notice, we end in state q_S which we know to be an accepting state. The "bottom-up" aspect of the acceptor comes from the fact that we assign states to the children of a tree before assigning states to the mother. Because of this, we don't know the state for the root node until the very last step! Before showing the next derivation, we note that if an acceptor is not defined for a transition $\delta(a, q_1 \cdots q_n)$, then the derivations terminates and the tree is not accepted/recognized/generated. We see this in the derivation of the invalid tree:

$$\begin{aligned}
\delta^*(S[a\ a\ b\ b]) &= \delta(S, \delta^*(a[\lambda])\ \delta^*(a[\lambda])\ \delta^*(b[\lambda])\ \delta^*(b[\lambda])) \\
&= \delta(S, \boldsymbol{\delta(a, \lambda)}\ \delta^*(a[\lambda])\ \delta^*(b[\lambda])\ \delta^*(b[\lambda])) \\
&= \delta(S, \boldsymbol{q_a}\ \delta^*(a[\lambda])\ \delta^*(b[\lambda])\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \boldsymbol{\delta(a, \lambda)}\ \delta^*(b[\lambda])\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ \boldsymbol{q_a}\ \delta^*(b[\lambda])\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ q_a\ \boldsymbol{\delta(b, \lambda)}\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ q_a\ \boldsymbol{q_b}\ \delta^*(b[\lambda])) \\
&= \delta(S, q_a\ q_a\ q_b\ \boldsymbol{\delta(b, \lambda)}) \\
&= \delta(S, q_a\ q_a\ q_b\ \boldsymbol{q_b}) \\
&= \texttt{undefined}
\end{aligned}$$

We are able to process all of the leaves of the tree, but the resulting string of states is undefined for our BDFTA. So, even though the two trees have the same "surface form" (i.e. $aabb$), *their underlying structure is different*. The BDFTA is able to differentiate between these two structures and inform us which one correctly belongs to our language. As linguists, being able to differentiate between different levels of structure is central to what we do and therefore shows the utility of different types of acceptors.

More on Tree Acceptors

What might the definition of a tree acceptor that accepts the tree S[S[a a] S[b b]] look like? How does it differ from the one we've seen?

2.4.3 String Transducers

So far we have looked at two types of acceptors. Recall that acceptors solve the *membership problem*: does a given string (or tree) belong to a set? In linguistics, we are sometimes interested in whether or not a given sequence/structure is a valid member of some set (e.g. - phonotactics), but we are also interested in how one sequence/structure is turned into another structure (e.g. - morphophonology, syntactic movement). This latter aspect of linguistics requires an understanding of the transformation problem for which we use *transducers* instead of acceptors.

We return now to strings and will begin our discussion of transducers by looking at a 1-way, deterministic, finite state string transducer (1DFST). Formally, a 1DFST is a mathematical object represented as a seven-tuple $\langle \Sigma, \Delta, Q, v_0, q_o, \delta, F \rangle$ where:

$$\begin{aligned} \Sigma &\text{ is the input alphabet/set of symbols} \\ \Delta &\text{ is the output alphabet/set of symbols} \\ Q &\text{ is a set of states} \\ v_0 \in \Delta^* &\text{ is the initial string} \\ q_0 \in Q &\text{ is a single initial state that is a member of } Q \\ \delta &\text{ is a transition function: } \Sigma \times Q \to \Delta^* \times Q \\ F &\text{ is a final function: } Q \to \Delta^* \end{aligned}$$

If transition $(\sigma, q, v, r) \in \delta$ it means there is a transition from state q to state r reading symbol σ on the input and writing string v on the output. We can refer to the first and second outputs of delta as δ_1 and δ_2 such that $\delta_1(\sigma, q) = v$ and $\delta_2(\sigma, q) = r$. These give us the output and the state transition.

With this in mind, we define a recursive process function $\pi : \Delta^* \times \Sigma^* \times Q \to \Delta^*$:

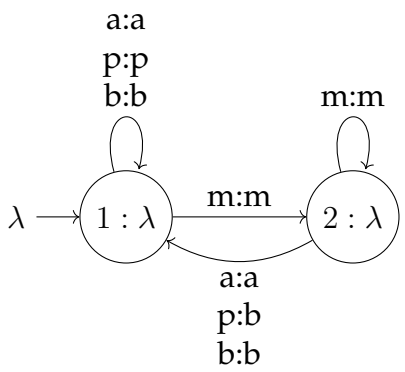
$$\begin{aligned} \pi(v, \lambda, q) &= v \cdot F(q) \\ \pi(v, \sigma w, q) &= \pi(v \cdot \delta_2(\sigma, q), w, \delta_1(\sigma, q)) \end{aligned}$$

Below we will look at a transducer influenced by the phonological process of post-nasal voicing found in some natural languages. This process turns voiceless segments immediately following nasal segments into their voiced counterparts.

We define the 1DFST for this process as follows:

$$\begin{aligned}
\Sigma &= \{a, b, p, m\} \\
\Delta &= \{a, b, p, m\} \\
Q &= \{1, 2\} \\
v_0 &= \lambda \\
q_0 &= 1 \\
\delta &= \{((a,1),(a,1)), ((a,2),(a,1)), ((b,1),(b,1)), ((b,2),(b,2)), \\
&\quad ((p,1),(p,1)), ((p,2),(b,1)), ((m,1),(m,2)), ((m,2),(m,2))\} \\
F &= \{(1,\lambda),(2,\lambda)\}
\end{aligned}$$

As was the case with string acceptors, string transducers are also hard to interpret just by looking at the formal definition. We can similarly use graphs to represent a specific transducer. The initial state once again has an ingoing arrow, but also includes the initial string. Generally, the formatting $x : y$ can be read as input x results in output y. The nodes now encode the states, but also the output of the final function F. The edges now include information about the transition function δ. The edge itself encodes the state transition, while the labels encode the string input/outputs.



Extending This Transducer

1. Extend the definition of this transducer to include other vowels, voiced and voiceless obstruents, and nasals of your choice.

2. Suppose we wanted to model the process of post-nasal *devoicing*. What would change about the definition of the machine?

3. Write a machine for **final devoicing** using $\rtimes, \ltimes$ as word boundary symbols.

4. **Bonus**: Note that this process is very local in the sense that each step is sensitive only to things that are immediately adjacent to it. Are there processes in natural language that you know of that do not seem to have this property? How would you account for them with this type of machinery?

Returning to our originally defined machine, let's take a look at the input form $mampa$ and see what it maps to:

$$\begin{aligned}
\pi(v_0, mampa, q_0) &= \pi(\lambda, mampa, 1) \\
&= \pi(\lambda \cdot \delta_1(m,1), ampa, \delta_2(m,1)) \\
&= \pi(\lambda \cdot m, ampa, 2) \\
&= \pi(\lambda \cdot m \cdot \delta_1(a,2), mpa, \delta_2(a,2)) \\
&= \pi(\lambda \cdot m \cdot a, mpa, 1) \\
&= \pi(\lambda \cdot m \cdot a \cdot \delta_1(m,1), pa, \delta_2(m,1)) \\
&= \pi(\lambda \cdot m \cdot a \cdot m, pa, 2)) \\
&= \pi(\lambda \cdot m \cdot a \cdot m \cdot \delta_1(p,2), a, \delta_2(p,2)) \\
&= \pi(\lambda \cdot m \cdot a \cdot m \cdot b, a, 1) \\
&= \pi(\lambda \cdot m \cdot a \cdot m \cdot b \cdot \delta_1(a,1), \lambda, \delta_2(a,1)) \\
&= \pi(\lambda \cdot m \cdot a \cdot m \cdot b \cdot a, \lambda, 1) \\
&= \lambda \cdot m \cdot a \cdot m \cdot b \cdot a \cdot F(1) \\
&= \lambda \cdot m \cdot a \cdot m \cdot b \cdot a \cdot \lambda \\
&= mamba
\end{aligned}$$

Compare this with input $mapa$, shown below:

$$\begin{aligned}
\pi(v_0, mapa, q0) &= \pi(\lambda, mapa, 1) \\
&= \pi(\lambda \cdot \delta_1(m,1), apa, \delta_2(m,1)) \\
&= \pi(\lambda \cdot m, apa, 2) \\
&= \pi(\lambda \cdot m \cdot \delta_1(a,2), pa, \delta_2(a,2)) \\
&= \pi(\lambda \cdot m \cdot a, pa, 1) \\
&= \pi(\lambda \cdot m \cdot a \cdot \delta_1(p,1), a, \delta_2(p,1)) \\
&= \pi(\lambda \cdot m \cdot a \cdot p, a, 1) \\
&= \pi(\lambda \cdot m \cdot a \cdot p \cdot \delta_1(a,1), \lambda, \delta_2(a,1)) \\
&= \pi(\lambda \cdot m \cdot a \cdot p \cdot a, \lambda, 1) \\
&= \lambda \cdot m \cdot a \cdot p \cdot a \cdot F(1) \\
&= \lambda \cdot m \cdot a \cdot p \cdot a \cdot \lambda \\
&= mapa
\end{aligned}$$

As you can see, the transducer processed the entire string, and returned the input with no changes. This is because there was no environment for the change to apply. So unlike acceptors, transducers never really fail in the same fashion. A transducer either completes a process if it can, or it doesn't.

String Acceptors as Transducers

We have defined acceptors and transducers as separate objects, but acceptors really are a subset of transducers. This also means the decision problem can be thought of as a special instance of the transformation problem. One way to think of this problem is to consider what you want the output alphabet to be. With that in mind, do the following:

1. Write a transducer that accepts the language $a^n b$.

2.4.4 Tree Transducers

Tree transducers are useful for many things, including modeling syntactic movement. Unfortunately due to time constraints, we will not be covering them in this course; however, references are listed in the box below for those interested.

Information on Tree Transducers

There is a rich literature in both theoretical computer science and computational linguistics on various types of tree automata and transducers. A definitive source for those interested is *Tree Automata, Techniques and Applications* by Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Loding, Sophie Tison, Marc Tommasi.

There is a free version of this book available at the following link:
https://jacquema.gitlabpages.inria.fr/files/tata.pdf

2.5 Closing Thoughts and Further Reading

Closing Thoughts

In this lecture, we continued to learn about some of the basic tools commonly employed by mathematical linguists. First, some elementary ideas in logic were introduced, which give us the machinery we need to discuss the strong relationship between logic and natural language across its many modules. Second, some ideas in automata theory such as string acceptors, string transducers, and tree acceptors were introduced, which helps lay some of the foundation for our discussion on the use of formal language theory for linguistic theorizing.

From this point, we will dive in and see how exactly these tools are used by mathematical linguists to understand the representations and computations that give rise to the human language faculty. Namely, we will use these tools to understand what inherent complexity constraints there are on natural language. We will be able to answer questions like:

- What types of patterns do we expect to find in natural language, and what types of patterns to we expect to be absent?
- Are some ways of mathematically representing these structures more computationally convenient? What are their similarities and differences?
- What can the mathematical properties we are exploring tell us about human (linguistic and non-linguistic) cognition?

While these methods and questions are applicable to *all* modules of linguistics, in these lectures, these questions will be tackled with our eyes predominantly on phonology and syntax.

Further Reading

For those interested in the relationship between subregularity and cognition, see Rogers and Pullum (2011), Jäger and Rogers (2012), Rogers et al. (2013), De Santo and Rawski (2020), and De Santo and Rawski (2022). For more on this specifically with respect to phonology, see Chandlee and Heinz (2017) and Heinz (2018), and more generally within linguistics, see Graf (2022b).

For recent discussion on what exactly the object of interest for theoretical syntax should be, see Hunter (2019), Stabler (2019), Graf (2022b, section 4).

The complexity of reduplication has long been of interest to computational linguists. Recently, the use of **2-Way Deterministic Finite State String Transducers** has shed new light on these questions. Interested readers should consult Dolatian and Heinz (2019) and Dolatian and Heinz (2020). Wang and Hunter (2023) is another recent exploration of the complexity of reduplication in terms of stringsets within the Chomsky-Schutzenberger Hierarchy.

If you are looking for a book-length treatment on Mathematical Linguistics, there are multiple options: Partee et al. (1993), Kracht (2003), Kornai (2007), and Keenan and Moss (2009).

Lesson 3

Formal Languages and Automata

3.1 Topics and Goals

In this lecture, we will dive a little deeper into formal language theory and talk about its place within theoretical linguistics. We will discuss some of the subregular languages and see how they can be useful in formally characterizing phonological phenomena. We will also discuss non-regular languages, and see how they can be useful in characterizing syntactic phenomena. By the end of this lecture, you should be able to describe (i) how studying computational complexity is useful for studying linguistic patterns (ii) how syntax and phonology differ in terms of their formal properties.

3.2 Formal Language Theory

Consider an alphabet Σ, which is simply a set of symbols. The set Σ^* consists of all possible strings over the alphabet Σ. We use the boundary symbols $\rtimes$ and $\ltimes$ to denote the left and right boundaries of a string, respectively. With these, we can also refer to the set of all strings enclosed by boundary markers: $\{\rtimes w \ltimes \mid w \in \Sigma^*\}$. The symbol λ is used to denote the empty string. We also refer to the set of all strings of at least length 1 as Σ^+.

A **substring** y of a string x (often denoted $y \trianglelefteq x$) is a string for which $x = uyv$ where $u, v \in \Sigma^*$. Note that either or both of u, v could also be the empty string.

A **subsequence** y of a string x (often denoted $y \sqsubseteq x$) is just like a substring, only the elements need not be linearly adjacent to one another. They can be at arbitrary distances so long as they generally precede one another.

Sub-structures

1. Which unique string is in Σ^* but not Σ^+?
2. What are some substrings of the string $establishment$?
3. What are some strings that the string $establishment$ is a substring of?
4. What are some subsequences of the string $establishment$?

A **language** L is a set of strings. Formally, we define this as $L \subseteq \Sigma^*$.

A **grammar** G is a formal device that generates or accepts a language L.

It is important to note that there are some languages which cannot be generated or accepted by any grammar; however, we are not interested in such cases. As linguists, we care about languages that can be generated with a grammar because language "makes infinite use of finite means" as Wilhelm von Humboldt has famously noted. In our case, the *finite* grammar allows us to describe *infinite* languages.

There are many different ways to formalize grammars that generate/accept languages – in this lecture, we will discuss Finite State Automata called **Finite State Acceptors** (FSA's).

These are machines that, for a given language L, accept strings[1] that are in L and reject strings that are not in L. Recall from Lecture 1 that this solves the **Membership Problem**.

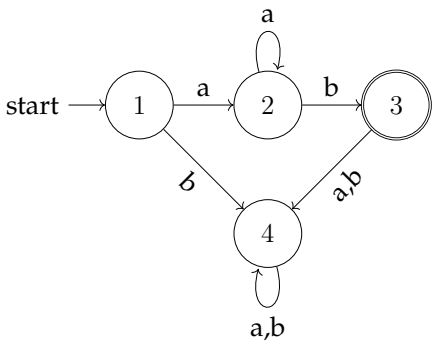
Now that we've seen some mathematical foundations, this terminology is very straightforward since a language is a set L of strings, and this machine determines whether or not a string w is a *member* of the language L. In phonology, this is very useful for reasoning about *phonotactic* constraints.

Generally, a Finite State Automaton is a 5-tuple $A := \langle \Sigma, Q, I, \Delta, F \rangle$, where:

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

The transition relation Δ has elements $\langle \sigma, p, q \rangle$ where $p, q \in Q$ are states and $\sigma \in \Sigma$ is an alphabet symbol. This triple signifies that in the state p if the symbol σ is read, the machine shall transition to state q.

An FSA is said to be **deterministic** iff there is only one initial state, and there are no two transitions starting from the same state with the same symbol ending in a different state.



$$\Sigma = \{a, b\}$$
$$Q = \{1, 2, 3, 4\}$$
$$q_0 = 1$$
$$\delta = \{((a,1),2), ((a,2),2), ((a,3),4), ((a,4),4), ((b,1),4), ((b,2),3), ((b,3),4), ((b,4),4)\}$$
$$F = \{3\}$$

[1] While formal language theory also deals with tree languages, in this course we will mostly deal with string langauges.

Looking at the graphical representation of our FSA's, this amounts to saying that there is no optionality in transitions from any given state. When you are in a state p and you see a symbol σ, you can only end up in one state q – you cannot optionally end up in some other state r. The FSA for the language $a^n b$ from Lecture 1 is shown above.

Determinism vs. Non-determinism

The uniqueness for each symbol-state pair makes the transition relation a function, and so the transitions in the deterministic case can be written as $\delta(\sigma, p) = q$.

1. Why is this the case?

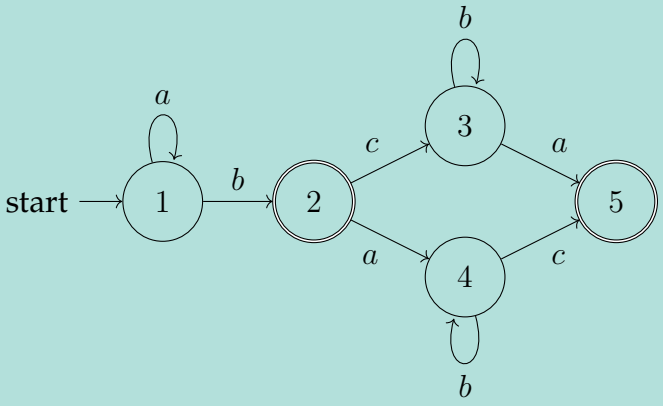
   Recall that a function is a special type of relation where each input has a unique output, it cannot have more than one output. Think about all of the elements in Δ – if for every $\langle \sigma, p, q \rangle$ transition there is not a $\langle \sigma, p, r \rangle$ transition where $r \in Q$ is another state, then there are unique transitions from each state for a given symbol, making Δ a function.

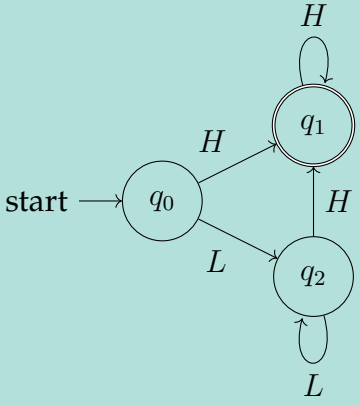
2. **Bonus**: Can you explain why there can be only one initial state for the machine to be deterministic?

Working with String Acceptors

Let's look at some string acceptors below and try to answer the following questions:

1. What languages do they accept?
2. Are they deterministic or non-deterministic?
3. What do the transition functions look like?
4. **Bonus**: The transitions for these acceptors are not defined using *total functions*, since there aren't transitions for every alphabet symbol at each state. What would they look like if we added sink states?





5. Draw a machine that accepts the language $(\text{CV})^{n \geq 1}$.
6. **Bonus**: How would you change the machine above to accept the language $(\text{CVC})^{n \geq 1}$?

Making an Acceptor Non-deterministic

The acceptor above only has two alphabet symbols, a and b. It is deterministic since there is only one initial state, and there is a unique transition for each symbol for every state.

1. Suppose we added a transition $((b, 1), 3)$ – what would change about this acceptor? Would it still be deterministic? What language would it accept?

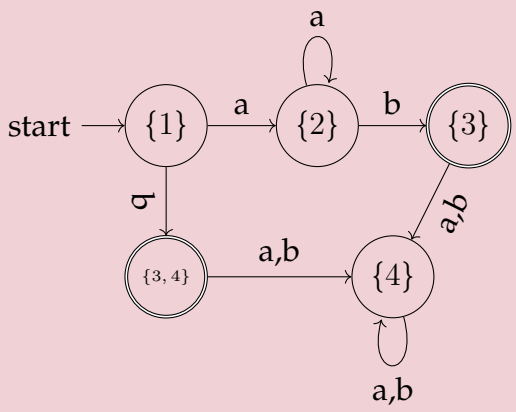
Determinizing a Non-deterministic String Acceptor

Any non-deterministic finite state string acceptor is equivalent to a deterministic finite-state string acceptor. In the NFSA, we pursue multiple paths of single states. In the equivalent DFSA, we pursue a single path of multiple states.

Suppose we have an NFSA $N = \langle \Sigma_N, Q_N, I_N, \Delta_N, F_N \rangle$, we can construct the equivalent DFSA $D = \langle \Sigma_D, Q_D, q_{0_D}, \delta_d, F_D \rangle$ where

$$
\begin{aligned}
\Sigma_D &= \Sigma_N \\
Q_D &= \mathcal{P}(Q_N) \\
q_{0_D} \in Q_D &= q \in Q_D \mid q = I_N \\
\delta_D &= ((\sigma \in \Sigma_D, q_D \in Q_D), q') \mid q' := \{q'' \mid \langle \sigma, q, q'' \rangle \in \Delta_N, q \in q_D\} \\
F_D \subseteq Q_D &= q \in Q_D \mid q \cap F_N \neq \emptyset
\end{aligned}
$$

In the example above, we added a transition to make the deterministic machine for $a^n b$ non-deterministic. This resulted in a new language: $a^n b \cup \{b\}$. Below is the determinized version of the machine that accepts this language. Note, the definition above would result in many more states, but those states are never reached, so they have been removed below. This formulation really highlights how this language is the union of our first language and one extra string.



$$
\begin{aligned}
\Sigma_D &= \{a, b\} \\
Q_D &= \{\{1\}, \{2\}, \{3\}, \{4\}, \{3, 4\}\} \\
q_{0_D} &= \{1\} \\
\delta_D &= \{((a, \{1\}), \{2\}), ((a, \{2\}), \{2\}), \\
&\quad ((a, \{3\}), \{4\}), ((a, \{4\}), \{4\}), \\
&\quad ((b, \{1\}), \{3, 4\}), ((b, \{2\}), \{3\}), \\
&\quad ((b, \{3\}), \{4\}), ((b, \{4\}), \{4\}), \\
&\quad ((a, \{3, 4\}), \{4\}), ((b, \{3, 4\}), \{4\})\} \\
F_D &= \{\{3\}, \{3, 4\}\}
\end{aligned}
$$

3.3 The Subregular Languages and Phonology

Now that we've seen some examples of formal languages and automata that represent them, this section will discuss how different types of formal languages have different corresponding complexities. We will dive into how the computational machinery required to generate distinct patterns can vary, and see some more linguistically motivated examples.

3.3.1 The Chomsky-Schützenberger Hierarchy

The Chomsky-Schützenberger Hierarchy is a hierarchy of formal language classes.[2,3] In the figure shown below, the darker colors represent the more formally complex classes. So in this hierarchy, the recursively enumerable languages are the most complex and the regular are the least complex; however, we will see that it is useful (particularly if you are a phonologist) to probe further down within the regular region.

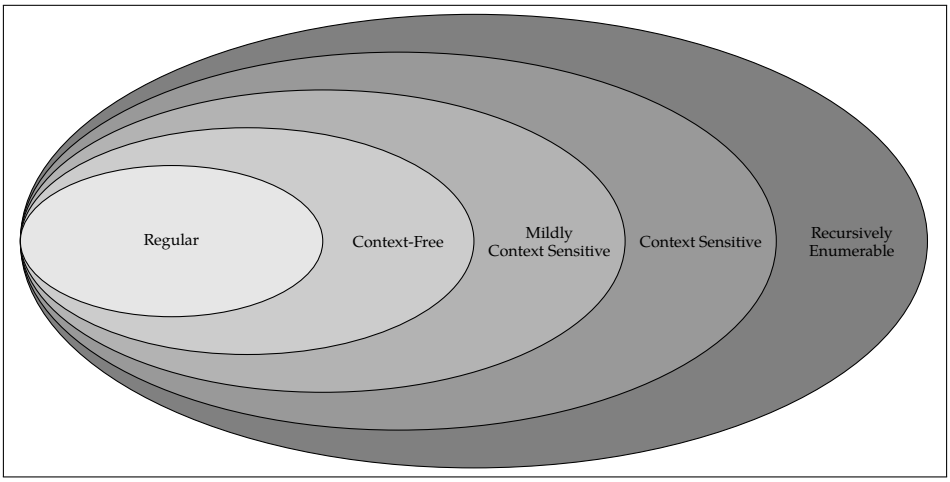


Figure 3.1: Chomsky-Schützenberger Hierarchy

So what makes a language regular or context sensitive?

The mathematical machinery used to express these differences spans a few different areas of mathematics. One can use automata theory, abstract algebra, mathematical logic, etc. This lecture will focus mainly on automata theory, while later lectures will focus on mathematical logic, specifically finite model theory.

Viewing things from an automata theoretic lens, each of these language classes has corresponding type of machine that generates or recognizes languages from that class. Observe the table below:

[2]This is not the only hierarchy of interest in formal language theory. This hierarchy represents stringsets, while there are other hierarchies that represent sets of pairs of (input,output) mappings for transformations. Such a hierarchy draws computational distinctions between classes of mappings as opposed to classes of languages.

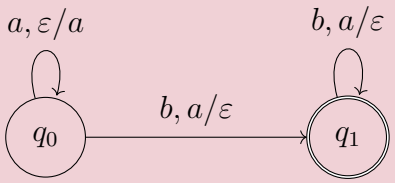
[3]By *language classes* here, all that is really meant is *sets of languages*. For example, the class of regular languages is just the set of all languages meeting the property of being regular.

Langage Class	**Machine**
Regular	Finite State Automaton
Context-Free	Pushdown Automaton
Context Sensitive	Linear Bounded Automaton
Recursively Enumerable	Turing Machine or the λ-calculus

Pushdown Automata and Context-Free Languages

A Pushdown Automaton is a type of automaton that makes use of what is called a *stack*, which allows the automaton to keep a memory of arbitrarily many steps.

A stack is a data structure which is a collection of elements and two operations `push` (which adds an element) and `pop` (which removes the most recently added). A Pushdown Automaton for the language $a^n b^n$ is shown below:



The intuition is that for a given string, the stack keeps track of exactly how many a's the automaton has seen, and once it sees a b, it unloads all of those a's, and the string will only be accepted if there are exactly that many b's remaining in the string.

Membership and Transformation

As we've seen, both *languages* and *transformations* can be used to formally characterize natural linguistic patterns, but let's look more closely at a linguistic example.

One can view vowel harmony as either (i) a well-formedness constraint on words, where ill-formed words violate harmony and well-formed words do not, or (ii) a process which turns UR's into SR's that satisfy harmony.

Well-formedness Constraint:	**Transformation:**
Good forms: ev-ler, ev-ler-e, at-lar, at-lar-a, . . .	ev-lAr $\mapsto$ ev-ler ev-lAr-A $\mapsto$ ev-ler-e at-lAr $\mapsto$ at-lar
Bad forms: *ev-lar, *ev-lar-a, *at-ler, *at-ler-e, . . .	at-lAr-A $\mapsto$ at-lar-a

On the left, vowel harmony is a restriction on allowed/disallowed words within a language (Membership Problem). On the right, vowel harmony is an input-output mapping where a UR is mapped to an SR (Transformation Problem).

3.3.2 Subregularity and Phonology

Consider the standard SPE Rewrite Rules of the form $A \rightarrow B / C\_\_D$, which is interpreted as 'An A is rewritten as a B in between a C and a D', where A, B, C, D are all regular stringsets.

A well-established result of computational phonology is that these rewrite rules, when applied in a unidirectional, simultaneous fashion, represent the regular relations. With this information, one could posit that the stringsets and mappings that are able to describe phonological patterns are properly regular. This correctly rules out pathologies like the Midpoint Pathology (a type of mapping which targets the middle of the input string) and Majority Rules Harmony (a type of mapping in which the entire output string harmonizes with whichever feature occurs more in the input string), which are both non-regular.

However, regularity also seems too strong for phonology since it has the capability to produce different types of unattested patterns. A logically possible regular language could enforce that the first and last segments of every word agree in roundness, which is called first-last harmony. Another possible regular language could enforce that words can only contain an even amount of high vowels and if they contain an odd number of high vowels they are illicit. Patterns like these are not found in natural language.

Any decent scientific theory should be sufficiently restrictive and expressive. Of course we want a theory of phonology that has the ability to straightforwardly capture the wide landscape of attested phenomena, but we don't want a theory so strong that it easily captures unattested phenomena. What sorts of advantages do the mathematical tools described here have for building such a theory?

There is beauty in using abstraction to study the computational properties of these patterns themselves because it allows us to quantify unambiguously what it means for one pattern to be "more complex" than another. This can ultimately shed light on why we have a tendency to see patterns of *this* type and not patterns of *that* type. Thus, our understanding of the relationship between linguistic typology and learnability ends up falling out quite naturally looking at these patterns from a computational perspective.

This leads us to the **Subregular Hypothesis**, which posits that all patterns in natural language phonology can be described by stringsets and mappings that are Subregular. Under this hypothesis, phonology does not make use of the full computational power of regularity, it can be naturally expressed with much simpler types of stringsets and mappings. Below, the Subregualar Hierarchy is shown – note that this entire hierarchy is situated properly within the regular region of the Chomsky-Schützenberger Hierarchy, meaning that these language classes are less formally complex than the regular languages.

A glossary of all of the terms in this hierarchy is given below, and the remainder of this section will expand upon some of these classes through some phonologically motivated examples. In a later lecture when we discuss finite model theory, we will go into more detail about how the levels of logical power and relations used give rise to different language classes.

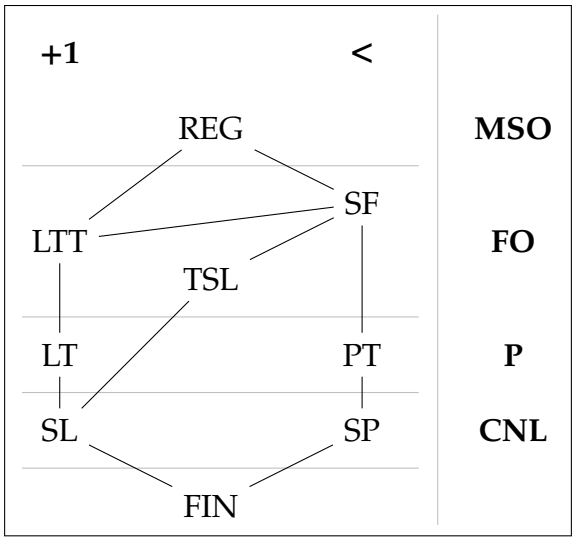


Figure 3.2: The Subregular Hierarchy

Relations:

+1	Immediate Successor Relation
<	General Precedence Relation

Logics:

MSO	Monadic Second Order Logic
FO	First Order Logic
P	Propositional Logic
CNL	Conjunction of Negative Literals

Language Classes:

FIN	Finite
SL	Strictly Local
SP	Strictly Piecewise
TSL	Tier-Based Strictly Local
LT	Locally Testable
PT	Piecewise Testable
LTT	Locally Threshold Testable
SF	Star-Free
REG	Regular

3.3.3 Finite Languages

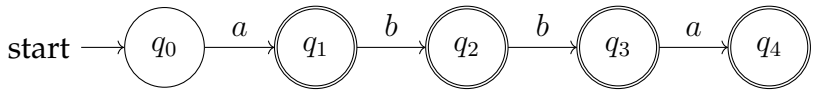
Simply, the **Finite** languages are those that contain only finitely many strings.

Or more formally, $FIN = \{L \mid \text{for some } n \in \mathbb{N},\ |L| = n\}$.

There is an automata characterization of the finite languages as well: deterministic **acyclic** finite-state acceptor (DAFSA; Daciuk et al., 2000). We already know what makes a finite state machine *deterministic*, but what does it mean to be *acylic*? It simply means that once you exit from a state in the machine, there is no way to return to that state.[4] More formally, a DAFSA is a 1DFSA where the transition function δ is acylic – there is no string w and state q such that $\delta^*(w, q) = q$. In other words, there is no path in the machine that starts and ends in the same state. These types of structures have been used in computational linguistics and natural language processing to efficiently represent **dictionaries** (which by their very nature contain only a finite set of strings).

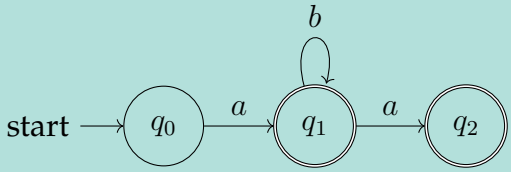
[4]It is impossible to have a sink state/total function and be acyclic. One way around this is to define the transition and process functions to return `false` when a transition is undefined. We ignore this for now.

As an example of a finite language, consider the language containing only the four strings $L = \{a, ab, abb, abba\}$. An FSA that recognizes this language is shown below:



Finite and Infinite Languages

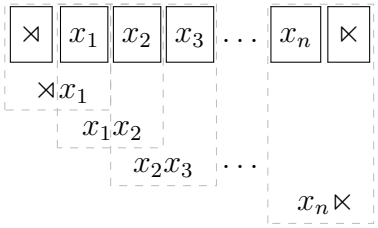
For the acceptor above, suppose instead of the b transitions from q_1 to q_2 and q_2 to q_3, we had a loop at q_1 with a b. This change is reflected in the FSA below.



1. Does this FSA recognize all of the forms in L?
2. Does this FSA recognize the same language?
3. If not, is the language that this FSA recognizes finite? Why or why not?

3.3.4 Strictly Local Languages

The **Strictly Local** languages are those that can be recognized by scanning a fixed k-length window of a given string and checking whether the k-length substring in each window is allowed in the language. This is schematized below for a string $x = x_1 \dots x_n$ and $k = 2$:



We formalize the k-length windows as the ***k*-factors** of a string s, where k-factor is defined as follows:

$$\texttt{factor}_k(s) = \begin{cases} \{u \mid u \trianglelefteq s, |u| = k\} & \text{if } k \leq |s| \\ \{s\} & \text{otherwise} \end{cases}$$

This definition is then further extended such that the domain is not a single string but instead a set of strings S:

$$\texttt{factor}_k(S) = \bigcup_{s \in S} \texttt{factor}_k(s)$$

We can now define an SL_k grammar $\mathcal{G} \subseteq \texttt{factor}_k(\{\rtimes w \ltimes \mid w \in \Sigma^*\})$ as a set of illicit k-factors. A word w satisfies an SL_k grammar $\mathcal{G}$ if and only if $\texttt{factor}_k(w) \cap \mathcal{G} = \emptyset$. If the intersection of the k-factors and the grammar is empty this means there are no illicit substrings in the word!

Finally, we can define the language $\mathcal{L}$ recognized by a grammar $\mathcal{G}$ as:

$$\mathcal{L}(\mathcal{G}) = \{w \mid \texttt{factor}_k(w) \cap \mathcal{G} = \emptyset\}$$

This just says the strings recognized by a grammar are those that contain none of the illicit substrings found in our grammar. We can therefore talk about the set of all strictly local languages SL_k as all the sets of strings S which are the language $\mathcal{L}$ of some strictly local grammar $\mathcal{G}$ for some $k \geq 1$:

$$SL_k = \{S \mid \mathcal{G} \subseteq \texttt{factor}_k(\{\rtimes w \ltimes \mid w \in \Sigma^*\}), \mathcal{L}(\mathcal{G}) = S\}$$

Let's now consider an example from natural language phonology. Intervocalic voicing is a phenomenon found in various natural languages. Recall from our discussion earlier that many phenomena can be viewed as either a transformation or a well-formedness condition. We could think about this phenomena as the result of a rule such as $[-\ \texttt{son}] \rightarrow [+\ \texttt{voi}]/[+\ \texttt{syl}]\_\_[+\ \texttt{syl}]$ or OT constraint ranking *VTV $\ggg$ IDENT-VOICE, but since we are talking about stringsets, we will view intervocalic voicing as all the sets of strings that do not contain a voiceless consonant in between two vowels.

We will simplify our analysis using the alphabet $\Sigma = \{V, T, D\}$ where V stands for all vowels, T stands for all voiceless consonants, and D stands for all voiced consonants. Consider the language $\mathcal{L}$ which accepts all and only strings in Σ^* which do not contain the substring VTV. This requires a window of size $k = 3$. All of the possible length 3 substrings in $\{\rtimes\Sigma^*\ltimes\}$ are shown in Table 3.1.

$\rtimes VV$	VVV	TVV	DVV	$VV\ltimes$
$\rtimes VT$	VVT	TVT	DVT	$VT\ltimes$
$\rtimes VD$	VVD	TVD	DVD	$VD\ltimes$
$\rtimes TV$	VTV	TTV	DVT	$TV\ltimes$
$\rtimes TT$	VTT	TTT	DTT	$TT\ltimes$
$\rtimes TD$	VTD	TTD	DTD	$TD\ltimes$
$\rtimes DV$	VDV	TDV	DDV	$DV\ltimes$
$\rtimes DT$	VDT	TDT	DDT	$DT\ltimes$
$\rtimes DD$	VDD	TDD	DDD	$DD\ltimes$

Table 3.1: All possible substrings of length 3 in $\{\rtimes\Sigma^*\ltimes\}$.

since we know that substrings like VTV and $DVTVD$ are disallowed since a voiceless segment occurs between two vowels, but substrings like VDV and $TVDVT$ are allowed since voiced segments occur between two vowels.

Intervocalic Voicing Acceptor

There is a relationship between the k-window size and the number of states necessary to represent a given pattern.

1. If a pattern is n-local, what is the minimum number of states required in the corresponding finite state machine?
2. Write an acceptor for the intervocalic voicing language.

3.3.5 Strictly Piecewise Languages

The **Strictly Piecewise** languages are similar to the Strictly Local languages, but differ in that the factors are over *subsequences* as opposed to *substrings*. Therefore, we can write a definition for k-subsequences and the Strictly Piecewise grammars and languages fall directly out of this new definition.

$$\texttt{subseq}_k(s) = \begin{cases} \{u \mid u \sqsubseteq s, |u| = k\} & \text{if } k \leq |s| \\ \{s\} & \text{otherwise} \end{cases}$$

Recall that subsequences don't care about intervening material within the string. For example, we can talk about substring constraints like the string contains an a *directly* followed by a b. On the other hand, we can also say something like the string contains an a that has a b somewhere in the string after it. This latter description would be a subsequence constraint. The two machines in Figure 3.3 below highlight this difference.

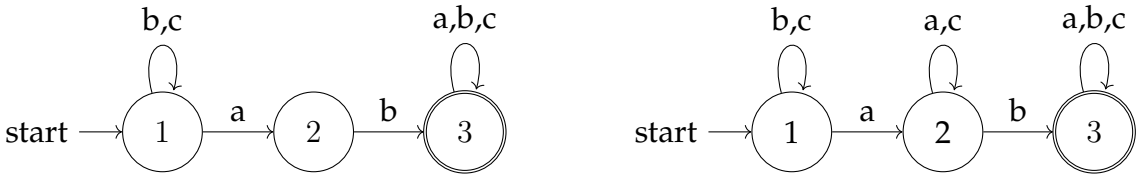


Figure 3.3: Left: machine recognizing "there must be a b directly following an a somewhere in the string"; Right: machine recognizing "there must a b somewhere in the string following an a.

The machines are quite similar! The crucial difference is the loop in state 2. This highlights the fact that b can be arbitrarily far away from a in the string, it just has to appear at some point. If you have studied phonology, you may be thinking of certain patterns that can seemingly have arbitrarily long dependencies. We can use the idea of subsequences to study and analyze this types of patterns, which we will do below.

Subsequences

Samala is a language native to the California area. It has a pattern of *symmetrical consonant harmony* where words cannot contain both [s] and [ʃ]. Words like [posokoso] and [poʃokoʃo] are well formed, while words like *[posokoʃo] and *[poʃokoso] are ill-formed.

1. Write an acceptor that will recognize "Samala" using the alphabet $\Sigma = \{\text{s}, \text{ʃ}, C, V\}$.

Sarcee is an Athapaskan language with an *asymmetrical consonant harmony* pattern. In Sarcee, [s] may follow [ʃ], but [ʃ] may not follow [s]. So words like [posokoso] and [poʃokoʃo], and [poʃokoso] are all well-formed, but words like *[posokoʃo] are ill-formed.

2. Write an acceptor that will recognize "Sarcee" using the alphabet $\Sigma = \{\text{s}, \text{ʃ}, C, V\}$.

3.3.6 Tier-based Strictly Local Languages

The last class of stringsets we will discuss is the **Tier-based Strictly Local** languages (TSL). TSL imported the idea of tiers from autosegmental phonology into the formal language theory domain. The only difference between TSL and SL languages is the fact that TSL only cares about some relevant subset of alphabet symbols, which are projected on a given *tier*.

We can define the TSL languages formally in the following way:

A language L is TSL-k if there exists a $k \in \mathbb{N}$ and a tier $T \subseteq \Sigma$ such that all strings in L are SL-k when restricted to only the symbols in T.

For example, we might define an *erasing* function E_T that takes a string as its input and "erases" all of the non-tier symbols: $E_T(\sigma_1 \cdots \sigma_n) = u_1 \cdots u_n$ where $u_i = \sigma_i$ iff $\sigma \in T$, otherwise $u_i = \lambda$.

$$E_T(S) = \overset{\boxed{}}{F \qquad\qquad\qquad\qquad\qquad\qquad F}$$
$$S = \quad F\ C\ C\ C\ C\ C\ C\ C\ C\ C\ C\ F$$

Figure 3.4: Tier Construction

There are many types of vowel harmony across the world's languages, but lets look at an abstract pattern of front-back harmony over the alphabet $\Sigma = \{F, B, C\}$ where F is a front vowel, B is a back vowel, and C is a consonant. Valid strings in the language are ones that contain no mismatch of F's and B's. So all vowels have to be "front" or all vowels have to be "back" (Notice in our description of this pattern, the consonants don't matter at all!) So we can analyze this by saying there is a vowel tier $T = \{F, B\}$.

What we really care about is the tier string of vowels. This language is TSL-2 because we only need to check *two adjacent symbols on the tier*. But if we were to look at the actual strings (before we erased the consonants) there's no guarantee that such a k would exist. For example, $FCCCCCCCCCCF$ has many consonants in a row, but if we were to apply the erasing function, the tier string is just FF, and so this long distance pattern is very local in a *relative* sense.

3.4 The Non-regular Languages and Syntax

The previous section showed why the regular languages/mappings are too strong for phonology. How do they fare for syntax?

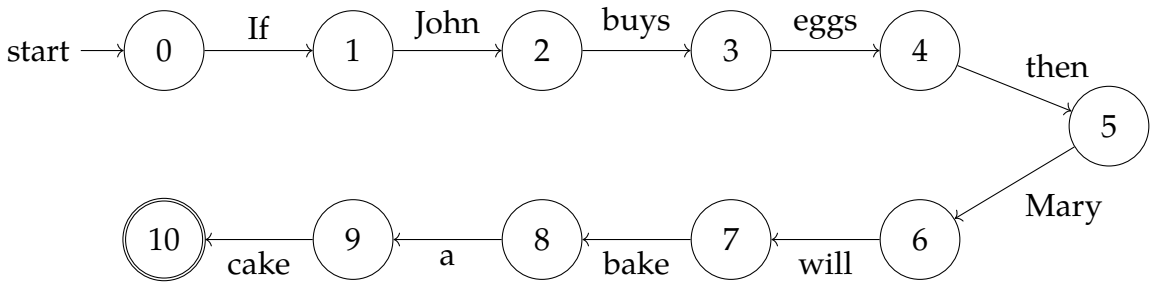
Recall that the Regular languages are those that can be recognized by an FSA. Thus, if syntax were Regular, that would mean that we could capture syntax with an FSA.

Recall our old friend $a^n b^n$, which we said is Context-Free and thus requires a Pushdown Automaton to recognize it. This is due to the fact that in order to know how many b's occur in the string, you must have kept track of exactly how many a's were encountered. This crucially cannot be done with an FSA because there are only *finitely many states*.

Consider the sentences below.

1. John buys eggs.
2. Mary will bake a cake.
3. If John buys eggs, then Mary will bake a cake.
4. If John buys either eggs or flour, then Mary will bake a cake.
5. If John buys either eggs with neither scratches nor cracks or flour, then Mary will bake a cake.

If we attempt to construct an FSA for sentence 3, it will look something like this:



In sentence 4, we see that within the dependency between 'if' and 'then', we can have another dependency between 'either' and 'or'. Adjusting our FSA would mean adding more states and transitions between states 1 and 4 for the added material between the dependency.

In sentence 5, there is another dependency between 'neither' and 'nor' within the 'either'-'or' dependency. This results in adding more states and transitions between the states we already needed to add to account for sentence 4.

We have all spoken enough human language to know that this can go on *ad infinitum*. Namely, every extra embedding added requires more states and more transitions between them and this can be done indefinitely. This requires more expressive power than an FSA has to offer. This is the same reason that there is no FSA that can recognize $a^n b^n$ – FSA's have a bounded memory.

Rather than trying to draw up a Pushdown Automaton to handle fragments of English, we can use a convenient system of rewrite rules to represent Context-Free languages.

A **Rewrite Grammar** or **Phrase Structure Grammar** is a tuple $\langle T, N, S, R \rangle$ where:

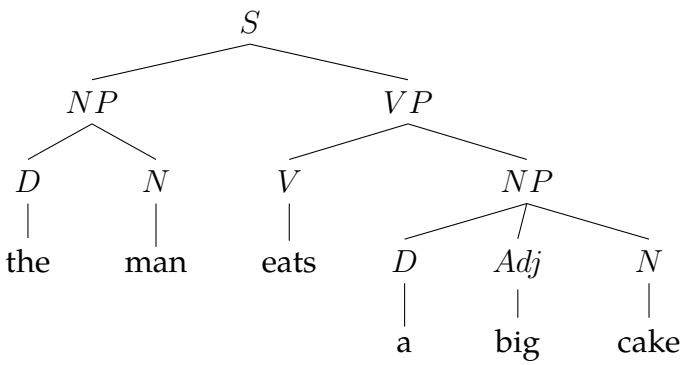
- T is a set of terminal symbols
- N is a set of non-terminal symbols
- S is a non-terminal starting symbol
- R is a set of rewrite rules where $A \to B$ and $A, B \in (N \cup T)^*$

Specifically, a **Context-Free Grammar** is a Phrase Structure Grammar such that for every rewrite rule $A \to B \in R$, it is the case that $A \in N$ is a single non-terminal and $B \in (N \cup T)^*$ is a sequence of non-terminals or a single terminal.

Observe the CFG below for a small fragment of English where parentheses indicate optionality of a given non-terminal and the Kleene star * continues to indicate iterativity:

- $S \to NP \ \ VP$
- $NP \to D \ (Adj)^* \ N$
- $D \to$ the | a
- $N \to$ man | cake
- $VP \to V \ (DP)$
- $V \to$ eats
- $Adj \to$ big | small

The derivation for an example sentence 'The man eats a big cake' using this CFG is shown below:



Context-Free Grammars

1. What are some other sentences that the Context-Free Grammar defined above can generate?

2. In the CFG defined above, note how there is only one verb *eats* which can be either transitive (as in *eat the cake*) and intransitive (as in *the man eats*). This is encoded in our CFG by the VP rewrite rule having an optional DP.

   Suppose we added to our list of non-terminals verbs that are either only transitive (*throws, brings, etc.*) or only intransitive (*dies, jumps, etc.*). How would this change our set of rewrite rules?

3. Let's say we wanted to add PP adjunction to account for sentences like *The man with purple hair*. How would this change our set of rewrite rules?

4. **Bonus**: Suppose we changed the form of our rewrite rules from $A \to B$ where both $A, B \in (N \cup T)^*$ to $A \to aB$ or $A \to a$ where $A, B \in N$ and $a \in T$. What would this change about the nature of our grammar? Does it become *more* or *less* expressive?

Not only has the view of syntax evolved considerably since the introduction of CFG's, but there are also some syntactic phenomena that appear require even more expressive power.[5] Cross-serial dependencies in Swiss German have been shown to require more expressive power than Context-Free has to offer. This has led to the introduction and development of formalisms such as **Tree Adjoining Grammars** (TAG's) and **Minimalist Grammars** (MG's), which hover in this mildly-context sensitive region.

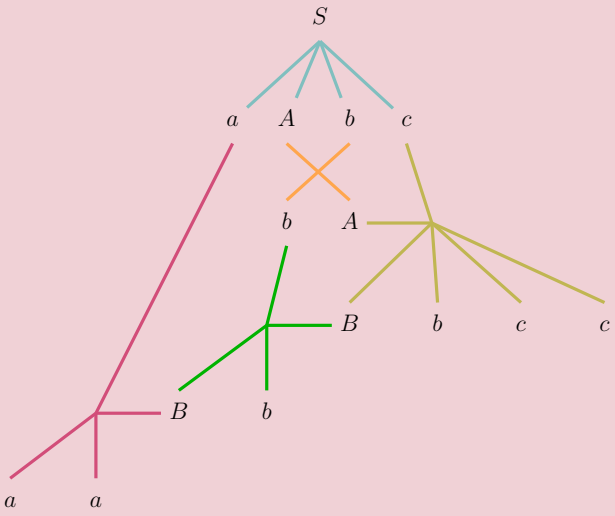
[5]The discussion here revolves around assuming that trees are string generators and under this assumption the argumentation holds, but see Graf (2022b) for a very interesting discussion on how rethinking certain representational assumptions, syntax can appear to be regular.

Type n grammars

The Chomsky hierarchy was originally formulated in terms of restrictions on rewrite grammars. Type 0 grammars are completely unrestricted and correspond to the *Recursively Enumerable* languages. Type 1 grammars correspond to the *Context-Sensitive* languages which require the right side of the rule to be greater than or equal to in size of the left side of the rule. Type 2 grammars correspond to the *Context-Free* as described above. Type 3 grammars correspond to the *Regular* languages and restrict the rewrite rules to be of the form $N \to T \mid TN$. Example are shown below.

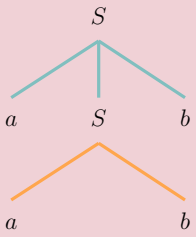
Context-Sensitive: $a^n b^n c^n$

S	$\to$	$abc \mid aAbc$
Ab	$\to$	bA
Ac	$\to$	$Bbcc$
bB	$\to$	Bb
aB	$\to$	$aa \mid aaA$



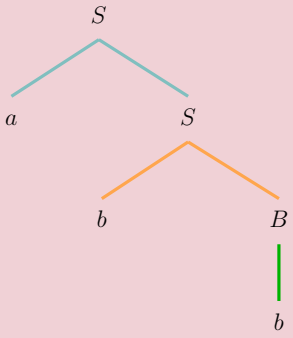
Context-Free: $a^n b^n$

S	$\to$	aSb
S	$\to$	ab



Regular: $a^n b^m$

S	$\to$	aS
S	$\to$	bB
S	$\to$	$a \mid b$
B	$\to$	$b \mid bB$



Minimalist Grammars

Minimalist Grammars (MG's) are a mathematical formalization of Chomsky's Minimalism, which have become very widely used among computational syntacticians. For reasons of time and space, we will not be covering MG's in this course.

While this list is not exhaustive, the interested reader can get more acquainted with them through the following papers: Stabler (1997), Michaelis (2001),Kobele (2006), Kobele et al. (2007), Salvati (2011), Graf (2012).

3.5 Closing Thoughts and Further Reading

Closing Thoughts

In this lecture, we used some of the machinery introduced in lectures 1 and 2 to investigate some linguistic phenomena. We began to look at some language classes within the subregular hierarchy from an automata-theoretic perspective. Some examples of finite state acceptors were shown for different language classes and how they can describe things like tonal generalizations, different types of consonant and harmony, intervocalic voicing, etc. It was also discussed how certain formal devices fail to capture syntactic generalizations and what types of expressive power is required.

From this point, we will use logic, one of our remaining mathematical tools to see how we can understand generalizations in phonology and syntax from a new perspective. While the tools that we will use to investigate will be different, we find that there is remarkable similarity in how unified all of the underlying properties are.

Further Reading

For some texts covering the topics of formal language theory and automata theory more generally, see Hopcroft and Ullman (1969); Moll et al. (1988); Sipser (1996).

TSL grammars were first introduced by Heinz et al. (2011). For some extensions of this class, see Graf and Mayer (2018) and De Santo and Graf (2019). To see how tiers can be used in defining transformations, see Andersson et al. (2019) and Burness et al. (2021).

While we looked at the relationship between well-formedness and locality mostly with respect to strings, to see how it can be extended to non-linear structures for studying tonal phenomena, see Jardine (2017, 2019).

We briefly mentioned the dual nature of phonological patterns as both transformations and well-formedness conditions. While not discussed here, transformations require *transducers* instead of *acceptors*. Because of this, the purely automata perspective makes it hard to compare the complexity across the two types of machines. If we instead take an algebraic approach, a unified analysis is possible. Lambert (2022) does so, and also extends the subregular hierarchy into what he calls a "subregular spiral".

Lesson 4

Model Theory and Logic

4.1 Topics and Goals

In the previous lecture, we looked at formal language theory from the perspective of automata. In this lecture, we will take an alternative approach that uses model theory, which is related to mathematical logic, as a way of characterizing language classes. First, we will discuss what a "model" is in terms of model theory and how this idea can be used to describe various types of linguistic structure. Our focus will primarily be on strings, but will show that the model-theoretic approach can be extended to *structures* more broadly defined.

After introducing models, we will discuss first-order logic and show the interplay between representation and logical language for describing different subregular classes. Here, we will focus primarily on the strictly local, strictly piecewise, and tier-based strictly local languages.

To conclude, we will discuss the idea of interpretations in logic and model theory, and how these can be used to describe phonological (and other types of linguistic) transformations. Additionally, we will discuss how interpretations can be used to turn one type of structure into another type of structure which is useful when thinking across different linguistic modules or doing theory comparison within modules.

By the end of the lecture, you should be able to (i) explain how model theory is useful for studying natural language phenomena, (ii) define a model signature and build a structure, (iii) explain how logical constraints relate to formal language theory, and (iv) interpret one structure in terms of another structure.

4.2 Models

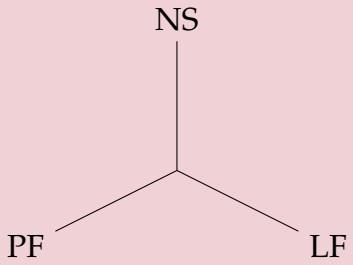
Model theory is a subfield of mathematical logic interested in whether or not a specified structure (model) satisfies a given constraint. Its history can be traced back to work in philosophy and logic in the 1800's, but it wasn't until the 1950's when Alfred Tarski claimed

that there was a new field of inquiry: the theory of models (Tarski, 1954).[1] In this sense, it is still a relatively young field.

The use of model theory for studying natural language has an even younger history. Richard Montague gave model theory a home within the study of natural language semantics (Montague, 1974), and it continues to be a fruitful area of inquiry to this day. Decades later, Jim Rogers used model theory to analyze the complexity of natural language syntax as it is conceived within a GB framework (Rogers, 1998), showing that semantics was not the only subfield of linguistic theory that could benefit from using model theory for formal analysis. Potts and Pullum (2002) brought model theory into the domain of phonology, thus showing that structure, form, and meaning (i.e. - The "inverted T/Y model" of the grammar) can all be fruitfully studied with model theory.

Inverted T/Y model

It is assumed among many generative linguists that the architecture of natural language follows an inverted-T/Y model where **Narrow Syntax** (NS) computes the form underlying an utterance and it splits off into **Phonetic Form** (PF) to be pronounced by Articulatory-Perceptual system and **Logical Form** (LF) to be interpreted by the Conceptual-Intentional system.



Roughly, the branch from NS to PF consists of the morphological, phonological, and phonetic modules, whereas the branch from NS to LF consists of the semantic module.

Later, when we discuss transformations between linguistic structures using some of our tools from mathematical logic, we will see the robustness these tools have when assuming this modular, feed-forward architecture underlying natural language.

Before we formally define what a model is, let's do a short exercise to think about what parts of a given structure we care about. What is the difference between the strings in the following three pairs?

aaa & aaaa
darp & drap
darp & tarp

[1] For a "short" history of model theory, see Hodges (2018)

In the first pair, there is variation between the *number of elements* in the string. In the second pair, there is variation between the *order of elements* in the string. In the third pair, there is variation between the *properties of elements* in the string.

The term **model** refers to a structure in some model signature. A **model signature** is a collection of functions, relations, and constants that are used to describe structures.[2] Model signatures are often denoted with Σ.[3] With this in mind, a Σ-structure A (equivalently a *model* A) contains a set called the **domain**, as well as denotations for each symbol in Σ. Below is how we denote functions, relations, and constants:

A denotation for a constant is a single element of the domain of A.
A denotation for a function symbol of arity k is a k-ary function on the domain of the A.
A denotation for a relation symbol of arity k is a k-ary relation on the domain of the A.

Different model signatures are useful for describing different types of linguistic phenomena. We will begin by looking at strings over the relational model signature $\langle \lhd, \{R_\sigma \mid \sigma \in \Sigma\}\rangle$. This signature defines strings using the "successor" relation ($\lhd$). It is standard to define the domain D as a subset of the natural numbers $\mathbb{N}$. Given this assumption, we can define the successor function as follows.

$$\lhd \stackrel{\text{def}}{=} (i, i+1) \in D \times D$$

The other relations in our signature are R_σ where these are unary relations for ever symbol in our alphabet. We can define strings using this model signature by labeling the *properties of elements* of the domain with our R_σ's and *order the elements* of the domain with $\lhd$. With the domain itself we get the *number of elements*. Think back to our thought exercise from above. These are exactly the parts of a string we are interested in.

Suppose we wanted to define string aaa. What would our model look like? Let's have our alphabet of symbols be $\Sigma = \{a, d, t, r, p\}$. Figure 4.1 shows what it would look like both formally (left) as well as graphically (right). Graph representations allow us to easily visualize strings and other structures we are interested in, but remember that the mathematical object itself is the domain and denotations.

$$\begin{aligned}\langle D &= \{0,1,2\},\\ \lhd &= \{(0,1),(1,2)\},\\ a &= \{0,1,2\},\\ d &= \{\}, t = \{\}, r = \{\}, p = \{\}\rangle\end{aligned}$$

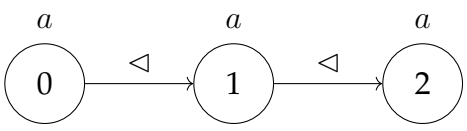


Figure 4.1: Σ-structure for string aaa.

[2] A constant is a symbol in our logic that is always given the same semantic interpretation.
[3] Sorry if this is confusing due to Σ also being used as the symbol for the 'alphabet' as well.

Defining Strings

Now that we have defined the structure of a string using model theory. Let's practice with the other strings from above. Using the model signature that has been given, write out the full domain and denotations for all strings, but also draw them graphically.

1. $aaaa$
2. $darp$
3. $drap$
4. $tarp$

What do you notice about the difference in terms of *number of elements*, *properties of elements*, and *order of elements*?

The model signature above is a relational model because it only contains relations. But there is no need to restrict the signature to only one type of thing. We can freely mix functions and relations. Another model signature that is sometimes used for talking about strings is $\langle\{\mathtt{p}(), \mathtt{s}()\}, \{R_\sigma \mid \sigma \in \Sigma\}\rangle$. This signature replaces the successor *relation* with both the successor and predecessor *functions*. As was the case with the above signature, *properties of elements* of the domain and the *number of elements* are defined the same way, but now we impose an order through *functions* instead of a single relation. Lets see what this looks like once again using the string aaa and the alphabet $\Sigma = \{a, d, t, r, p\}$. Figure 4.2 shows this.

$$
\begin{aligned}
&\langle D = \{0, 1, 2\} \\
&\mathtt{p}() \stackrel{\text{def}}{=} \mathtt{p}(0) = 0; \mathtt{p}(1) = 0; \mathtt{p}(2) = 1 \\
&\mathtt{s}() \stackrel{\text{def}}{=} \mathtt{s}(0) = 1; \mathtt{s}(1) = 2; \mathtt{s}(2) = 2 \\
&\quad a = \{0, 1, 2\} \\
&\quad d = \{\} \\
&\quad t = \{\} \\
&\quad r = \{\} \\
&\quad p = \{\}\rangle
\end{aligned}
$$

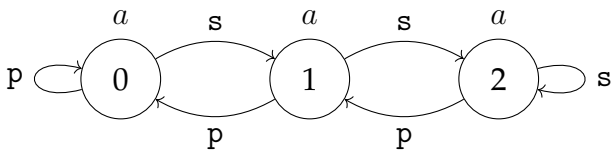


Figure 4.2: A different Σ-structure for string aaa.

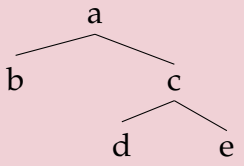
We have focused on strings, but trees are also easily definable with the model theory approach (see block below!). But it's not limited to just strings and trees. What's nice about model-theoretic representations is that they are both rigid in that they are mathematically well-defined objects, but flexible in the sense that many different types of linguistic objects can be defined this way. Really, your imagination is the limit!

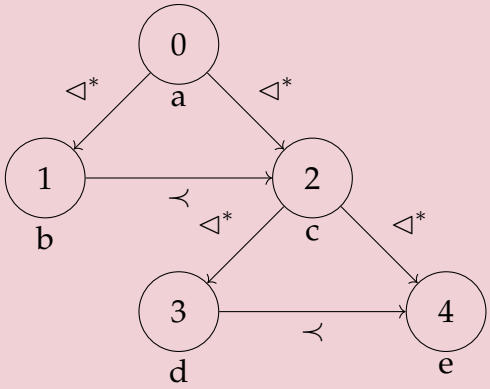
For example, phonologists working with model theory have used it to describe not only segmental strings as we have looked at so far, but also strings of feature values, autosegmental representations, and syllable trees. Some have even used it to look at phonological structures in signed languages as well as gestural representations used in theories like Articulatory Phonology.

Model-Theoretic Representations of Trees

We have seen both relational and functional models that formalize our notion strings. We can do exactly the same for trees by using different types of relations and functions. Here, we will briefly introduce relational models for trees.

Consider an alphabet Σ and a model $\langle D; \lhd^*, \prec, \{R_\sigma \mid \sigma \in \Sigma\}\rangle$ where $D \subsetneq \mathbb{N}$ is a set of nodes, $\lhd^*$(x,y) is the (binary) general dominance relation, $\prec (x, y)$ is the (binary) sisterhood relation, and $\sigma_i(x)$ is a (unary) relation for each $\sigma_i \in \Sigma$. An example is given below, where the left side is the tree itself and the right side is a model representation of the tree:





The signature uses the general dominance relation $\lhd^*$, but for simplicity of presentation only those nodes which follow immediate dominance are shown on the tree models. The immediate domination relation $\lhd$ can be defined in the following way:

$$\lhd(x, y) \stackrel{\text{def}}{=} [x \lhd^* y \wedge x \neq y \wedge \forall z[(x \lhd^* z \wedge z \lhd^* y) \rightarrow (x = z \vee z = y)]]$$

In this tree, the relations are configured in the following way:

$$\begin{aligned} \lhd^* &= \{(0,1),(0,2),(0,3),(0,4),(2,3),(2,4)\} \\ \prec &= \{(1,2),(3,4)\} \\ a = \{0\}, b &= \{1\}, c = \{2\}, d = \{3\}, e = \{4\} \end{aligned}$$

Questions about Tree Structures

1. Describe the properties of the general dominance relation. More specifically, is it reflexive, transitive, symmetric, etc.? What about the sisterhood relation?
2. Informally, how would you define the C-command relation $C(x, y)$ using the relations we have at hand?
3. What are some other relations you could imagine that can be defined in terms of $\lhd^*$ and $\prec$ that could be useful for formalizing syntactic operations?
4. Using an alphabet Σ of your choice, define some tree structures that correspond to valid syntactic trees for English sentences.

4.3 First-Order Logic

The model-theoretic representations we defined above are only one piece of the puzzle. Once we have the representations, we also need a way of talking about them. Our way of talking about these objects will be with mathematical logic. Specifically, we will use first-order logic (and fragments thereof) because it provides variables which in turn allow us to target specific elements within the domain.[4] An informal introduction is given below.

Recall our discussion of Propositional Logic from Lectures 1 and 2. We discussed the standard boolean connective AND ($\wedge$), OR ($\vee$), NOT ($\neg$), IF ($\rightarrow$), and IFF ($\leftrightarrow$). These were used to reason about *propositions* and different logical truths emerged depending on how we defined the assignment/interpretation function i.

First-order logic extends Propositional Logic in the following ways. First, we add **first-order variables** which range over elements of a given domain. Second, we add **quantification** which allow us to infer properties that range over the domain. There are two quantifiers that are used: the universal quantifier ($\forall$) and the existential quantifier ($\exists$). The universal quantifier is often read as "for all" and allows us to check if a property is true for *all* elements in the domain. The existential quantifier is often read as "there exists" and allows us to check if a property is true for *at least one* element in the domain. Third, we add the concept of **equality**. This allows us to check the equality of two variables. Table 4.1 lists all of the symbols (excluding things like parentheses and commas) that we are allowed to use in first-order logic.

We define a **logical language** in first-order logic by combining the symbols available to us (Table 4.1) with a specific model signature Σ. From this, we define a Σ-formula as any logical formula where all the non-logical symbols are drawn from Σ. In other words, it is a formula that uses the relations and functions from the specified model signature Σ.

[4] First-order logic is often called *predicate* logic because it reasons over properties of the domain and properties are determined based on functions and relations (i.e. predicates).

	Name	Meaning
x, y, z	variables	Elements of the domain
$R_1 ... R_n$	relations	Order/Properties of the domain
$F_1 ... F_n$	functions	Order/Properties of the domain
$C_1 ... C_n$	constants	
$\wedge$	conjunction	"And"
$\vee$	disjunction	"Or"
$\neg$	negation	"Not"
$\rightarrow$	implication	"If...then"
$\leftrightarrow$	bi-direction	"Same"
$\exists$	existential quantifier	"There Exists"
$\forall$	universal quantifier	"For All"
$=$	equality	"Equals"

Table 4.1: Symbols and their meaning in First-Order Logic

Σ-formula

Suppose we have the following model signature: $\Sigma = \langle \mathtt{p}(), \mathtt{s}(), R_C, R_V, R_{\ltimes}, R_{\ltimes} \rangle$. Determine which of the following formula are Σ-formula.

1. $V(x) \wedge \ltimes(\mathtt{s}(x))$
2. $G(x) \wedge \ltimes(\mathtt{s}(x))$
3. $G(x) \wedge \ltimes(\mathtt{A}(x))$

The model signature above is used to define strings of C's and V's (which stand for consonants and vowels). Turn the following statements into their corresponding Σ-formula.

4. If an element is a vowel, then it is followed by a consonant.
5. An element is both a vowel and preceded by a consonant.
6. An element is at the beginning of a word if and only if it is a consonant.

A Σ-formula like $Qx(\phi)$ (where Q is a quantifier, x a variable, and ϕ a well-formed formula) is called a Σ**-sentence** given that ϕ contains no variables other than x. We say that ϕ is in the scope of Qx and any instance of x in ϕ are **bound** by it. Any variables in ϕ other than x are said to be **free**. The full definition of a Σ-sentence is any Σ-formula with no free variables.

Why do we care about Σ-sentences and not just Σ-formula? Σ-Sentences are interpreted without needing to assign any of the variable to a specific element of the domain. On the other hand, Σ-formula require explicit assignment of variables to elements of the domain in order to be interpreted. A Σ-formula is **satisfiable** if it evaluates to true under *some* assignment of the variables in the domain. A Σ-formula is **valid** if it evaluates to true

under *every* assignment of the variables in the domain. Notice, this corresponds directly to existential (some) and universal (every) quantification. So we can turn any Σ-formula into a Σ-sentence depending on if we want a given property to hold for a specific element in the domain or every element in the domain.

Recall above, where we defined a Σ-structure as any structure built from a model signature Σ. Suppose we have a Σ-structure A and a Σ-sentence ϕ. If ϕ evaluates to `true` then we write $A \vDash \phi$ to mean A *satisfies/models* ϕ. If ϕ evaluates to `false` then we write $A \nvDash \phi$ and say A *does not satisfy/model* ϕ. From this definition of satisfiability, we can ask the following three questions:

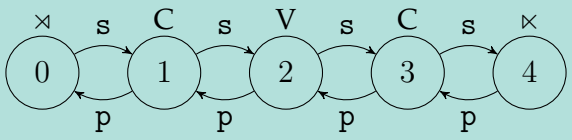
1. For a fixed Σ-structure A, which Σ-sentences does it satisfy?
   - ♦ $\text{Theory}(A) \stackrel{\text{def}}{=} \{\phi : A \vDash \phi\}$
2. For a fixed Σ-sentence ϕ, which Σ-structures satisfy it?
   - ♦ $\text{Spectrum}(\phi) \stackrel{\text{def}}{=} \{A : A \vDash \phi\}$
3. Given a set of Σ-sentences T, what are the Σ-structures that satisfy *all* Σ-sentences in T?
   - ♦ $\text{Model}(T) \stackrel{\text{def}}{=} \{A : \forall \phi \in T[A \vDash \phi]\}$

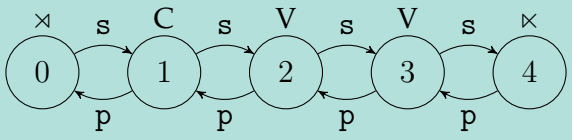
Σ-Formula/Σ-Sentences and Satisfiability

Suppose we have the following model signature: $\Sigma = \langle \mathtt{p}(), \mathtt{s}(), R_C, R_V, R_{\rtimes}, R_{\ltimes} \rangle$. Determine which of the following formula are Σ-formula and which are Σ-sentences.

1. $V(x) \wedge \ltimes(\mathtt{s}(x))$
2. $\exists x(V(x)) \wedge \ltimes(\mathtt{s}(x))$
3. $\exists x(V(x) \wedge \ltimes(\mathtt{s}(x)))$

Given the Σ-sentence $\phi = \exists x(V(x) \wedge \ltimes(\mathtt{s}(x)))$, which of the following structures model/satisfy ϕ?





Bonus: Write a set of Σ-sentences T such that T models all and only the strings in the language $(CV)^+$.

4.4 Model-theoretic Formal Language Theory

Recall the Subregular Hierarchy from Lecture 1 and 2, repeated below:

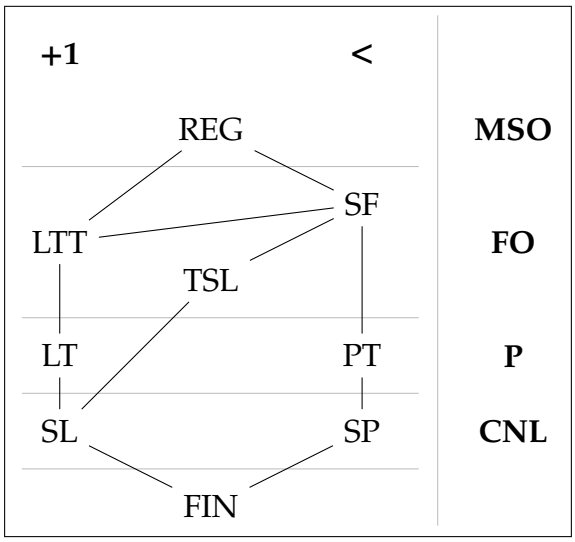


Figure 4.3: The Subregular Hierarchy; +1 refers to constraints defined using immediate successor, whereas < referes to constraints defined using general precedence; MSO refers to Monadic Second Order Logic, FO refers to First Order Logic, P refers to Propositional Logic, and CNL refers to Conjunction of Negative Literals; Combinations of logic and representation result in different language classes (explained in more detail in Lecture 3).

4.4.1 Strictly Local Languages

Remember that in Lecture 3 we discussed the Strictly Local languages and defined a language to describe intervocalic voicing. We showed an automata-theoretic characterization of this phenomenon by constructing a finite state acceptor which accepted all and only such strings. Here however, we will be seeing how a certain restriction on Propositional Logic will give a convenient model-theoretic characterization.

In the definition of Propositional Logic, atoms are propositions $P, Q, R, \ldots$ which are statements that are either true or false. The term **literal** simply means any atomic formula or its negation: so if P is an atom, then both P and $\neg P$ are called literals. The former is called a positive literal and the latter is caleld a negative literal. If we take Propositional Logic as we've defined it previously and restrict it so that the only connectives allowed are *conjunction* ($\wedge$) and *negation* ($\neg$), we get the logic called **Conjunction of Negative Literals** (CNL). Note though that the negation is restricted in that it may only be used to negate a literal P (see Exercise 1 on the following page).

Interestingly, note that the definition of a Σ-sentence is a Σ-formula with no free variables, which ultimately describes a structure through a statement that is either true or false. This is exactly what a proposition is. So in a sense, the literals that we use in Propositional Logic and CNL are just Σ-sentences of First-Order Logic interpreted as individual units.

Using the successor relation $\lhd(x, y)$ in combination with Conjunction of Negative Literals defines languages by simply stating all of the k-length (or less) combinations that are disallowed from occurring in any string in the language. Thus, a Strictly Local language is one that can be specified by statements of the following form:

$$\neg(X_1) \wedge \neg(X_2) \wedge \ldots \neg(X_n),$$

where $n \in \mathbb{N}$ and each $X_{1 \leq i \leq n}$ is a string of k or less alphabet symbols $\sigma \in \Sigma$. In other words, a string w is well-formed if it doesn't contain any $X_{1 \leq i \leq n}$. This logic using the immediate successor relation $\lhd$ yields exactly the Strictly Local languages.

Conjunction of Negative Literals

1. Notice that we've defined CNL in such a way that the only connectives it has the ability to use are conjunction and negation. Recall the exercise from Lecture 1 and 2 where we used $\neg, \wedge$ to define all of the other connectives of Propositional Logic. What restriction *must hold* of negation so that we don't simply get back immediately to Propositional Logic?

2. What are some other phenomena that can be formalized as a well-formedness condition using a Strictly Local language? What are some other phenomena *cannot* be formalized in this way?

Returning to the example, intervocalic voicing can be viewed as a well-formedness condition on strings such that no strings contain a voiceless consonant in between two vowels.

We formalize this using the alphabet $\Sigma = \{V, T, D\}$ where V stands for all vowels, T stands for all voiceless consonants, and D stands for all voiced consonants. The language L accepts all and only strings in Σ^* which do not contain the substring VTV.

Recall that this requires a window of size $k = 3$ since we know that substrings like VTV and $DVTVD$ are disallowed since a voiceless segment occurs between two vowels, but substrings like VDV and $TVDVT$ are allowed since voiced segments occur between two vowels. Very succintly, we can use CNL to state this language exactly: $\neg(VTV)$

Of course one can think of this well-formedness purely in terms of restrictions on substrings within a string; however, that would be missing a huge generalization seeing as these tools have the ability to characterize finite structures other than strings as well. Namely, we can think about well-formedness in terms of *restrictions on sub-structures*. Let's see this explicitly in terms of strings as Σ-structures of some signature.

With intervocalic voicing in mind, consider the definition and corresponding graph representation of the following Σ-structure:

$$\begin{aligned}\langle D &\stackrel{\text{def}}{=} \{0,1,2,3,4,5\}\\ \lhd &\stackrel{\text{def}}{=} \{(0,1),(1,2),(2,3),(3,4),(4,5)\}\\ V &\stackrel{\text{def}}{=} \{0,2,4\}\\ D &\stackrel{\text{def}}{=} \{1,5\}\\ T &\stackrel{\text{def}}{=} \{3\}\rangle\end{aligned}$$

In Figure 4.4, the red color indicates a substructure that is disallowed by this language. In other words, the negative literal $\neg VTV$ is included in this string, and so it is not in the language. Note that in this graphical representation of our banned substructure, it is expressed entirely by a window of nodes which are directly connected, which is a hallmark property of *strict locality*.

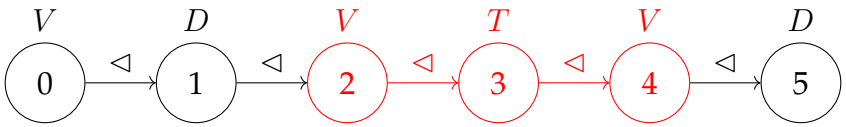


Figure 4.4: Σ-structure for the string $VDVTVD$

Banned Substructures and Strict Locality

1. For the intervocalic voicing language, try to define some well-formed Σ-structures with respect to this language, graphically and/or pictorally.

2. Using the same alphabet $\Sigma = \{V, T, D\}$, consider the phenomenon of word final devoicing where a string cannot end in a voiced obstruent. Is this a Strictly Local language as well? If so, try to explain what sorts of banned substructures would describe this language, formally and/or graphically (Hint: use $\rtimes, \ltimes$).

3. **Bonus**: Knowing that we can define other types of finite structures, like trees or autosegmental graphs for instance, what might banning substructures in this strictly local sense look like in those types of structures?

Since we now know (i) the distinction between the immediate successor $\lhd$ and general precedence relations $<$ and (ii) CNL combined with $\lhd$ yields precisely the Strictly Local languages, a natural question to ask is: *What language class do we get when we combine CNL with the general precedence relation?*

This combination yields the Strictly Piecewise Languages. In the string case, while Strictly Local languages enforce inviolable constraints on *substrings* of length k (or less), the Strictly Piecewise do the same but for *subsequences* of length k (or less).

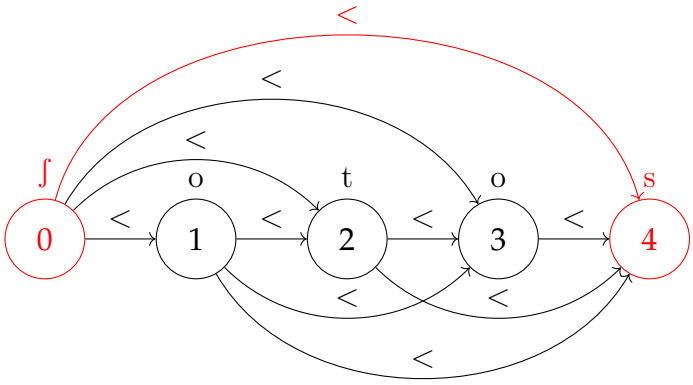
4.4.2 Strictly Piecewise Languages

Recall the examples on subsequences from Lecture 3 repeated below:

Samala is a language native to the California area. It has a pattern of *symmetrical consonant harmony* where words cannot contain both [s] and [ʃ]. Words like [posokoso] and [poʃokoʃo] are well formed, while words like *[posokoʃo] and *[poʃokoso] are ill-formed.

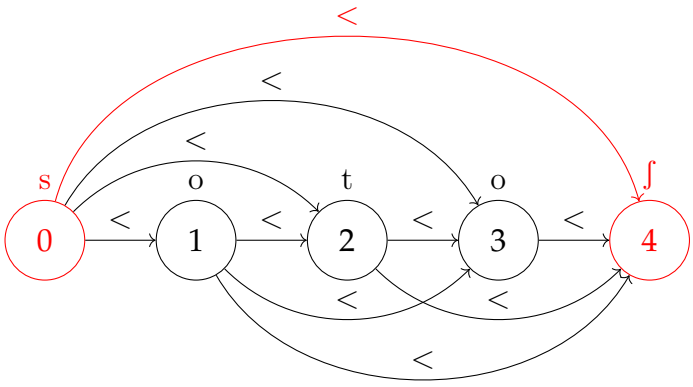
This would simply require banning both the subsequences ʃ...s and s...ʃ, since any occurrence of both in a string makes it ill-formed. Thus, this language is Strictly Piecewise since it is described using bans on the occurrence of particular subsequences. Consider the Σ-structure, defined formally and graphically below. Notice that this is defined with the general precedence relation $<$.

$$
\begin{aligned}
\langle D &\stackrel{\text{def}}{=} \{0,1,2,3,4\} \\
< &\stackrel{\text{def}}{=} \{(0,1),(0,2),(0,3),(0,4),(1,2),\\
&\qquad (1,3),(1,4),(2,3),(2,4),(3,4)\} \\
\text{ʃ} &\stackrel{\text{def}}{=} \{0\} \\
\text{s} &\stackrel{\text{def}}{=} \{4\} \\
\text{o} &\stackrel{\text{def}}{=} \{1,3\} \\
\text{t} &\stackrel{\text{def}}{=} \{2\}\rangle
\end{aligned}
$$



As before, the banned substructure in the graphical representation of our structure is represented by the color red. Note that in this case, the restriction is on not a window of nodes immediately adjacent to each other, it is on a window of nodes generally preceding eachother with arbitrarily many nodes in between them. Since in Samala this is a symmetrical restriction where neither ʃ can precede s nor s can precede ʃ, the following Σ-structure is also ill-formed. The only difference in their structure is that the ʃ and s that nodes 0 and 4 bore were switched.

$$
\begin{aligned}
\langle D &\stackrel{\text{def}}{=} \{0,1,2,3,4\} \\
< &\stackrel{\text{def}}{=} \{(0,1),(0,2),(0,3),(0,4),(1,2),\\
&\qquad (1,3),(1,4),(2,3),(2,4),(3,4)\} \\
\text{ʃ} &\stackrel{\text{def}}{=} \{4\} \\
\text{s} &\stackrel{\text{def}}{=} \{0\} \\
\text{o} &\stackrel{\text{def}}{=} \{1,3\} \\
\text{t} &\stackrel{\text{def}}{=} \{2\}\rangle
\end{aligned}
$$



So, any structures for which $\text{ʃ}(x) \wedge \text{s}(y) \wedge (x < y \vee y < x))$ holds will not be in the language.

Sarcee is an Athapaskan language with an *asymmetrical consonant harmony* pattern. In Sarcee, [s] may follow [ʃ], but [ʃ] may not follow [s]. So words like [posokoso] and [poʃokoʃo], and [poʃokoso] are all well-formed, but words like *[posokoʃo] are ill-formed.

Note that due to the asymmetrical nature of the pattern, this would only require banning the subsequence s...ʃ, since strings where s folllows ʃ are permissible. Thus, this language is also Strictly Piecewise. So the substrcutres banned by this language are only those where s precedes ʃ. In the structures shown above, while Samala requires banning substructures of both forms, Sarcee only requires banning those of the second form.

Strictly Piecewise Languages

1. Let L_{sar} and L_{sam} be the languages described by the Sarcee and Samala patterns, respectively. What are some examples of strings $w \in L_{sar}$? $w \in L_{sam}$? $w \in L_{sar} \cup L_{sam}$? $w \in L_{sar} \cap L_{sam}$?

2. Define a Strictly Piecewise language of your own, using an alphabet Σ of your choice.

3. Are all Strictly Local languages examples of Strictly Piecewise languages? Why or why not?

4.4.3 Tier-Based Strictly Local Languages

Recall our example from Lecture 3 regarding Tier-Based Strict Locality. We defined Tier-Based Strict Locality in the following way:

A language L is TSL-k if there exists a $k \in \mathbb{N}$ and a tier $T \subseteq \Sigma$ such that all strings in L are SL-k when restricted to only the symbols in T.

This just means that it is Strictly Local when relativized to some relevant subset of the alphabet. Our toy vowel harmony example is repeated below, where F stands for front segments, B stands for back segments, and C stands for consonants.

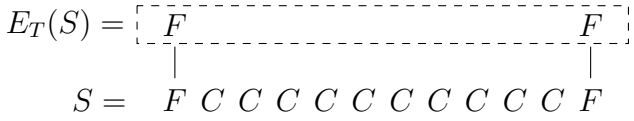


Figure 4.5: Tier Construction

In Figure 4.5 we see that even though the two F's can be arbitrarily far from one another, when only considering F's and B's while erasing all C's, this language is SL-2, since any occurence of FB or BF on the tier will make the entire string ill-formed.

In our models so far, we have a way of talking about segments being immediately adjacent, or generally preceding one another. If we want to restrict our view to only some relevant

subset of symbols, we can use these relations to do so. Suppose we wanted to express a relation between elements x and y in the string that states x generally precedes y and both are either F or B. This is simply done by saying x precedes y and both bear either F or B as a label:

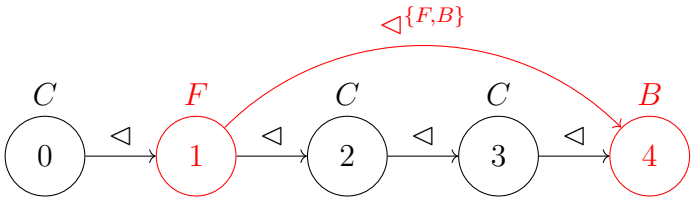
$$<^{\{F,B\}}(x,y) \stackrel{\text{def}}{=} <(x,y) \ \wedge \ (F(x) \vee B(x)) \ \wedge \ (F(y) \vee B(y))$$

Call this $\{F,B\}$-precedence. From this, we can directly define immediate $\{F,B\}$-precedence denoted $\lhd^{\{F,B\}}(x,y)$ by stating that $<^{\{F,B\}}(x,y)$ and there is no intervening element that bears either F or B in between x and y. Formally,

$$\lhd^{\{F,B\}}(x,y) \stackrel{\text{def}}{=} <(x,y) \wedge \neg\exists[z \mid <^{\{F,B\}}(x,z) \wedge <^{\{F,B\}}(z,y)]$$

Observe the Σ-structure below for our toy vowel harmony example:

$$\begin{aligned}\langle D &\stackrel{\text{def}}{=} \{0,1,2,3,4\}\\ \lhd &\stackrel{\text{def}}{=} \{(0,1),(1,2),(2,3),(3,4)\}\\ \lhd^{\{F,B\}} &\stackrel{\text{def}}{=} \{(1,4)\}\\ F &\stackrel{\text{def}}{=} \{1\}\\ B &\stackrel{\text{def}}{=} \{4\}\\ C &\stackrel{\text{def}}{=} \{0,2,3\}\end{aligned}$$



In this example, the banned substructure still makes reference to a local window of elements, but relativized to only a certain subset of them, namely F and B. This casts non-local phenomena in such a way that they are evaluated locally, just in a *relative* sense. This is different than the Strictly Piecewise case because the relation used to ban substructures requires (relative) immediate adjacency thus remaining local, whereas Strictly Piecewise languages require general precedence.

Of course this is simply one example, where in general we have a relation $\lhd^T(x,y)$ for any subset of relevant symbols $T \subseteq \Sigma$. When combining CNL with $\lhd^T$, we get precisely the Tier-Based Strictly Local languages.

Tier-Based Strict Locality

1. Are all Strictly Local languages Tier-Based Strictly Local languages? Why or why not?

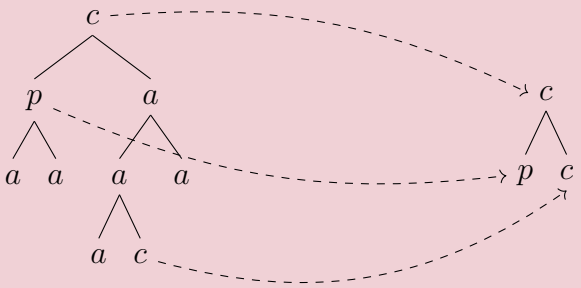
2. What sorts of patterns can TSL capture that SP can't? Consider the pattern in Slovenian where an ʃ cannot occur after an s unless there is an intervening t (Jurgec, 2011). Would this be TSL or SP, and why? (Hint: it is not both!)

3. Consider $\Sigma = \{C$,u,ʊ,e,ɛ$\}$ where u,e are +ATR and ʊ,ɛ are -ATR. Show the banned substructures for a language that satisfies ATR harmony.

Tier-Based Strict Locality in Syntax

This idea of Tier-Based Strict Locality can be extended to trees and can account for a wide range of syntactic phenomena. See Vu et al. (2019); Graf (2022a,b,c); Hanson (2023).

As a brief toy example, consider the arbitrary alphabet $\Sigma = \{a, c, p\}$, and the set of all possible trees over this alphabet. Suppose I want to ban configurations where more than one c dominates a p. The relevant symbols in the tree-tiers are going to be c and p. We can check a $\{c, p\}$ tree-tier for banned tree configurations:



This tree on the left is then in the language since this $\{c, p\}$ tier doesn't have two c's dominating any p.

Using some other machinery (minimalist grammars), nodes can bear features, and one can construct tier projections of elements bearing a particular feature.

For example, if in a tree you project a wh-tier and you have a mismatch of numbers of elements (a wh^+ and wh^- check each other, but one is leftover) you can use this to derive certain types of well-formedness constraints in syntactic derivations.

4.5 Interpretations

Up to this point we have primarily been discussing *string sets* in relation to linguistics and formal language theory, but there is parallel work, especially in phonology, about *mappings* and formal language theory. A *subregular* hierarchy for function classes is shown in Figure 4.6 below. We won't have the time to get into the details of this hierarchy in this class, but you may notice some similar terminology here. As was the case with the subregular hierarchy of stringsets, the idea of locality (and relativized locality) plays a big role in these function classes as well.

Instead of focusing on the specific formalization of these classes, we will instead focus on the idea of **interpretations** as they pertain to model theory. The crude idea is that we can *interpret* one structure in terms of another structure. While interpretation in formal logic is often used for checking the decidability of different mathematical theories, computer

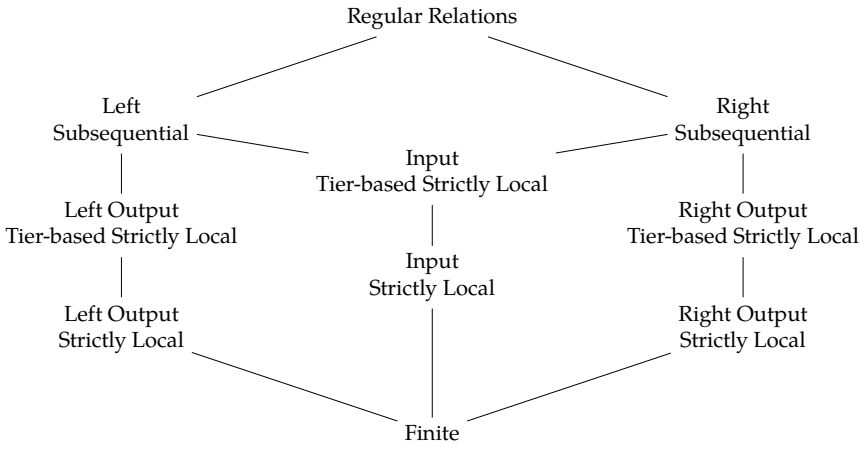


Figure 4.6: A hierarchy for *subregular* functions.

scientists realized that it can also be used to describe mappings. Recall that a map is just another term for a function. So what we are going to describe is a mapping from one linguistic object to another.

We will introduce interpretations informally in the following way. An interpretation is a function $\pi : \Sigma \to \Gamma$ where Σ and Γ are both model signatures. The denotation of the function, requires the following elements: a function $\phi_{\texttt{domain}}$ which determines which elements of the input are in the domain of the function, a copy set C which determines how many copies of each input element to include in the output, a function $\phi_{\texttt{license}}$ which determines which elements in the output structure are licensed, and a denotation for every function and relation in the output signature Γ described only with Σ-formula.

Suppose we had the following two model signatures:

$$\Sigma = \langle \texttt{p}(), \texttt{s}(), R_a, R_b \rangle$$
$$\Gamma = \langle \texttt{p}(), \texttt{s}(), R_i, R_d \rangle$$

We can interpret Γ in terms of Σ in the following way. For all functions and relations, ϕ_F or ϕ_R refer to the *output* denotations while F or R refer to the input denotations.

$$\phi_{\texttt{domain}} \stackrel{\text{def}}{=} \texttt{True}$$
$$C \stackrel{\text{def}}{=} \{1\}$$
$$\phi_{\texttt{license}(x)} \stackrel{\text{def}}{=} \texttt{True}$$
$$\phi_{\texttt{p(x)}} \stackrel{\text{def}}{=} \texttt{p(x)}$$
$$\phi_{\texttt{s(x)}} \stackrel{\text{def}}{=} \texttt{s(x)}$$
$$\phi_i(x) \stackrel{\text{def}}{=} a(x)$$
$$\phi_d(x) \stackrel{\text{def}}{=} b(x)$$

What does this do? Well we keep the predecessor and successor values from the original structure, but swap all of our a's to i's and b's to d's. So a string like $bababa$ which is a Σ-structure becomes the Γ-structure $dididi$. We say that the d elements in $dididi$ are interpreted as d only because their corresponding element in $bababa$ is a b.

The logical formula above provide a denotation of a function or *transduction*. Under this view we are mapping from a Σ-structure to a Γ-structure. But ultimately we are interpreting a new Γ-structure in terms of an old Σ-structure.

4.5.1 Phonological Maps

Due to their domain-general nature, model-theoretic interpretations prove to be extremely useful for understanding the interaction between different linguistic structures. In the previous section we saw how an output Γ-structure can be interpreted in terms of an input Σ-structure. Specifically within phonology, interpretations can give very natural computational explanations of how underlying representations (UR's) transform into surface representations (SR's). There is a sense in which this interpretation of SR's in terms of UR's is essentially the same thing as a transformation, or transduction, from UR's to SR's.

For example, consider the rule $a \to b / c\_\_d$. Ultimately what this tells is that whenever we have a UR containing the string cad, there will be a corresponding SR that *interprets* that substructure as cbd. Given the machinery for interpretations we have sketched above, this is equivalent to saying there is an interpretation with the functions ϕ_a, ϕ_b, ϕ_c, and ϕ_d with the following denotations:[5]

$$\phi_a \stackrel{\text{def}}{=} a(x) \wedge \neg[c(\mathtt{p}(x)) \wedge d(\mathtt{s}(x))]$$
$$\phi_b \stackrel{\text{def}}{=} b(x) \vee [a(x) \wedge c(\mathtt{p}(x)) \wedge d(\mathtt{s}(x))]$$
$$\phi_c \stackrel{\text{def}}{=} c(x)$$
$$\phi_d \stackrel{\text{def}}{=} d(x)$$

As we will see toward the end of these lecture notes, this formal machinery is not only mathematically robust and far reaching in its ability to formalize linguistic phenomena across a variety of different domains, but it also allows us to be sufficiently restrictive while curating a theory of linguistic computation. Striking this balance between expressivity and restrictivity is crucial not only for phonology, but for scienctific inquiry in general.

Also important to note is the difference between what is called an imperative and declarative approach to computation. When computing something imperatively, to complete a given task the programmer explicitly states the steps sequentially as a series of commands. The exact control flow to perform a given task is specified from beginning to end as a series of instructions that correspond to statements along the lines of if X happens, then do Y, otherwise Z happens, then do.... On the other hand, when using a declarative approach, the form of the desired outcome of a procedure is stated without stating the steps of the

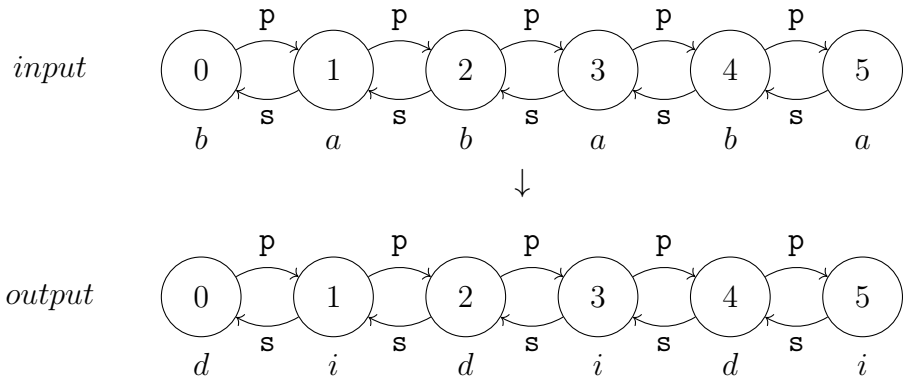
[5]We omit ordering relations as well as domain, licensing, and copy set info since they are irrelevant to the map at hand.

procedure itself explicitly. These interpretations are declarative in nature because we are describing the form of the output structure given some configurations in the input structure. This says nothing about where to start, where to finish, or how to do it: rather, it simply states that in the output a particular piece of the structure will have *this* property if a corresponding piece of the input structure had *that* property. Statements simply pick out pieces of the structure where some property holds true, and ensures that corresponding properties hold true to the output structure. Thus, we are describing the desired outcome of the computation more or less agnostically to things like steps in a sequence.

In the previous section we saw an example where the input structures were strings that consisted of a's and b's, and the output structures were strings that consisted of i's and d's in the place of the input a's and b's. There are two pieces of this mapping that are important:

- Labeling relations: When is something in the output a d or an i?
- Ordering relations: Where do `p` and `s` hold in the output?

Very simply, something is an i in the output if it was an a in the input, something is a d in the input if it was a b in the input, and the order of the elements remains the same. These were defined in the previous section, but below a diagram of this interpretation is shown:

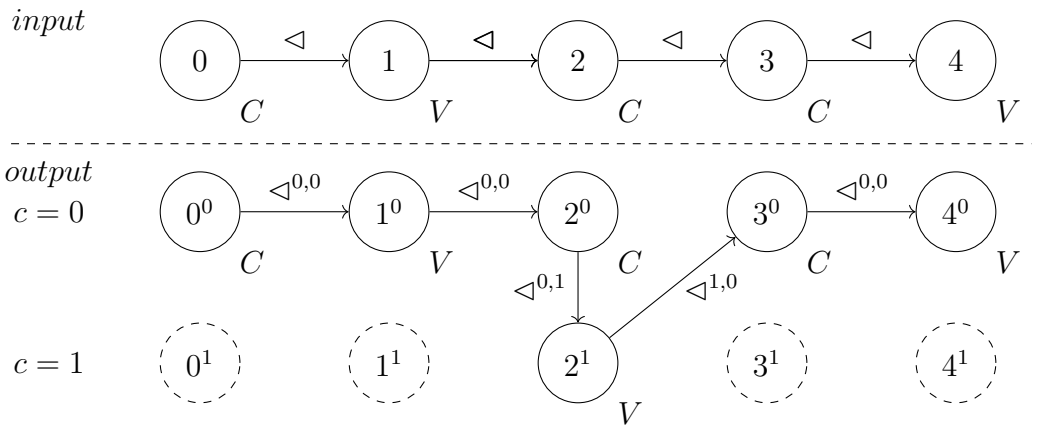


This example maps strings to strings of the same size since nothing is added or deleted. This is because the copy set was $C = \{1\}$, and so the string cannot have more elements in the output than it did in the input. Suppose we wanted an example where we complete epenthesis, where a vowel is inserted in between two consonants in the input. As we will see, this will require a copyset $C = \{0, 1\}$.

Consider the alphabet $\{C, V\}$ to represent consonants and vowels. We want to define an interpretation where any given input string that contains a CC substring will result in a CVC substring. This requires a copyset of $C = \{0, 1\}$ since the output can be larger than the input, there must be a place to insert the vowel in between the two consonants and this is what the copies are used for.

Below, a relational model of the String $CVCCV$ is shown as the input Σ structure. Our output Γ structure is the string $CVCVCV$, in accordance with our rule. Below the inter-

pretation is shown explicitly. At first this may look slightly daunting, but a little unpacking of each piece will make it clear how natural this mapping is.



First note that in the digram above, the dotted nodes indicate unused copies, those that are not relevant to the output structure. Any copies have the potential to be used, but it is the explicit definitions of the interpretation determine which are relevant.

From here, let's start with the labeling relations. In the output structure, something will be a C iff it was a C in the input. A C will never be inserted since our mapping will only ever insert V's between two C's. The 0-th copies, x^0's, are reserved for those nodes that were already in the input string and the 1-st copies, x^1's, are reserved for those nodes that *add* to the string. So $C(x)$ will only ever be true for the x^0 copies and not the x^1 copies. We can make the following definitions to reflect this below:

$$\phi_{C^0}(x) \stackrel{\text{def}}{=} C(x)$$
$$\phi_{C^1}(x) \stackrel{\text{def}}{=} \bot$$

The V's are slightly more complicated since something will be a V in the output if it was a V in the input; however, it will *also* be a V in the output if that piece of the string consisted of a CC substring. Thus, a V will occur in the output if it was a V in the input or if it was a C in the input immediately followed by another C.

$$\phi_{V^0}(x) \stackrel{\text{def}}{=} V(x)$$
$$\phi_{V^1}(x) \stackrel{\text{def}}{=} C(x) \wedge \exists y[C(y) \wedge \lhd(x, y)]$$

This covers all of the labeling relations. Let's move to the ordering relations. Our input signature is ddefined in terms of immediate precedence $\lhd(x, y)$, and so will our output signature; however, there are some adjustments that must be made. Since we have different copies, we must be able to mediate precedence between different copies. Since we have two copies, 0 and 1, this entails splitting in up into $2^2 = 4$ different relations. These are discussed below where $\lhd^{i,j}(x, y)$ indicates that the i-th copy of x immediately precedes the j-th copy of y: Let's break each of these down.

The $\phi_{\lhd^{0,0}(x,y)}$ relation mediates immediate precedence between nodes where both were unchanged from the input. This will only be true whenever both x and y are not C's that immediately precede eachother. This is defined formally below:

$$\phi_{\lhd^{0,0}(x,y)} \stackrel{\text{def}}{=} \neg(C(x) \wedge C(y) \wedge \lhd(x,y))$$

The $\phi_{\lhd^{0,1}(x,y)}$ relation mediates immediate precedence between nodes where the second is an inserted V, since this is the only thing that will occupy the x^1 copies in our interpretation. This is defined formally below:

$$\phi_{\lhd^{0,1}(x,y)} \stackrel{\text{def}}{=} x = y \wedge C(x) \wedge \exists z[C(z) \wedge \lhd(x,z)]$$

Similarly, the $\phi_{\lhd^{1,0}(x,y)}$ relation mediates immediate precedence between nodes where the first is an inserted V. This is defined in the following way:

$$\phi_{\lhd^{1,0}(x,y)} \stackrel{\text{def}}{=} x \neq y \wedge C(x) \wedge C(y) \wedge \lhd(x,y)$$

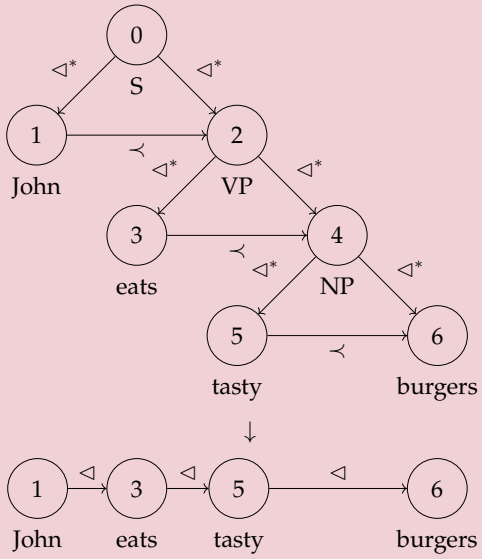
The $\phi_{\lhd^{1,1}(x,y)}$ relation mediates immediate precedence between inserted V's; however, you may notice that this will never occur since a V will only be inserted between two C's, so this one is always false.

$$\phi_{\lhd^{1,1}(x,y)} \stackrel{\text{def}}{=} \bot$$

Thus, all of the output relations have been defined

Translation Between Different Structures

We have seen that we have the can use Σ-structures to formally characterize both strings and trees. If we think about the procedure of linearization, this can be formalized as a function from tree structures to string structures. This procedure will not be defined explicitly here, but an example is shown below.



As before, our tree structures are defined using general dominance $\lhd^*$ and our string structures are defined using the immediate successor relation $\lhd$. In this sense, the symbol $\lhd$ happens to be overloaded but these are ultimately different relations.

4.6 Closing Thoughts and Further Reading

Closing Thoughts

In this lecture, we saw some of the tools from formal logic that mathematical linguists use for linguistic theorizing. More specifically, it was shown how model theory can be used to characterize different classes of languages in the subregular hierarchy, which is particularly useful for reasoning about phonological generalizations. We also saw briefly that these tools can be extended to trees for reasoning about syntactic generalizations. While these generalizations can be expressed using a variety of mathematical tools, there are well established connections between these tools, which show that there are very deep and fundamental properties underlying these patterns and classifications. It was also discussed how logic can be used to define mappings between structures of different kinds, which is of course very useful for the working linguist, who often thinks in terms of finite structures such as strings and trees.

Further Reading

For more on representation and comparison between varying representations using model theory, see Strother-Garcia (2019); Oakden (2020); Jardine et al. (2021); Nelson (2022).

For more on model-theoretic syntax, see (Rogers, 1996, 1998).

For more on formally expressing phonological processes using model theory, see Bhaskar et al. (2020); Chandlee and Jardine (2021); Oakden (2021).

For more on applications of model-theory to understanding some non-linear structures see Jardine (2017, 2019); Vu et al. (2022).

For more on learnability, see Rogers et al. (2013); Lambert et al. (2021); Rawski (2021).

For more on linguistically-motivated explanations of the underlying similarities between automata-theoretic, model-theoretic, and algebraic approaches to formal language theory, see Lambert (2022).

Practice Problems

The following are a list of practice problems that students may work on after the completion of each lesson. Individual practice problems are marked 1-3 stars indicating relative difficulty. Problems marked with ★ should be easily solved after reading the course notes. Problems marked ★★ & ★★★ may require you to extrapolate and think about the material in new ways.

Lesson 1

★ 1. Consider the two sentences below:

(a) John often goes to the store with Mary.

(b) Mary goes to the supermarket.

Let S_1 be the set of words in (a) and S_2 be the set of words in (b). Write out as sets:

- their union $S_1 \cup S_2$
- their intersection $S_1 \cap S_2$
- their set difference $S_1 \setminus S_2$

★ 2. Let $I = \{\text{i}, \text{ɪ}, \text{ɨ}\}$ and $U = \{\text{u}, \text{ʊ}, \text{ü}, \text{ʉ}\}$ be two sets of vowels. Write out their Cartesian Product $I \times U$. How many elements are in this set?

★★ 3. Consider the a set of segments $\Sigma = \{\text{n}, \text{m}, \text{ŋ}, \text{d}, \text{b}, \text{g}, \text{a}\}$ and a nasalization process that takes a form and adds a nasal with the corresponding place of articulation to any non-nasal consonant that appears right before a vowel. For example, the form bada would result in banda, the form baba would result in bamba, the form baga would result in baŋga, and so on.

This process can be considered a relation $\texttt{nas}(x, y)$ where x is the input and y is the output. Does this relation constitute a function? Why or why not? Give concrete examples to support your reasoning.

★★ 4. In the first lesson, we saw an example of a context-free language $a^n b^n$. Recall that recognizing this language required keeping a memory of arbitrarily many characters: in order to know how many b's are in the string, you have to know how many a's are in the string.

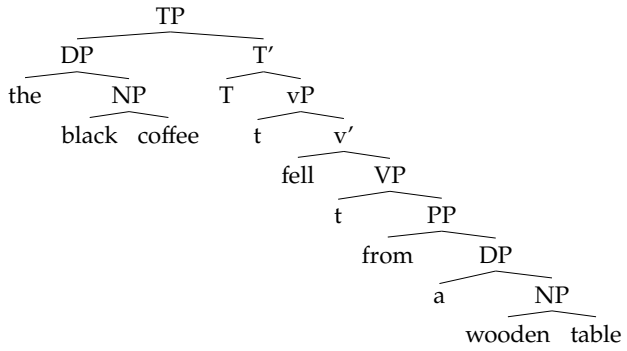
In stress assignmnet patterns, there is something called the Midpoint Pathology. In this pattern, stress targets the middle of a string for odd-length strings, and falls on the last syllable of the first half of the string for even length strings. Let σ be an unstressed syllable and $\acute{\sigma}$ be a stressed syllable. The Midpoint Pathology can be described as a mapping of inputs$\mapsto$outputs with some examples below:

$$\begin{aligned}
\acute{\sigma}\sigma\sigma &\mapsto \sigma\acute{\sigma}\sigma \\
\acute{\sigma}\sigma\sigma\sigma &\mapsto \sigma\acute{\sigma}\sigma\sigma \\
\acute{\sigma}\sigma\sigma\sigma\sigma &\mapsto \sigma\sigma\acute{\sigma}\sigma\sigma \\
\acute{\sigma}\sigma\sigma\sigma\sigma\sigma &\mapsto \sigma\sigma\acute{\sigma}\sigma\sigma\sigma \\
\sigma\acute{\sigma}\sigma\sigma\sigma\sigma &\mapsto \sigma\sigma\sigma\acute{\sigma}\sigma\sigma \\
&\vdots
\end{aligned}$$

Is the *resulting language* context-free? Why or why not?

Bonus: Does your answer change if for even-length strings, the stress falls on the first syllable of the second half of the string, or is it the same?

★★★ 5. In syntax, the **c-command** relation holds true of two nodes x, y in a tree if and only iff neither x nor y dominate each other and every node that dominates x also dominates y. Observe the tree below, where the following statements are true: "the" c-commands "coffee", "fell" c-commands "from", "a" does not c-command "fell", "coffee" does not c-command "fell".



- Is the c-command relation symmetric? Is it transitive? Why or why not?
- Explain briefly why this relation cannot be a function.

★★★ 6. In thinking about language production and perception, we may want to consider two types of functions: *phonological* functions which map between syntactic representations (SYN) and phonological representations (PF) and *semantic* functions which map between syntactic representations (SYN) and semantic representations (LF). To begin, define the types for the following four functions:

- $\texttt{Prod}_{\texttt{Phon}}$ - $\texttt{Perc}_{\texttt{Phon}}$ - $\texttt{Prod}_{\texttt{Sem}}$ - $\texttt{Perc}_{\texttt{Sem}}$

If you have done this correctly, you should now be able to define two composed functions. Do so now and then answer the following questions:

- What types do these functions have?
- What is the role of syntax under this view?
- Does this differ from the way in which syntax is normally viewed?

Lesson 2

★ 1. Prove using a truth table that $(P \to Q) \wedge (Q \to P)$ is logically equivalent to $P \leftrightarrow Q$.

★ 2. Write a recursive string to string function (like we did for the length of a string) that takes in an input string w and returns a string ww^R where w^R refers to the reversed version of the string. For example, the input string abc would return $abccba$. You should be able to do so with a single base case and single recursive case.

★★ 3. Let the symbols $\sigma, \acute{\sigma}$ represent unstressed and stressed syllables, respectively. Draw the graphical representation of the following Finite State Automaton and explain the language that it accepts.

$$\begin{aligned}\Sigma &= \{\sigma, \acute{\sigma}\}\\ Q &= \{0, 1, 2, 3\}\\ q_0 &= \{0\}\\ \delta &= \{((\sigma, 0), 1), ((\acute{\sigma}, 0), 3), ((\sigma, 1), 3), ((\acute{\sigma}, 1), 2), ((\sigma, 2), 1), ((\acute{\sigma}, 2), 3)\}\\ F &= \{2\}\end{aligned}$$

★★ 4. In the "Defining Connectives with Connectives" exercise block on Page 17 there was a bonus exercise asking you to define all of the basic logical connectives using only NAND ($\dagger$). If you have not done that exercise, do so now. Once completed, use the following truth table for NOR ($\downarrow$) to define all of the same logical connectives using only NOR.

P	Q	$P \downarrow Q$
0	0	1
0	1	0
1	0	0
1	1	0

★★★ 5. We have been discussing two formal languages throughout our notes: $a^n b$ and $a^n b^n$. In general, for any string w, w^n refers to n repetitions of w. In describing formal languages, it is common to use two other superscripted symbols: w^* refers to string w repeated *0 or more times*; w^+ refers to string w repeated *1 or more time*.

- Suppose we wanted to describe the language that contains all and only the strings containing one or more a. Well, we could do so with a^+. Instead, define this formal language without using $+$ (but with using any of the other tools introduced above).

In the previous lesson, we discussed union and intersection in relation to sets. One additional property is set complement which is everything in the sets domain that is not in a given set. We can also talk about the union, intersection, and complement in relation to the stringsets (languages) accepted by finite state acceptors. This requires us to have a *complete* machine which corresponds to it being a *total function*. What this means is that for each state q in the machine, and each symbol σ in our alphabet Σ, $\delta(\sigma, q)$ is defined. Before explaining how to define the intersection, union, and complement of two acceptors, draw two complete machines for the following two formal languages with the alphabet $\Sigma = \{a, b, c\}$:

- $A = (ab)^+cb(ab)^+$
- $B = (ab)^*$

Suppose that the first language is represented as $A = \langle Q_A, q_{0A}, \delta_A, F_A\rangle$ and the second language is represented as $B = \langle Q_B, q_{0B}, \delta_B, F_B\rangle$. We can construct machines that will generate/recognize their union, intersection, and complement using the following definitions.

- Construct an acceptor $C = \langle Q, q_o, \delta, F\rangle$ which will recognize the union of A and B as follows:

  $Q = Q_A \times Q_B$

  $q_0 = (q_{0A}, q_{0B})$

  $\delta((q_a, q_b), a) = (q'_a, q'_b)$ where $\delta_A(q_a, a) = q'_a$ and $\delta_B(q_b, a) = q'_b$

  $F = \{(q_a, q_b) \mid q_a \in F_A \text{ or } q_b \in F_B\}$

- Construct an acceptor $C = \langle Q, q_o, \delta, F\rangle$ which will recognize the intersection of A and B as follows:

  $Q = Q_A \times Q_B$

  $q_0 = (q_{0A}, q_{0B})$

  $\delta((q_a, q_b), a) = (q'_a, q'_b)$ where $\delta_A(q_a, a) = q'_a$ and $\delta_B(q_b, a) = q'_b$

  $F = \{(q_a, q_b) \mid q_a \in F_A \text{ and } q_b \in F_B\}$

- Construct an acceptor $C = \langle Q, q_o, \delta, F\rangle$ which will recognize the complement of A as follows:

  $Q = Q_A$

  $q_0 = q_{0A}$

  $\delta = \delta_A$

  $F = \{q_a \mid q_a \notin F_A\}$

- Draw the machines for $A \cup B$, $A \cap B$, and $\overline{A}$.

★★★ 6. One point of debate within generative theories of phonological mappings is whether or not the different processes that occur in a given language happen in serial or parallel. For example, ordered rules are applied serially while OT grammars perform the entire mapping in parallel. We can think of the serial/parallel distinction in the same way with Finite State Transducers. Suppose we have the following two phonological rules:

(a) $a \rightarrow b / c\_\_d$

(b) $d \rightarrow e / b\_\_\#$

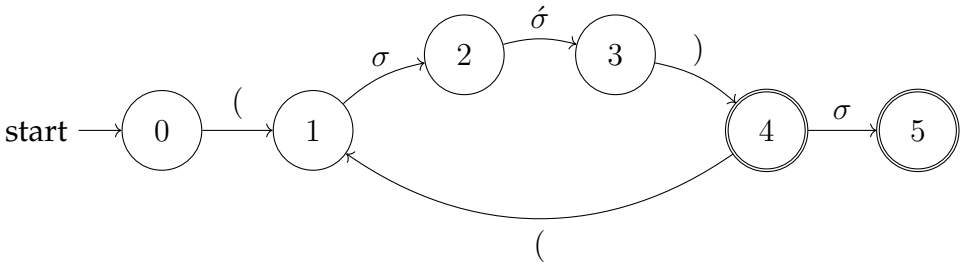
Given an alphabet $\Sigma = \{a, b, c, d, e\}$, draw two transducers (one for each process). If we wanted to order these processes, we could have the output of one transducer be the input to the other. This corresponds to the serial view of phonological knowledge and supports the psychological reality of intermediate forms. Suppose instead we wanted to draw a single machine that would be *extensionally equivalent* to ordering rule (a) before rule (b). Draw this machine now.

The corresponding machine has a single input/output pair and corresponds with the parallel view of phonological knowledge. Here there is no support for intermediate forms because intermediate forms do not exist in our single machine. If you are struggling drawing any of the three transducers from this exercise, the following reminders may help: 1) in the definition for finite state transducers, the input must be a single symbol but the output can be a string of any length (including the empty string); 2) the output function may be useful when thinking about processes that happen at the end of a word.

Lesson 3

★ 1. Recall that we denoted the set of all strings enclosed by boundary markers $\rtimes, \ltimes$ in the following way: $\{\rtimes w \ltimes \mid w \in \Sigma^*\}$. Explain the difference between this set and the set $\Sigma^* \cup \{\rtimes, \ltimes\}$.

★ 2. Observe the following FSA which is defined over the alphabet $\Sigma = \{\sigma, \acute{\sigma}, (,)\}$:



- Describe the language accepted by this automaton.
- Write out the automaton's transition relation and explain why it is a function. Is it a total function? If it isn't, which transitions could be added to make it total?

★★ 3. Regarding the subregular hierarchy, so far we've seen some examples of Strictly Local languages and Strictly Piecewise Languages.

Recall that Strictly Local languages are defined in terms of which substrings of a fixed length k are permissible in a given word, and analagously Strictly Piecewise languages are defined in terms of which subsequences of a fixed length k are permissible in a given word.

Suppose we have an alphabet $\Sigma = \{a, b, c\}$ and a language called `Has-B` which contains any and all strings that contain a b. Here are some randomly chosen examples of strings in this language:

$$b, cb, abc, acccccccccb, acacacacacacaccab, \ldots$$

Is this language a Strictly Local language? In other words, is there some uniform window of size k where it will be possible to tell whether a string of arbitrary length will contain a b? Explain your reasoning.

Bonus: Thinking from a logical perspective as opposed to an automata-theoretic one, does it become clearer which family `Has-B` is a member of? *Hint: Quantification.*

★★ 4. Write a context free grammar that is able to generate the sentences "I shot an elephant in my pajamas". Keep in mind that this sentence has two available readings, which should be reflected in the definition of the CFG. *Hint: where is the PP?*

★★★ 5. A **monoid** $(S, \cdot, e)$ is an algebraic object consisting of a carrier set S equipped with a binary operation $\cdot(x, y) = z$ where $x, y, z \in S$, and an element e, that follows the following properties:

- Associativity: $\forall x, y \in S, (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Identity element e: $\forall x \in S, e \cdot x = x \cdot e = x$

For example, the following are monoids:

$(\mathbb{N}, +, 0)$ where the set $\mathbb{N}$ is the natural numbers and $+$ is ordinary addition.

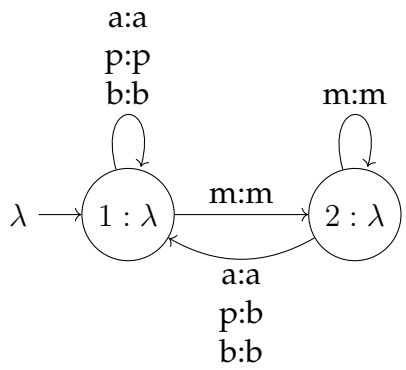
$(\mathbb{R}, \times, 1)$ where the set $\mathbb{R}$ is the real numbers and $\times$ is ordinary multiplication.

$(\{T, F\}, \wedge, T)$ where T, F are true and false and $\wedge$ is ordinary conjunction.

So far we have only seen string-to-string transducers, but transducers can be generalized by incorporating monoids. We will not make the defininitions explicitly, but below we will explore what other sorts of mappings these transducers can accomplish with some minor tweaks.

In string-to-string transducers, we are using the monoid where the carrier set is the of all strings Σ^*, the binary operation is string concatenation $\cdot(x, y)$, and the identity element is the empty λ.

Recall the graphical representation of the string transducer from Page 29 that turns voiceless segments immediately following nasal segments into their voiced counterpart.



Consider a constraint *mp that penalizes substrings of "mp" in a given string.

- Change the transducer above so that it counts violations *mp. Which monoid could be used to do this?
- Change the transducer above so that it assigns a lower real-valued number to strings with more violations of *mp. Which monoid could be used to do this?
- Show how this works on both transducers with a string of your choice as an example.

Hint: What does the transducer output at each step?

★★★ 6. Recall the definition of a Rewrite Grammars and the restriction that gives rise specifically to Context-Free Grammars, repeated below:

A **Rewrite Grammar** or **Phrase Structure Grammar** is a tuple $\langle T, N, S, R\rangle$ where:

- T is a set of terminal symbols
- N is a set of non-terminal symbols
- S is a non-terminal starting symbol
- R is a set of rewrite rules where $A \to B$ and $A, B \in (N \cup T)^*$

Specifically, a **Context-Free Grammar** is a Phrase Structure Grammar such that for every rewrite rule $A \to B \in R$, it is the case that $A \in N$ is a single non-terminal and $B \in (N \cup T)^*$ is a sequence of non-terminals or a single terminal.

Consider the following Rewrite Grammar:

$$S \to aTb \mid abc$$
$$aT \to aaTb \mid ac$$

Notice that this is not a Context-Free Grammar since there is a rewrite rule $A \to B$ that contains an A that is not single non-terminal. Below, there are three examples of derivations for strings that are in the language:

- $S \to abc$

- $S \to aTb$
  $\to aaTbb$
  $\to aacbb$
- $S \to aTb$
  $\to aaTbb$
  $\to aaaTbbb$
  $\to aaacbbb$

Describe the language that is generated by this Rewrite Grammar.

Lesson 4

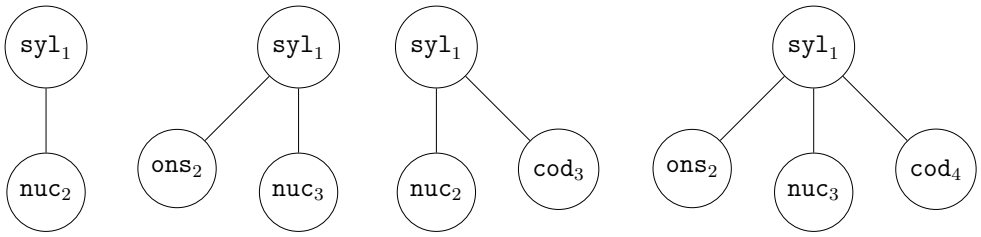
★ 1. Recall the syntactic relation **c-command** from the exercises in Lesson 1. This relation holds true of two nodes x, y in a tree if and only iff neither x nor y dominate eachother and every node that dominates x also dominates y.

In Lesson 4 we saw that tree models can be defined using general dominance $\lhd^*(x, y)$ and selection $\prec\ (x, y)$. Define a predicate $\texttt{c-com}(x, y)$ that is true when a node x c-commands a node y in a tree using First-Order logic.

★ 2. Suppose we had the following model signature for describing syllables:

$$\langle \texttt{syl}(x), \texttt{ons}(x), \texttt{nuc}(x), \texttt{cod}(x), \mathfrak{R}(x, y) \rangle$$

This allows us to build Σ-structures such as the following:



- Write out the mathematical definitions for each of these Σ-structures (assume if two domain elements are connected, they satisfy $\mathfrak{R}(x, y)$).
- Notice that all of these structures have a nucleus (nuc). What would a Σ-sentence look like that only structures with a nucleus would satisfy?
- **Bonus**: There is no mechanism currently that will enforce that a coda doesn't come before a nucleus (in other words they are unordered). How could the model signature be altered to account for ordering within the syllable?

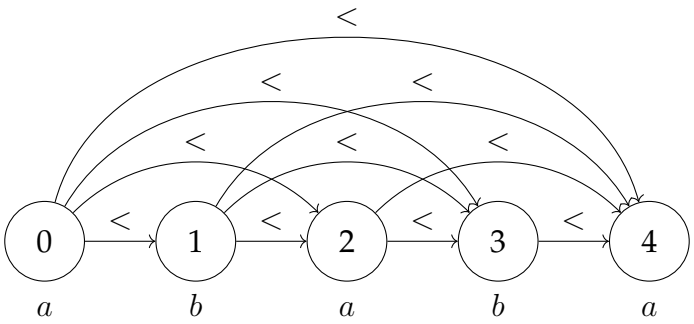
★★ 3. Let $\Sigma = \{L, H\}$ represent an alphabet of high toned and low toned syllables. Using CNL, write a grammar for a language that bans an H tone immediately followed by an L tone in word-final positions. You can (but need not!) use boundary symbols

$\rtimes, \ltimes$. Show the graphical representation of the model of a well-formed and an ill-formed word in the language.

★★ 4. In going from Propositional to First-Order Logic, adding quantification grants the ability to define strict precedence $\lhd(x, y)$ in terms of general precedence $< (x, y)$. In short, a node x strictly precedes a node y iff x generally precedes y and there's no intervening z between them. More formally,

$$\lhd(x, y) :=< (x, y) \wedge \neg\exists z[x < z < y]$$

Consider the following graphical representation of a string model for the string $ababa$, defined using general precedence $< (x, y)$.



- This definition works one way, but can general precedence $< (x, y)$ be defined in terms of strict precedence $\lhd(x, y)$? Why or why not?
- Define a relativized precedence called a-precedence $<^{\{a\}} (x, y)$ that applies to only nodes bearing a, and immediate a-precedence $\lhd^{\{a\}}(x, y)$.
- Use immediate a-precedence to write a formula describing when a domain element is in between two a's on the "a-tier".
- Now, write a formula using strict precedence (i.e. successor) to write a formula describing when a domain element is in between two $b's$ generally.
- Suppose we extend out alphabet to $\{a, b, c\}$ and imagine a process that turns an a that is in between two a's on the "a-tier" into a c, unless it is immediately between two b's. Thus, $ababa \mapsto ababa$ and $abababa \mapsto abababa$, but $abaca \mapsto abcca$ and $ababaca \mapsto ababcca$. Use your formula from above to write a single formula describing the precise conditions that lead to a node surfacing as a c in the output (don't forget to consider segments that were already c's in the input structure).

★★★ 5. Now that you have had experience thinking about intervocalic voicing in terms of well-formed strings, we want you to shift gears and think about it in terms of a *process*. This would look like a rule $T \rightarrow D \ / \ V\_V$. Furthermore, we want you to describe this process model-theoretically as an interpretation from one structure to another structure. Use the set $\{V, T, D\}$ as both the input and output alphabets. Use the following steps:

- First, define an interpretation such that the model signatures contain the ordering functions $\mathtt{s}(x)$ and $\mathtt{p}(x)$.
- Second, define an interpretation such that the model signatures contain the ordering relation $\lhd(x, y)$.
- Once you've have defined both your interpretations, describe what symbols of First-Order Logic are included/not-included in the specific sets of formulas in each part (hint: consult Table 4.1). Does the inclusion of certain symbols within one interpretation suggest it belongs to a more expressive class than the other? Think about this in terms of what other types of properties you could describe about structures with specific logic symbols.

★★★ 6. Interpretations define an output structure in terms of input properties, but sometimes it seems that we might want to define things in terms of output properties. Consider a phonological process such as iterative regressive voice assimilation. This is a process whereby the voicing feature of the final segment spreads leftward throughout the string, ensuring all segments to its left share the same value for voicing (this is somewhat of a simplification). Using the alphabet $\Sigma = \{V, N, T, D\}$ where V stands for all vowels, N stands for all sonorant consonants, T stands for voiceless obstruents, and D stands for voiced obstruents.

- Try to write a formula $\phi_{\mathtt{voice}}(x)$ for describing the voicing properties of output segments. In our input structure, assume that $\mathtt{voice}(V) = \mathtt{voice}(N) = \mathtt{voice}(D) = \mathtt{true}$ and $\mathtt{voice}(T) = \mathtt{false}$. One possibility is to determine the voicing of non-final elements based on the voicing property of its successor in the *output*. This would require a recursive definition such that you are using the output function you are defining in the denotation itself. In your attempt to do so, you should run into a problem. Describe what this problem is.

One way to solve this is to define our interpretations using a different style of formal machinery (but ultimately is quite similar to mathematical logic). Boolean Monadic Recursive Schemes (BMRS) are an available formalism that was developed to explain patterns like the one above. We will not get into the full details here, but interested parties can see Bhaskar et al. (2020) and Chandlee and Jardine (2021) for more details. The important aspect for our purposes is the use of IF…THEN…ELSE syntax. With just this simple syntax, we can capture all of the logical formula we already were using. In general, a BMRS function ϕ evaluates to $\top$ meaning "true" or $\bot$ meaning "false". This is done by writing statements in each block of the IF…THEN…ELSE statements. As an example:

$$\phi_{\mathtt{voice}}(x) = \mathtt{IF}\ \mathtt{stop}(x)\ \mathtt{THEN}\ \bot\ \mathtt{ELSE}\ \mathtt{voice}(\mathtt{s}(x))$$

would be a statement saying domain element x has the property of being $[-\mathtt{voice}]$ in the output structure if it is a stop in the input structure, otherwise its value for voicing depends on its successor in the input structure. The only restriction is that the function should return a boolean value. Therefore, you should be careful of the *types* as you fill in the subparts of the IF…THEN…ELSE statements. Before returning to the

linguistics problem at hand, the following problem will be useful for understanding how the BMRS syntax works:

- Assume we just have general predicates P and Q and want to write BMRS functions that capture various logical connectives. Write functions that capture $P \wedge Q$, $P \vee Q$, $\neg P$, and $P \rightarrow Q$.

Now that you have had some practice with BMRS syntax, let's go back to our original problem. Can we write an interpretation using BMRS that will correctly define the output voicing properties of segments and also converge? Remember that you will want to write a *recursive* function and therefore using $\phi_{\texttt{voice}}$ in its own definition.

If you successfully wrote the function for general voicing assimilation you can attempt the following more complicated variations:

- Change your function above so that only obstruents (T, D) act as triggers and targets. That is, the voicing assimilation process should only occur if the final consonant is T or D and should only continue until it reaches a sonorant consonant N or vowel V which act as *blockers*.
- Change your function above so that only obstruents (T, D) act as triggers but everything is a target. That is, the voicing assimilation process should only occur if the final consonant is T or D but will continue all the way to the beginning of the string.
- Change your function above so that only obstruents act as targets but everything is a trigger. That is, the voicing assimilation process should only occur if there is an obstruent in the *second to last* position in the string and spread leftward for as many obstruents it has to its left.

Bibliography

Andersson, S., Dolatian, H., and Hao, Y. (2019). Computing vowel harmony: The generative capacity of search & copy. In Proceedings of the Annual Meetings on Phonology.

Balakrishnan, V. K. (2012). Introductory discrete mathematics. Courier Corporation.

Bhaskar, S., Chandlee, J., Jardine, A., and Oakden, C. (2020). Boolean monadic recursive schemes as a logical characterization of the subsequential functions. In Language and Automata Theory and Applications: 14th International Conference, LATA 2020, Milan, Italy, March 4–6, 2020, Proceedings 14, pages 157–169. Springer.

Burness, P. A., McMullin, K. J., Chandlee, J., Burness, P., and McMullin, K. (2021). Long-distance phonological processes as tier-based strictly local functions. Glossa: a journal of general linguistics, 6(1).

Chandlee, J. (2017). Computational locality in morphological maps. Morphology, 27(4):599–641.

Chandlee, J. and Heinz, J. (2017). Computational phonology. In Oxford Research Encyclopedia of Linguistics.

Chandlee, J. and Jardine, A. (2021). Computational universals in linguistic theory: Using recursive programs for phonological analysis. Language, 97(3):485–519.

Church, A. (1932). A set of postulates for the foundation of logic. Annals of mathematics, pages 346–366.

Church, A. (1933). A set of postulates for the foundation of logic. Annals of mathematics, pages 839–864.

Daciuk, J., Mihov, S., Watson, B. W., and Watson, R. E. (2000). Incremental construction of minimal acyclic finite-state automata. Computational linguistics, 26(1):3–16.

De Santo, A. and Graf, T. (2019). Structure sensitive tier projection: Applications and formal properties. In Formal Grammar: 24th International Conference, FG 2019, Riga, Latvia, August 11, 2019, Proceedings 24, pages 35–50. Springer.

De Santo, A. and Rawski, J. (2020). What can formal language theory do for animal cognition studies? Royal Society open science, 7(2):191772.

De Santo, A. and Rawski, J. (2022). Mathematical linguistics and cognitive complexity. In Handbook of Cognitive Mathematics, pages 1–38. Springer.

Dinnsen, D. and Garcia-Zamor, M. (1971). The three degrees of vowel length in german. Research on Language & Social Interaction, 4(1):111–126.

Dolatian, H. and Heinz, J. (2019). Learning reduplication with 2-way finite-state transducers. In International Conference on Grammatical Inference, pages 67–80. PMLR.

Dolatian, H. and Heinz, J. (2020). Computing and classifying reduplication with 2-way finite-state transducers. Journal of Language Modelling, 8(1):179–250.

Eccles, P. J. (2013). An Introduction to Mathematical Reasoning: numbers, sets and functions. Cambridge University Press.

Graf, T. (2012). Movement-generalized minimalist grammars. In Logical Aspects of Computational Linguistics: 7th International Conference, LACL 2012, Nantes, France, July 2-4, 2012. Proceedings 7, pages 58–73. Springer.

Graf, T. (2022a). Diving deeper into subregular syntax. Theoretical Linguistics, 48(3-4):245–278.

Graf, T. (2022b). Subregular linguistics: bridging theoretical linguistics and formal grammar. Theoretical Linguistics, 48(3-4):145–184.

Graf, T. (2022c). Typological implications of tier-based strictly local movement. In Proceedings of the Society for Computation in Linguistics 2022, pages 184–193.

Graf, T. and Mayer, C. (2018). Sanskrit n-retroflexion is input-output tier-based strictly local. In Proceedings of the fifteenth workshop on computational research in phonetics, phonology, and morphology, pages 151–160.

Hagerup, A. (2011). A phonological analysis of vowel allophony in west greenlandic. Master's thesis, Norges teknisk-naturvitenskapelige universitet, Det humanistiske fakultet .

Hanson, K. (2023). A tsl analysis of japanese case. Proceedings of the Society for Computation in Linguistics, 6(1):15–24.

Heinz, J. (2018). The computational nature of phonological generalizations. In Hyman, L. and Plank, F., editors, Phonological Typology, Phonetics and Phonology, chapter 5, pages 126–195. De Gruyter Mouton.

Heinz, J., Rawal, C., and Tanner, H. G. (2011). Tier-based strictly local constraints for phonology. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics, pages 58–64.

Hodges, W. (2018). A short history of model theory. In Button, T. and Walsh, S., editors, Philosophy and Model Theory, chapter 18, pages 439–505. Oxford University Press, Oxford.

Hopcroft, J. E. and Ullman, J. D. (1969). Formal languages and their relation to automata. Addison-Wesley Longman Publishing Co., Inc.

Howe, P. (2021). Central malagasy. Journal of the International Phonetic Association, 51(1):103–136.

Hunter, T. (2019). What sort of cognitive hypothesis is a derivational theory of grammar? Catalan journal of linguistics, pages 89–138.

Isac, D. and Reiss, C. (2013). I-language: An introduction to linguistics as cognitive science. Oxford University Press, USA.

Jäger, G. and Rogers, J. (2012). Formal language theory: refining the chomsky hierarchy. Philosophical Transactions of the Royal Society B: Biological Sciences, 367(1598):1956–1970.

Jardine, A. (2017). The local nature of tone-association patterns. Phonology, 34(2):363384.

Jardine, A. (2019). The expressivity of autosegmental grammars. Journal of Logic, Language and Information, 28:9–54.

Jardine, A., Danis, N., and Iacoponi, L. (2021). A formal investigation of q-theory in comparison to autosegmental representations. Linguistic Inquiry, 52(2):333–358.

Jurgec, P. (2011). Feature spreading 2.0: A unified theory of assimilation.

Keenan, E. L. and Moss, L. S. (2009). Mathematical Structures in Language. CSLI.

Kobele, G. M. (2006). Generating copies: An investigation into structural identity in language and grammar. PhD thesis, University of California, Los Angeles.

Kobele, G. M., Retoré, C., and Salvati, S. (2007). An automata-theoretic approach to minimalism. Model theoretic syntax at, 10:71–80.

Kornai, A. (2007). Mathematical linguistics. Springer Science.

Kracht, M. (2003). The mathematics of language, volume 63. Walter de Gruyter.

Lambert, D., Rawski, J., and Heinz, J. (2021). Typology emerges from simplicity in representations and learning. Journal of Language Modelling, 9.

Lambert, D. J. (2022). Unifying Classification Schemes for Languages and Processes With Attention to Locality and Relativizations Thereof. PhD thesis, State University of New York at Stony Brook.

Michaelis, J. (2001). Derivational minimalism is mildly context–sensitive. In Logical Aspects of Computational Linguistics: Third International Conference, LACL98 Grenoble, France, December 14–16, 1998 Selected Papers 3, pages 179–198. Springer.

Moll, R. N., Arbib, M. A., and Kfoury, A. (1988). An introduction to formal language theory.

Montague, R. (1974). Formal Philosophy. Yale University Press.

Nelson, S. (2022). A model theoretic perspective on phonological feature systems. Proceedings of the Society for Computation in Linguistics, 5(1):1–10.

Oakden, C. (2020). Notational equivalence in tonal geometry. Phonology, 37(2):257–296.

Oakden, C. D. (2021). Modeling phonological interactions using recursive schemes. PhD thesis, Rutgers The State University of New Jersey, School of Graduate Studies.

Partee, B. B., ter Meulen, A. G., and Wall, R. (1993). Mathematical methods in linguistics. Springer.

Potts, C. and Pullum, G. K. (2002). Model theory and the content of ot constraints. Phonology, 19(3):361–393.

Rawski, J. (2021). Structure and Learning in Natural Language. PhD thesis, State University of New York at Stony Brook.

Riad, T. (2014). The phonology of Swedish. Phonology of the World's Langu.

Rogers, J. (1996). A model-theoretic framework for theories of syntax. arXiv preprint cmp-lg/9604023.

Rogers, J. (1998). A descriptive approach to language-theoretic complexity. Citeseer.

Rogers, J., Heinz, J., Fero, M., Hurst, J., Lambert, D., and Wibel, S. (2013). Cognitive and sub-regular complexity. In Formal Grammar: 17th and 18th International Conferences, FG 2012, Opole, Poland, August 2012, Revised Selected Papers, FG 2013, Düsseldorf, Germany, August 2013. Proceedings, pages 90–108. Springer.

Rogers, J. and Pullum, G. K. (2011). Aural pattern recognition experiments and the subregular hierarchy. Journal of Logic, Language and Information, 20:329–342.

Salvati, S. (2011). Minimalist grammars in the light of logic. In Logic and Grammar: Essays Dedicated to Alain Lecomte on the Occasion of his 60th Birthday, pages 81–117. Springer.

Sipser, M. (1996). Introduction to the theory of computation. ACM Sigact News, 27(1):27–29.

Stabler, E. (1997). Derivational minimalism, page 6895. Springer Berlin Heidelberg.

Stabler, E. P. (2019). Three mathematical foundations for syntax. Annual Review of Linguistics, 5:243–260.

Strother-Garcia, K. (2019). Using model theory in phonology: A novel characterization of syllable structure and syllabification. PhD thesis.

Tarski, A. (1954). Contributions to the theory of models. i. In Indagationes Mathematicae (Proceedings), volume 57, pages 572–581. Elsevier BV.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. Journal of Math, 58(345-363):5.

van Rooij, I. and Baggio, G. (2021). Theory before the test: How to build high-verisimilitude explanatory theories in psychological science. Perspectives on Psychological Science, 16(4):682–697.

Velleman, D. J. (2019). How to prove it: A structured approach. Cambridge University Press.

Vu, M. H., De Santo, A., and Dolatian, H. (2022). Logical transductions for the typology of ditransitive prosody. In Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology, pages 29–38.

Vu, M. H., Shafiei, N., and Graf, T. (2019). Case assignment in tsl syntax: A case study. In Proceedings of the Society for Computation in Linguistics (SCiL) 2019, pages 267–276.

Wang, Y. and Hunter, T. (2023). On regular copying languages. Journal of Language Modelling, 11(1):1–66.