Theoretical Computational Linguistics: Finite-state Automata

Jeffrey Heinz

February 20, 2023

Contents

1 Introduction **5**

1.1 Computational Linguistics: Course Overview 5

1.1.1 Linguistic Theory 5

1.1.2 Automata Theory 6

2 Formal Language Theory **9**

2.1 Formal Languages 9

2.2 Grammars 10

2.3 Expression Grammars 11

2.3.1 Cat-Union Expressions 12

2.3.2 Generalized Regular Expressions 12

2.3.3 Star Free Expressions 13

2.3.4 Piecewise Local Expressions 13

2.4 Rewrite Grammars 14

2.5 The Chomsky Hierarchy 16

2.6 First Order and Monadic Second Order Logic 17

2.6.1 Syntax of FO logic 18

2.6.2 Semantics of FO logic 19

2.6.3 Syntax of MSO logic 21

2.6.4 Semantics of MSO logic 21

2.6.5 Theorems 22

2.6.6 Other Logics 22

3 Strings and Trees **25**

3.1 Strings 25

3.2 Trees 27

3.2.1 String Exercises 29

3.2.2 Tree Exercises 30

Chapter 1

Introduction

1.1 Computational Linguistics: Course Overview

In this class, we will study:

1. Formal Language Theory
2. Automata Theory
3. Haskell
4. ... as they pertain to problems in linguistics:
   (a) *Well-formedness* of linguistic representations
   (b) *Transformations* from one representation to another

1.1.1 Linguistic Theory

Linguistic theory often distinguishes between *well- and ill-formed representations.*

Strings. In English, we can coin new words like *bling.* What about the following?

1. *gding*
2. *θwɪk*
3. *spɪf*

Trees. In English, we interpret the compound *deer-resistant* as an adjective, not a noun. What about the following?

1. *green-house*
2. *dry-clean*
3. *over-throw*

Linguistic theory is often also concerned with *transformations.*

Strings. In generative phonology, underlying representations of words are *transformed* to surface representations of words.

1. /kæt-z/ → [kæts]
2. /wɪʃ-z/ → [wɪʃɪz]

Trees. In derivational theories of generative syntax, the deep sentence structure is *transformed* into a surface structure.

1. Mary won the competition.
   (a) The competition was won by Mary.
   (b) What did Mary win?

1.1.2 Automata Theory

Automata are *abstract* machines that answer questions like these.

The Membership Problem

Given: A possibly infinite set of strings (or trees) X.

Input: A input string (or tree) x.

Problem: Does x belong to X?

The Transformation Problem

Given: A possible infinite function of strings to strings (or trees to trees) $f : X \to Y$.

Input: A input string (or tree) x.

Problem: What is $f(x)$?

There are many kinds of automata. Two common types of automata address these specific problems.

Recognizers Recognizers solve the membership problem.

Transducers Transducers solve the transformation problem.

Different kinds of automata instantiate different kinds of memory.

Finite-state Automata An automata is finite-state whenever the amount of memory necessary to solve a problem for input x is fixed and **independent** of the size of x.

Linear-bounded Automata An automata is linear-bounded whenever the amount of memory necessary to solve a problem for input x is **bounded by a linear function** of the size of x.

In this class we will study finite-state recognizers and transducers. There are many types of these as well, some are shown below.

- deterministic vs. non-deterministic
- 1way vs. 2way (for strings)
- bottom-up vs. top-down vs. walking (for trees)

The simplest type is the deterministic, 1way recognizer for strings. We will start there and then complicate them bit by bit:

1. add non-determinism
2. add output (transducers)
3. add 2way-ness
4. generalize strings to trees and repeat

What do automata mean for linguistic theory?

Fact 1: Finite-state automata over strings are sufficient for phonology and morphology (Johnson, 1972; Kaplan and Kay, 1994; Roark and Sproat, 2007; Dolatian and Heinz, 2020).

Fact 2: Finite-state automata over strings are *NOT* sufficient for syntax, but linear-bounded automata are (Chomsky, 1956; Huybregts, 1984; Shieber, 1985, among others).

Fact 3: Finite-state automata over trees *ARE* sufficient for syntax (Rogers, 1998; Kobele, 2011; Graf, 2011; Stabler, 2019).

Hypothesis: Linguistic phenomena can be modeled with special kinds of finite-state automata with even stricter memory requirements over the right representations (Heinz, 2018; Graf and De Santo, 2019; Graf, 2022).

Chapter 2

Formal Language Theory

The material in this chapter is covered in much greater detail in a number of textbooks including McNaughton and Papert (1971); Harrison (1978); Hopcroft *et al.* (1979); Davis and Weyuker (1983); Hopcroft *et al.* (2001) and Sipser (1997). Here we will state definitions and theorems, but we will not cover the proofs of the theorems.

We begin with the following question: If we choose to model natural languages with formal languages, what kind of formal languages are they? We have some idea what natural languages are. After all, you are reading this! A satisfactory answer to answer this question however also requires being clear about what a formal language is.

2.1 Formal Languages

A formal language is a set of strings. Strings are sequences of symbols of finite length. The symbol Σ commonly denotes a finite set of symbols. There is a unique string of length zero, which is the empty string. This is commonly denoted with λ or ϵ.

A key operation on strings is *concatenation*. The concatenation of string x with string y is written xy. Concatenation is associative: for all strings x, y, z, it holds that $(xy)z = x(yz)$. The empty string is an identity element for concatenation: for all strings x, $x\lambda = \lambda x = x$. If we concatenate a string x with itself n times we write x^n. For example, $(ab)^3 = ababab$.

We can also concatenate two formal languages X and Y.

$$XY = \{xy : x \in X, y \in Y\}$$

Language concatenation is also associative. The empty string language $\{\lambda\}$ is an identity element for language concatenation: for all languages L, $L\{\lambda\} = \{\lambda\}L = L$. Also, the empty set $\varnothing$ is a zero element for language concatenation: for all languages L, $L\varnothing = \varnothing L = \varnothing$.

If we concatenate a language X with itself n times we write X^n. For example, $XX = X^2$. Finally for any language X, we define X^* as follows.

$$X^* = \{\lambda\} \cup X \cup X^2 \cup X^3 \ldots = \bigcup_{n \geq 0} X^n$$

where X^0 is defined as $\{\lambda\}$. The asterisk (*) is called the Kleene star after Kleene (1956) who introduced it.

It follows that the set of all strings of finite length can be denoted Σ^*. Consequently formal languages can be thought of as subsets of Σ^*. How can we talk about such subsets?

One way is to use set notation and set construction. Example 1 present some examples of formal languages defined in these ways.

Example 1. In this example, assume $\Sigma = \{a, b, c\}$.

1. $\{\lambda, a\}$.
2. $\{\lambda, a, aa\}$.
3. $\{a^n \in \Sigma^* : n \leq 10\}$.
4. $\{a^n \in \Sigma^* : n \geq 0\}$.
5. $\{w \in \Sigma^*\}$.
6. $\{w \in \Sigma^* : w \textit{ contains the string } aa\}$.
7. $\{w \in \Sigma^* : w \textit{ does not contain the string } aa\}$.
8. $\{w \in \Sigma^* : w \textit{ contains a } b \textit{ somewhere after an } a\}$.
9. $\{w \in \Sigma^* : w \textit{ does not contain a } b \textit{ somewhere after an } a\}$.
10. $\{w \in \Sigma^* : w \textit{ contains either the string } aa \textit{ or the string } bb\}$.
11. $\{w \in \Sigma^* : w \textit{ contains both the string } aa \textit{ and the string } bb\}$.
12. $\{w \in \Sigma^* : w \textit{ does not contain the string } bb \textit{ on the } \{b,c\} \textit{ tier}\}$.
13. $\{w \in \Sigma^* : w \textit{ contains an even number of } as\}$.
14. $\{a^n b^n \in \Sigma^* : n \geq 1\}$.
15. $\{a^n b^m \in \Sigma^* : m > n\}$.
16. $\{a^n b^n c^n \in \Sigma^* : n \geq 1\}$.
17. $\{a^n b^m c^\ell \in \Sigma^* : \ell > m > n\}$.
18. $\{w \in \Sigma^* : \textit{the number of } bs \textit{ is the same as the number of } cs \textit{ in } w\}$.
19. $\{w \in \Sigma^* : \textit{the number of } as, bs, \textit{ and } cs \textit{ is the same in } w\}$.
20. $\{a^n \in \Sigma^* : n \textit{ is a prime number}\}$.

2.2 Grammars

There are two important aspects to defining grammar formalisms. They are distinct, but related, aspects.

1. The grammar itself. This is an object and in order to be well-formed it has to follow certain rules and/or conditions.

2. How the grammar is associated with a language. A separate set of rules/conditions explains how to *interpret* the grammar. This aspect explains how the grammar *generates/recognizes/accepts* a language.

In other words, by itself, a grammar is more or less useless. But combined with a way to interpret it—a way to associate it with a formal language—it becomes a very powerful form of expression.

2.3 Expression Grammars

As a first example, consider regular expressions. These consist of both a syntax (which define well-formed regular expressions) and a semantics (which associate them unambiguously with formal languages). They are defined inductively.

Syntax		**Semantics**		
REs include				
• each $\sigma \in \Sigma$	*(singleton letter set)*	$[\![\sigma]\!]$	$=$	$\{\sigma\}$
• ϵ	*(empty string set)*	$[\![\epsilon]\!]$	$=$	$\{\epsilon\}$
• $\varnothing$	*(empty set)*	$[\![\varnothing]\!]$	$=$	$\{\}$
If R, S are REs then so are:				
• (R∘S)	*(concatenation)*	$[\![(R \cdot S)]\!]$	$=$	$[\![R]\!] \circ [\![S]\!]$
• (R+S)	*(union)*	$[\![(R + S)]\!]$	$=$	$[\![R]\!] \cup [\![S]\!]$
• (R^*)	*(Kleene star)*	$[\![(R^*)]\!]$	$=$	$[\![R]\!]^*$

We say a language is *regular* if there is a regular expression denoting it. The class of regular languages is denoted $[\![RE]\!]$.

Exercise 1. Write regular expressions for as many of the languages in Example 1 as you can.

2.3.1 Cat-Union Expressions

Syntax		**Semantics**		
CUEs include				
• each $\sigma \in \Sigma$	*(singleton letter set)*	$[\![\sigma]\!]$	$=$	$\{\sigma\}$
• ϵ	*(empty string set)*	$[\![\epsilon]\!]$	$=$	$\{\epsilon\}$
• $\varnothing$	*(empty set)*	$[\![\varnothing]\!]$	$=$	$\{\}$
If R, S are CUEs then so are:				
• (R∘S)	*(concatenation)*	$[\![(R \cdot S)]\!]$	$=$	$[\![R]\!] \circ [\![S]\!]$
• (R+S)	*(union)*	$[\![(R + S)]\!]$	$=$	$[\![R]\!] \cup [\![S]\!]$

So CUEs are a fragment of REs that exclude Kleene star.

Exercise 2. Write CUEs for as many of the languages in Example 1 as you can. What kinds of formal languages do cat-union expressions describe?

Theorem 1. $[\![CUE]\!] = \{L \subseteq \Sigma^* : |L| \text{ is finite}\} \subsetneq [\![RE]\!]$

2.3.2 Generalized Regular Expressions

Syntax		**Semantics**		
GREs include				
• each $\sigma \in \Sigma$	*(singleton letter set)*	$[\![\sigma]\!]$	$=$	$\{\sigma\}$
• ϵ	*(empty string set)*	$[\![\epsilon]\!]$	$=$	$\{\epsilon\}$
• $\varnothing$	*(empty set)*	$[\![\varnothing]\!]$	$=$	$\{\}$
If R, S are GREs then so are:				
• (R∘S)	*(concatenation)*	$[\![(R \cdot S)]\!]$	$=$	$[\![R]\!] \circ [\![S]\!]$
• (R+S)	*(union)*	$[\![(R + S)]\!]$	$=$	$[\![R]\!] \cup [\![S]\!]$
• (R*)	*(Kleene star)*	$[\![(R^*)]\!]$	$=$	$[\![R]\!]^*$
• (R&S)	*(intersection)*	$[\![(R\&S)]\!]$	$=$	$[\![R]\!] \cap [\![S]\!]$
• ($\overline{\text{R}}$)	*(complement)*	$[\![\overline{\text{R}}]\!]$	$=$	$\Sigma^* - [\![R]\!]$

Theorem 2. $[\![RE]\!] = [\![GRE]\!]$

Exercise 3. Write GREs for as many of the languages in Example 1 as you can.

2.3.3 Star Free Expressions

Syntax		**Semantics**		
SFEs include				
• each $\sigma \in \Sigma$	*(singleton letter set)*	$[\![\sigma]\!]$	$=$	$\{\sigma\}$
• ϵ	*(empty string set)*	$[\![\epsilon]\!]$	$=$	$\{\epsilon\}$
• $\varnothing$	*(empty set)*	$[\![\varnothing]\!]$	$=$	$\{\}$
If R, S are SFEs then so are:				
• (R∘S)	*(concatenation)*	$[\![(R \cdot S)]\!]$	$=$	$[\![R]\!] \circ [\![S]\!]$
• (R+S)	*(union)*	$[\![(R + S)]\!]$	$=$	$[\![R]\!] \cup [\![S]\!]$
• (R&S)	*(intersection)*	$[\![(R\&S)]\!]$	$=$	$[\![R]\!] \cap [\![S]\!]$
• $(\overline{\mathrm{R}})$	*(complement)*	$[\![\overline{\mathrm{R}}]\!]$	$=$	$\Sigma^* - [\![R]\!]$

So SFEs are a fragment of GREs that exclude Kleene star.

Exercise 4. Write SFEs for as many of the languages in Example 1 as you can.

Theorem 3. $[\![CUE]\!] \subsetneq [\![SFE]\!] \subsetneq [\![RE]\!] = [\![GRE]\!]$

For more information on the theorems in this section, see McNaughton and Papert (1971).

2.3.4 Piecewise Local Expressions

Dakotah Lambert developed PLEs over the past ten years. His 2022 dissertation provides a written treatment. I present a large fragment of them here (some more details are in the thesis). Part of the motivation for PLEs is to develop linguistically motivated expression-builders.

As an example, Lambert introduces a tier operator which takes two arguments: a set of symbols T (the tier elements) and a language L. Non-tier elements are freely insertable and deleteable (they have no effect on whether a string belongs to the language or not). Removing the non-tier symbols from a word yields a string of symbols on the tier. Given a language L, let us call the language obtained from removing the non-tier symbols from all of its words, the tier-projection of L. Then Lambert's tier operator produces the largest language in Σ^* such that its tier-projection equals the tier-projection of L. Lambert's operator is thus the *maximal, inverse* tier-projection.

Formally, for all $\sigma \in \Sigma$ and all $T \subseteq \Sigma$, let $I_T(\sigma)$ denote the string σ iff $\sigma \in T$ and λ otherwise. Then, for all $w = \sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$, we let let $[T]w$ be the language $S^* I_T(\sigma_1)S^* I_T(\sigma_2)S^* \ldots S^* I_T(\sigma_n)S^*$ where $S = \Sigma - T$. Finally, for any language L, we let $[T]L = \bigcup_w \in L[T]w$.

Syntax		**Semantics**		
For all $\sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$ PLEs include				
• $\langle\sigma_1\sigma_2\ldots\sigma_n\rangle$	*(unanchored substring)*	$[\![\langle\sigma_1\sigma_2\ldots\sigma_n\rangle]\!]$	=	$\Sigma^*\sigma_1\sigma_2\ldots\sigma_n\Sigma^*$
• $\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle$	*(unanchored subsequence)*	$[\![\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle]\!]$	=	$\Sigma^*\sigma_1\Sigma^*\sigma_2\Sigma^*\ldots\Sigma^*\sigma_n\Sigma^*$
• $\rtimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle$	*(left-anchored substring)*	$[\![\rtimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle]\!]$	=	$\sigma_1\sigma_2\ldots\sigma_n\Sigma^*$
• $\rtimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle$	*(left-anchored subsequence)*	$[\![\rtimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle]\!]$	=	$\sigma_1\Sigma^*\sigma_2\Sigma^*\ldots\Sigma^*\sigma_n\Sigma^*$
• $\ltimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle$	*(right-anchored substring)*	$[\![\ltimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle]\!]$	=	$\Sigma^*\sigma_1\sigma_2\ldots\sigma_n$
• $\ltimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle$	*(right-anchored subsequence)*	$[\![\ltimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle]\!]$	=	$\Sigma^*\sigma_1\Sigma^*\sigma_2\Sigma^*\ldots\Sigma^*\sigma_n$
• $\rtimes\ltimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle$	*(anchored substring)*	$[\![\rtimes\ltimes\langle\sigma_1\sigma_2\ldots\sigma_n\rangle]\!]$	=	$\{\sigma_1\sigma_2\ldots\sigma_n\}$
• $\rtimes\ltimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle$	*(anchored subsequence)*	$[\![\rtimes\ltimes\langle\sigma_1,\sigma_2,\ldots,\sigma_n\rangle]\!]$	=	$\sigma_1\Sigma^*\sigma_2\Sigma^*\ldots\Sigma^*\sigma_n$
If $R_1, R_2, \ldots R_n$ are PLEs then so are:				
• $\neg R_1$	*(complement)*	$[\![\neg R_1]\!]$	=	$\Sigma^* - [\![R_1]\!]$
• $*R_1$	*(Kleene star)*	$[\![*R_1]\!]$	=	$[\![R_1]\!]^*$
• $[\sigma_1,\sigma_2,\ldots\sigma_n]R_1$	*(tier max-inv-projection)*	$[\![[\sigma_1,\sigma_2,\ldots\sigma_n]R_1]\!]$	=	$[\sigma_1,\sigma_2,\ldots\sigma_n][\![R_1]\!]$
• $\wedge\{R_1,R_2,\ldots R_n\}$	*(intersection)*	$[\![\wedge\{R_1,R_2,\ldots R_n\}]\!]$	=	$\bigcap_{1\le i\le n}[\![R_i]\!]$
• $\vee\{R_1,R_2,\ldots R_n\}$	*(union)*	$[\![\vee\{R_1,R_2,\ldots R_n\}]\!]$	=	$\bigcup_{1\le i\le n}[\![R_i]\!]$
• $\circ\{R_1,R_2,\ldots R_n\}$	*(concatenation)*	$[\![\circ\{R_1,R_2,\ldots R_n\}]\!]$	=	$[\![R_1]\!]\circ[\![R_2]\!]\circ\ldots\circ[\![R_n]\!]$

Theorem 4. $[\![CUE]\!] \subsetneq [\![SFE]\!] \subsetneq [\![RE]\!] = [\![GRE]\!] = [\![PLE]\!]$

Exercise 5. Write PLEs for as many of the languages in Example 1 as you can.

2.4 Rewrite Grammars

There are many ways to define grammars which describe formal languages. Another influential approach has been rewrite grammars (Hopcroft *et al.*, 1979).

Definition 1. *A* rewrite grammar[1] *is a tuple* $\langle T, N, S, \mathcal{R}\rangle$ *where*

- $\mathcal{T}$ *is a nonempty finite alphabet of symbols. These symbols are also called the* terminal *symbols, and we usually write them with lowercase letters like* $a, b, c, \ldots$

[1]For a slightly different definition and some more description of rewrite grammars, see Partee *et al.* (1993, chap. 16).

- $\mathcal{N}$ *is a nonempty finite set of* non-terminal *symbols, which are distinct from elements of* $\mathcal{T}$*. These symbols are also called* category *symbols, and we usually write them with uppercase letters like* $A, B, C, \ldots$

- S *is the* start *category, which is an element of* $\mathcal{N}$*.*

- *A finite set of* production rules $\mathcal{R}$*. A production rule has the form*

$$\alpha \rightarrow \beta$$

where α, β *belong to* $(\mathcal{T} \cup \mathcal{N})^*$*. In other words,* α *and* β *are strings of non-terminal and terminal symbols. While* β *may be the empty string we require that* α *include at least one symbol.*

Rewrite grammars are also called *phrase structure grammars*.

Example 2. Consider the following grammar G_1:

- $\mathcal{T} = \{$***john***, ***laughed***, ***and***$\}$;
- $\mathcal{N} = \{$S, VP1, VP2$\}$; and
- $$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \rightarrow \textbf{\textit{john}}\ \text{VP1} \\ \text{VP1} \rightarrow \textbf{\textit{laughed}} \\ \text{VP1} \rightarrow \textbf{\textit{laughed}}\ \text{VP2} \\ \text{VP2} \rightarrow \textbf{\textit{and laughed}} \\ \text{VP2} \rightarrow \textbf{\textit{and laughed}}\ \text{VP2} \end{array} \right\}$$

Example 3. Consider the following grammar G_2:

- $\mathcal{T} = \{$a, b$\}$;
- $\mathcal{N} = \{$S, A, B$\}$; and
- $$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \rightarrow \text{ABS} \\ \text{S} \rightarrow \lambda \\ \text{AB} \rightarrow \text{BA} \\ \text{BA} \rightarrow \text{AB} \\ \text{A} \rightarrow \text{a} \\ \text{B} \rightarrow \text{b} \end{array} \right\}$$

Example 4. Consider the following grammar G_3:

- $\mathcal{T} = \{$a, b$\}$;
- $\mathcal{N} = \{$S$\}$; and

- $$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \rightarrow \text{ba} \\ \text{S} \rightarrow \text{baba} \\ \text{S} \rightarrow \text{bab} \end{array} \right\}$$

The language of a rewrite grammar is defined recursively below.

Definition 2. *The (partial)* derivations *of a rewrite grammar $G = \langle \mathcal{T}, \mathcal{N}, S, \mathcal{R} \rangle$ is written $D(G)$ and is defined recursively as follows.*

1. The base case: *S belongs to $D(G)$.*
2. The recursive case: *For all $\alpha \rightarrow \beta \in \mathcal{R}$ and for all $\gamma_1, \gamma_2 \in (\mathcal{T} \cup \mathcal{N})^*$, if $\gamma_1 \alpha \gamma_2 \in D(G)$ then $\gamma_1 \beta \gamma_2 \in D(G)$.*
3. *Nothing else is in $D(G)$.*

Then the language of the grammar $L(G)$ is defined as

$$L(G) = \{ w \in \mathcal{T}^* : w \in D(G) \}.$$

Exercise 6. How does G_1 generate ***John laughed and laughed and laughed***?

Exercise 7. What language does G_2 generate?

Exercise 8. What language does G_3 generate?

2.5 The Chomsky Hierarchy

"By putting increasingly stringent restrictions on the allowed forms of rules we can establish a series of grammars of decreasing generative power. Many such series are imaginable, but the one which has received the most attention is due to Chomsky and has come to be known as the Chomsky Hierarchy." (Partee *et al.*, 1993, p. 451)

Recall that rules are of the form $\alpha \rightarrow \beta$ with $\alpha, \beta \in (\mathcal{T} \cup \mathcal{N})^*$, with the further restriction that α was not the empty string.

Type 0 There is no further restriction on α or β.

Type 1 Each rule is of the form $\alpha \rightarrow \beta$ where α contains at least one symbol $A \in \mathcal{N}$ and β is not the empty string.

Type 2 Each rule is of the form $A \rightarrow \beta$ where $A \in \mathcal{N}$ and $\beta \in (\mathcal{T} \cup \mathcal{N})^*$.

Type 3 Each rule is of the form $A \rightarrow aB$ or $A \rightarrow a$ where $A, B \in \mathcal{N}$ and $a \in \mathcal{T}$.

There is one exception to the above restrictions for Types 1, 2 and 3. For these types, the production $S \rightarrow \lambda$ is allowed. If this production is included in a grammar then the formal language it describes will include the empty string. Otherwise, it will not.

To this we will add an additional type which we will call finite:

finite Each rule of is of the form $S \rightarrow w$ where $w \in \mathcal{T}^*$.

Each of these types goes by other names.

Type 0	recursively enumerable, computably enumerable
Type 1	context-sensitive
Type 2	context-free
Type 3	regular, right-linear[2]
finite	finite

Table 2.1: Names for classes of formal languages.

These names refer to both the *grammars* and the *languages*. These are different kinds of objects, so it is important to know which one is being referred to in any given context.

Theorem 5 (Chomsky Hierarchy). *1.* $[\![type-3]\!] \subseteq [\![type-2]\!]$ *(Scott and Rabin, 1959).*

2. $[\![type-2]\!] \subseteq [\![type-1]\!]$ *(Bar-Hillel* et al.*, 1961).*

3. $[\![type-1]\!] \subseteq [\![type-0]\!]$ [3]

For details, see, for instance, Davis and Weyuker (1983).

Exercise 9. Write rewrite grammars for as many of the languages in Example 1 as you can. Are they type 1, 2, 3 or finite grammars?

If we choose to model natural languages with formal languages, what kind of formal languages are they?

2.6 First Order and Monadic Second Order Logic

We can also define formal languages with logic, and this section explains one way to do that drawing on mathematical logic and model theory Enderton (1972, 2001); Hedman (2004); Rogers *et al.* (2013); **?**.

[2]Technically, right-linear grammars are defined as those languages where each rule is of the form $A \rightarrow aB$ or $A \rightarrow a$ where $A, B \in \mathcal{N}$ and $a \in \mathcal{T}$. Consequently is not possible for a right linear grammar to define a language which includes the empty string.

[3]This is a diagnolization argument of the kind originally due to Cantor. Rogers (1967) is a good source for this kind of thing.

In what follows, we use the fact that every string $w \in \Sigma^*$ is equal to an indexed sequence of symbols, so $w = \sigma_1 \dots \sigma_n$. The positions in the string w correspond to the set of indices. It is common to call this set the *domain of* w, or w*'s domain*. So for a string of length $n \geq 1$ then its domain is the set $\{1, \dots n\}$. If w is the empty string then its domain is empty.

We begin with First Order (FO) logic and then expand it to Monadic Second Order (MSO) logic.

2.6.1 Syntax of FO logic

We assume a countably infinite set of symbols $x, y \in V_x = \{x_0, x_1, \dots\}$ disjoint from Σ. These symbols will ultimately be interpreted as variables which range over the domains of strings, and we refer to these symbols as variables.

Definition 3 (Formulas of FO logic)**.**

The base cases. *For all variables x, y, and for all $\sigma \in \Sigma$, the following are formulas of FO logic.*

(B1)	$x = y$	***(equality)***
(B2)	$x < y$	***(precedence)***
(B3)	$\sigma(x)$	***(does σ occupy position x?)***

The inductive cases. *If φ, ψ are formulas of FO logic, then so are*

(I1)	$(\neg\varphi)$	***(negation)***
(I2)	$(\varphi \vee \psi)$	***(disjunction)***
(I3)	$(\varphi \wedge \psi)$	***(conjunction)***
(I4)	$(\varphi \rightarrow \psi)$	***(implication)***
(I5)	$(\varphi \leftrightarrow \psi)$	***(biconditional)***
(I6)	$(\exists x)[\varphi]$	***(existential quantification for individuals)***
(I7)	$(\forall x)[\varphi]$	***(universal quantification for individuals)***

Nothing else is a formula of FO logic.

Of course it is possible to define a FO logic with some subset of the above inductive cases and to derive the remainder. For example, negation, disjunction, and existential quantification are sufficient to derive the remainder. We include them all to facilitate writing logical formulas.

Exercise 10. Which of the following expressions are syntactically valid formulas of FO logic? Assume $\Sigma = \{a, b, c\}$.

1. $a(x)$
2. $a(x) \wedge b(y)$

3. $(a(x) \wedge b(y))$
4. $\forall x[a(x)]$
5. $(\forall x)\ a(x)$
6. $(\forall x)\ [a(x)]$
7. $(\forall x)\ [x = a]$
8. $(\forall x, y)\ [x = y]$
9. $(\forall x)[(\forall y)\ [x = y]]$
10. $(\forall x)[(\exists y)[y = x + 1]]$
11. $(\exists x)\big[(a(x) \wedge (\forall y)[(a(y) \rightarrow x = y)])\big]$
12. $\exists x\big[a(x) \wedge (\forall y)[a(y) \rightarrow x = y]\big]$
13. $((\exists x)[a(x)] \wedge (\forall y)[(a(y) \rightarrow x = y)])$

2.6.2 Semantics of FO logic

The *free* variables of a formula φ are those variables in φ that are not quantified. A formula is a *sentence* if *none* of its variables are free. Only sentences can be interpreted.

Exercise 11. Which of the following expressions are sentences of FO logic? Assume $\Sigma = \{a, b, c\}$.

1. $a(x)$
2. $(\forall x)[a(x)]$
3. $(\exists x)\big[(a(x) \wedge (\forall y)[(a(y) \rightarrow x = y)])\big]$
4. $((\exists x)[a(x)] \wedge (\forall y)[(a(y) \rightarrow x = y)])$

It will also be useful to think of the interpretation of a sentence φ as a function that maps strings to the set $\{\texttt{true}, \texttt{false}\}$. How that is done is explained below.

However, there is notation here to consider. We will write $[\![\varphi]\!]$ to denote this function. In other words, for a sentence φ of FO logic and a string w, the expression $[\![\varphi]\!](w)$ will evaluate to true or false.

In the logical tradition, it is more common to write $w \models \varphi$, which is read as both "w satisfies φ" and "w models φ," and which means that $[\![\varphi]\!](w) = \texttt{true}$. If $[\![\varphi]\!](w)$ evaluates to false, one would write $w \not\models \varphi$. Since here I want to explain how $[\![\varphi]\!](w)$ is calculated, I will use this notation here.

In order to evaluate $[\![\varphi]\!](w)$, variables must be assigned values. For this reason, we will actually think of the function $[\![\varphi]\!]$ taking two arguments: one is the string w and one is the *assignment function*. The assignment function $\mathbb{S}$ maps individual variables like x to elements of then domain (positions). You can think of it like a dictionary which maps keys (the variables) to their values (the positions). Formally, $\mathbb{S} : V_x \rightarrow D$. The assignment function $\mathbb{S}$ may be partial, even empty. The empty assignment is denoted $\mathbb{S}_0$.

We evaluate $[\![\varphi]\!](w, \mathbb{S}_0)$. Throughout the evaluation, the assignment function $\mathbb{S}$ gets updated. The notation $\mathbb{S}[x \mapsto e]$ updates the assignment function to add a binding of element e to variable x. Then whether $w \models \varphi$ can be determined inductively by the below definition.

Definition 4 (Interpreting sentences of FO logic)**.**

The base cases. *For all variables x, y, for all $\sigma \in \Sigma$, and for all $w = \sigma_1\sigma_2 \ldots \sigma_n$:*

$$\begin{array}{llll}
\textit{(B1)} & [\![x = y]\!](w, \mathbb{S}) & \leftrightarrow & \mathbb{S}(x) = \mathbb{S}(y) \\
\textit{(B2)} & [\![x < y]\!](w, \mathbb{S}) & \leftrightarrow & \mathbb{S}(x) < \mathbb{S}(y) \\
\textit{(B3)} & [\![\sigma(x)]\!](w, \mathbb{S}) & \leftrightarrow & \sigma_{\mathbb{S}(x)} = \sigma
\end{array}$$

The inductive cases.

$$\begin{array}{llll}
\textit{(I1)} & [\![(\neg\varphi)]\!](w, \mathbb{S}) & \leftrightarrow & \neg [\![\varphi]\!](w, \mathbb{S}) \\
\textit{(I2)} & [\![(\varphi \vee \psi)]\!](w, \mathbb{S}) & \leftrightarrow & [\![\varphi]\!](w, \mathbb{S}) \vee [\![\psi]\!](w, \mathbb{S}) \\
\textit{(I3)} & [\![(\varphi \wedge \psi)]\!](w, \mathbb{S}) & \leftrightarrow & [\![\varphi]\!](w, \mathbb{S}) \wedge [\![\psi]\!](w, \mathbb{S}) \\
\\
\textit{(I4)} & [\![(\varphi \rightarrow \psi)]\!](w, \mathbb{S}) & \leftrightarrow & [\![\varphi]\!](w, \mathbb{S}) \rightarrow [\![\psi]\!](w, \mathbb{S}) \\
\\
\textit{(I5)} & [\![(\varphi \leftrightarrow \psi)]\!](w, \mathbb{S}) & \leftrightarrow & [\![\varphi]\!](w, \mathbb{S}) \leftrightarrow [\![\psi]\!](w, \mathbb{S}) \\
\\
\textit{(I6)} & [\![(\exists x)[\varphi]]\!](w, \mathbb{S}) & \leftrightarrow & (\bigvee_{e \in D} [\![\varphi]\!](w, \mathbb{S}[x \mapsto e])) \\
\textit{(I7)} & [\![(\forall x)[\varphi]]\!](w, \mathbb{S}) & \leftrightarrow & (\bigwedge_{e \in D} [\![\varphi]\!](w, \mathbb{S}[x \mapsto e]))
\end{array}$$

The formal language that a sentence φ denotes is given by

$$[\![\varphi]\!] = \{w \in \Sigma^* : w \models \varphi\} \ ,$$

i.e. all and only those strings w such that $[\![\varphi]\!](w, \mathbb{S}_0) = \texttt{true}$.

Exercise 12. Determine the formal languages of the following logical sentences.

1. $(\forall x)[a(x)]$
2. $(\exists x)[a(x)]$
3. $(\exists x)\big[(a(x) \wedge (\forall y)[(a(y) \rightarrow x = y)])\big]$
4. $(\exists x)[(\exists y)[((a(x) \wedge a(y) \wedge x < y)]]$

Exercise 13.

1. Write FO sentences for the following languages.

(a) All words which begin with a (so $a\Sigma^*$)
(b) All words which end with a (so $\Sigma^* a$)

2. Write FO sentences for as many of the formal languages in Example 1 as you can.

Hint: it will be useful to define logical predicates for the successor relation, and the tier successor relation and to use those.

Next we turn to Monadic Second Order (MSO) logic.

2.6.3 Syntax of MSO logic

Every formula of FO logic is a formula of MSO logic. MSO logic extends FO logic as follows.

In addition to the countably infinite set of symbols $V_x = \{x_0, x_1, \ldots\}$, we assume another countably infinite set of symbols $V_X = \{X_0, X_1, \ldots\}$, disjoint from Σ. These symbols will ultimately be interpreted as variables which range over *subsets* of the domains of strings. We refer to the symbols of V_x as set variables and the elements of V_x as individual variables.

Definition 5 (Formulas of MSO logic).

The base cases. *The base cases are the same as FO logic along with*

$$\textit{(B4)} \quad x \in X \quad \textit{\textbf{(membership)}}$$

The inductive cases. *If φ, ψ are formulas of FO logic, then so are*

$$\textit{(I8)} \quad (\exists X)[\varphi] \quad \textit{\textbf{(existential quantification for sets)}}$$
$$\textit{(I9)} \quad (\forall X)[\varphi] \quad \textit{\textbf{(universal quantification for sets)}}$$

Nothing else is a formula of FO logic.

2.6.4 Semantics of MSO logic

Recall the assignment function $\mathbb{S}$ we used to interpret sentences of FO logic. We also use $\mathbb{S}$ to keep track of the assignments of set variables, and the notation $\mathbb{S}[X \mapsto S]$ updates the assignment function to add a binding of the set of elements S to variable X.

With that in mind, the interpretation of sentences of MSO logic is the same as FO logic along with the following.

Definition 6 (Interpreting sentences of MSO logic).

The base cases.

$$\textit{(B4)} \quad [\![x \in X]\!](w, \mathbb{S}) \quad \leftrightarrow \quad \mathbb{S}(x) \in \mathbb{S}(X)$$

The inductive cases.

$$
\begin{array}{llcl}
\textit{(I8)} & [\![(\exists X)[\varphi]]\!](w,\mathbb{S}) & \leftrightarrow & (\bigvee_{S\subseteq D}[\![\varphi]\!](w,\mathbb{S}[X\mapsto S])\\
\textit{(I9)} & [\![(\forall X)[\varphi]]\!](w,\mathbb{S}) & \leftrightarrow & (\bigwedge_{S\subseteq D}[\![\varphi]\!](w,\mathbb{S}[X\mapsto S])
\end{array}
$$

That's it!

Exercise 14. 1. What language does the following MSO expression describe?

$$
\begin{array}{rl}
(\exists X)[(\exists Y)[& \\
(\forall x)[(\forall y)[& \\
((((& \\
& (x\in X \leftrightarrow (\neg x\in Y))\\
\wedge & (\texttt{first}(x)\rightarrow x\in X))\\
\wedge & (\texttt{last}(x)\rightarrow x\in Y))\\
\wedge & ((x\triangleleft y \wedge x\in X)\rightarrow y\in Y))\\
\wedge & ((y\triangleleft x \wedge y\in Y)\rightarrow x\in X))\\
&]]]]
\end{array}
$$

(Make sure `first` and `last` are defined appropriately.)

2. Write a MSO sentence which denotes the language whose strings are all and only those with an even number of a symbols. Assume $\Sigma=\{a,b\}$.

2.6.5 Theorems

Let $[\![MSO]\!]$ denote the class of formal languages definable with sentences of MSO logic and $[\![FO]\!]$ denote the class of formal languages definable with sentences of FO logic.

Theorem 6 (Büchi, Elgot, and Trakhtenbrot)**.** $[\![MSO]\!]=[\![RE]\!]$.

Theorem 7 (Schutzenberger)**.** $[\![FO]\!]=[\![SFE]\!]$.

Consequently, it follows that $[\![FO]\!]\subsetneq[\![MSO]\!]$.

2.6.6 Other Logics

In the logical languages defined above, we used the precedence ($<$) as a primitive formula. So the MSO and FO languages defined above are often referred to as MSO($<$) and FO($<$).

What if we replace precedence with successor ($\triangleleft$) so that $[\![x\triangleleft y]\!]$ is true iff $y=x+1$ (so y is the next position after x).

Theorem 8 (Thomas 1982)**.** $[\![FO(\triangleleft)]\!]\subsetneq[\![FO(<)]\!]$.

This is because successor is definable from precedence with first order logic but precedence is not first-order definable with successor.

However, precedence is MSO-definable with successor. Consequently we have the following hierarchy.

$$[\![FO(\triangleleft)]\!] \subsetneq [\![F(<)]\!] \subsetneq [\![MSO(\triangleleft)]\!] = [\![MSO(<)]\!]$$

There are many other kinds of logical languages, including quantifier free logic, modal logic, and Boolean Recursive Monadic Schemes. What is especially nice about logic is that it separates the *representational aspects* of the computation from the *computational actions* that operate on those representations, as we can see from the four classes considered above.

Kind of Logic	Representation of Order	
	Successor	Precedence
Monadic Second Order	MSO($\triangleleft$)	MSO($<$)
First Order	FO($\triangleleft$)	FO ($<$)

What are the representational primitives of linguistic structures and what kind of operations act on them? What logic encodes these linguistic representations and operations?

Chapter 3

Strings and Trees

In this chapter, we define strings and trees of finite size inductively.

3.1 Strings

Informally, strings are sequences of symbols.

What are symbols? It is standard to assume a set of symbols called the *alphabet*. The Greek symbol Σ is often used to represent the alphabet but people also use S, A, or anything else. The symbols can be anything: IPA letters, morphemes, words, part-of-speech categories. Σ can be infinite in size, but we will usually consider it to be finite.

There are different ways strings can be defined formally. Here we define them as a recursive data structure. They are defined inductively with a constructer $(\cdot)$, the alphabet $Sigma$ and the base case λ. What is λ? It is the empty string. It is usually written with one of the Greek letters ϵ or λ. It's just a matter of personal preference. The empty string is useful from a mathematical perspective in the same way the number zero is useful. Zero is a special number because for all numbers x it is the case that $0 + x = x + 0 = x$. The empty string serves the same special purpose. It is the unique string with the following special property with respect to concatenation (denoted $\circ$).[1]

$$\text{For all strings } w,\ \lambda \circ w = w \circ \lambda = w \tag{3.1}$$

Definition 7 (Strings)**.**

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

Example 5. Let $\Sigma = \{a, b, c\}$. Then the following are strings.

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

[1]Concatenation will be defined in an exercise below. Our goal would be to ensure the property mentioned holds once concatenation is defined between strings.

2. $a \cdot (a \cdot (a \cdot (\lambda)))$
3. $a \cdot (b \cdot (c \cdot (c \cdot (\lambda))))$

Frankly, writing all the parentheses and "$\cdot$" is cumbersome. So the above examples are much more readable if written as follows.

1. abc
2. aaa
3. $abcc$

Technically, when we write the string abc, we literally mean the following structure: $a \cdot (b \cdot (c \cdot (\lambda)))$.

The above definition provides a unique "derivation" for each string.

Example 6. Leet $\Sigma = \{a, b, c\}$. We claim $w = a \cdot (b \cdot (a \cdot (\lambda)))$ is a string. There is basically one way to show this. First we observe that $a \in \Sigma$ so whether w is a string depends, by the inductive case, on whether $x = b \cdot (a \cdot (\lambda))$ is a string. Next we observe that since $b \in \Sigma$ whether x is a string depends on whether $y = a \cdot (\lambda)$ is a string, again by the inductive case. Once more, since $a \in \Sigma$ whether y is a string depends on whether λ is a string. Finally, by the base case λ is a string and so the dominoes fall: y is a string so x is a string and so w is a string.

This unique derivabality is useful in many ways. For instance, suppose we want to determine the length of a string. Here is how we can do it.

Definition 8 (string length)**.** *The* length *of a string w, written $|w|$, is defined as follows. If $w = \lambda$ then $|w| = 0$. If not, then $w = a \cdot (x)$ where x is some string and $a \in \Sigma$. In this case, $|w| = |x| + 1$.*

Note length is an inductive definition!

Example 7. What is the length of string $w = abcc$? Well, as before we see that $w = a \cdot x$ where $x = bcc$. Thus, $|w| = 1 + |bcc|$. What is the length of bcc? Well, $x = b \cdot y$ where $y = cc$. So now we have $|w| = 1 + (1 + |cc|)$. Since $y = c \cdot z$ where $z = c$ we have $|w| = 1 + (1 + (1 + |c|))$. Since c is the structure $c(\lambda)$, its length will be $1 + |\lambda|$. Finally, by the *base case* we have $|w| = 1 + (1 + (1 + (1 + (0)))) = 4$.

It's interesting to observe how the structure of the computation of length is the *same* structure as the object itself.

$$\begin{array}{cccccccccc} a & \cdot(& b & \cdot(& c & \cdot(& c & \cdot(& \lambda &)))) \\ 1 & +(& 1 & +(& 1 & +(& 1 & +(& 0 &)))) \end{array}$$

We can now define concatenation between strings. First we define ReverseAppend, which takes two strings as arguments and returns a third string.

Definition 9 (reverse append)**.** Reverse append *is a binary operation over strings, which we denote* $\otimes_{\texttt{revapp}}$*. You can also think of it as a function which takes two strings* w_1 *and* w_2 *as arguments and returns another string. Here is the base case. If* $w_1 = \lambda$ *then it returns* w_2*. So we can write* $\lambda \otimes_{\texttt{revapp}} w = w$*. Otherwise, there is* $a \in \Sigma$ *such that* $w_1 = a \cdot (x)$ *for some string* x*. In this case,* reverse append *returns* $x \otimes_{\texttt{revapp}} a \cdot (w_2)$*.*

Exercise 15. Work out what $abc \otimes_{\texttt{revapp}} def$ equals.

Exercise 16. What is $abc \otimes_{\texttt{revapp}} \lambda$? Write a definition for string reversal.

Exercise 17. Define the concatenation of two strings w_1 and w_2 using reverse append and string reversal. Prove this definition satisfies Equation 3.1.

The set of all strings of finite length from some alphabet Σ, including the empty string, is written Σ^*. A *stringset* is a subset of Σ^*.

Stringsets are often called *formal languages.* From a linguistic perspective, is is the study of string well-formedness.

3.2 Trees

Trees are like strings in that they are recursive structures. Informally, trees are structures with a single 'root' node which dominates a sequence of trees.

Formally, trees extend the dimensionality of string structures from 1 to 2. In addition to linear order, the new dimension is dominance.

Unlike strings, we will not posit "empty" trees because every tree has a root.

Like strings, we assume a set of symbols Σ. This is sometimes partitioned into symbols of different types depending on whether the symbols can only occur at the leaves of the trees or whether they can dominate other trees. We don't make such a distinction here.

Definition 10 (Trees)**.** *If* $a \in \Sigma$ *and* w *is a string of trees then* $a[w]$ *is a tree.*

A tree $a[\lambda]$ is called a *leaf.* Note if $w = \lambda$ we typically write $a[\,]$ instead of $a[\lambda]$. Similarly, if $w = t_1 \cdot (t_2 \cdot (\ldots \cdot (t_n \cdot (\lambda)) \ldots))$, we write $a[t_1 t_2 \ldots t_n]$ for readability.

The definition of trees above may appear circular. It appears circular since it defines trees in terms of strings of trees. However, this circularity is an illusion. The definition has a solid recursive base case, as I will now explain. The key to resolving this illusion is to construct the full set of trees in steps. For example, λ is a string (of trees) by the definition of string. With the empty string λ and the finite alphabet Σ we can define a set of trees $\mathrm{T}_0 = \{a[\lambda] \mid a \in \Sigma\}$. T_0 is the set of all logically possible leaves (trees of depth 0). T_0 is a finite alphabet and so T_0^* is a well defined set of strings over this alphabet. For example $w = a[\,] \cdot (b[\,] \cdot (c[\,] \cdot (b[\,] \cdot (\lambda))))$ is a string of trees.

So far, with Σ and λ we built T_0. The definition of strings gives us T_0^*. Now with Σ and T_0^* we can build $\mathrm{T}_1 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_0^*\} \cup \mathrm{T}_0$. T_1 includes T_0 in addition to all trees of depth 1. With T_1, and the definition of strings we have T_1^*. Now with Σ and T_1^* we can build $\mathrm{T}_2 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_1^*\} \cup \mathrm{T}_1$.

More generally, we define $\mathrm{T}_{n+1} = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_n^*\} \cup \mathrm{T}_n$. Finally, let the set of all logically possible trees be denoted with $\Sigma^{\mathrm{T}} = \bigcup_{i \in \mathbb{N}} \mathrm{T}_i$. Figure 3.1 illustrates this construction.

Figure 3.1: The inductive definition of trees.

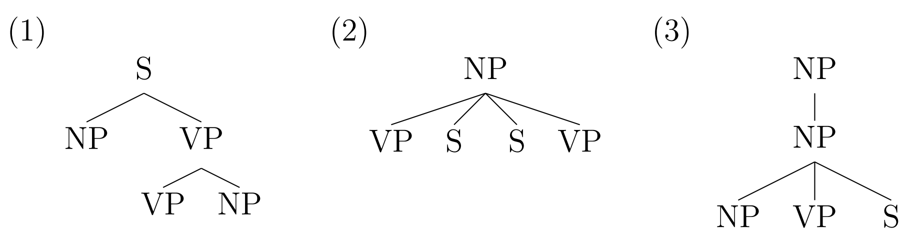
$$\begin{tikzcd}
 & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \\
\lambda \arrow[r] & \mathrm{T}_0 \arrow[r] & \mathrm{T}_0^* \arrow[r] & \mathrm{T}_1 \arrow[r] & \mathrm{T}_1^* \arrow[r] & \mathrm{T}_2 \arrow[r] & \cdots \arrow[r] & \mathrm{T}_n \arrow[r] & \mathrm{T}_n^* \arrow[r] & \mathrm{T}_{n+1}
\end{tikzcd}$$

Here are some examples of trees.

Example 8. Let $\Sigma = \{\text{NP, VP, S}\}$. Then the following are trees.

1. S[NP[ ] VP[VP[ ] NP[ ]]]
2. NP[VP[ ] S[ ] S[ ] VP[ ]]
3. NP[NP[NP[ ] VP[ ] S[ ]]]

We might draw these structures as follows.



Regarding the tree in (1), its leaves are NP, VP, and NP.

As before, we can now write definitions to get information about trees. For instance here is a definition which gives us the number of nodes in the tree.

Definition 11. *The* size *of a tree t, written $|t|$, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its size is* 1*. If not, then $t = a[t_1 t_2 \dots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then $|t| = 1 + |t_1| + |t_2| + \dots + |t_n|$.*

Exercise 18. Using the above definition, calculate the size of the trees (1)-(3) above. Write out the calculation explicitly.

Here is a definition for the *width* of a tree.

Definition 12. *The* depth *of a tree t, written* $\textbf{\texttt{depth}}(t)$*, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its depth is 0. If not, then $t = a[t_1 t_2 \dots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then* $\textbf{\texttt{depth}}(t) = 1 + \texttt{max}\big\{\texttt{depth}(t_1), \texttt{depth}(t_2), \dots, \texttt{depth}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

Definition 13. *The* width *of a tree t, written* $\textbf{\texttt{width}}(t)$*, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its width is 0. If not, then $t = a[t_1 t_2 \dots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then* $\textbf{\texttt{width}}(t) = \texttt{max}\big\{n, \texttt{width}(t_1), \texttt{width}(t_2), \dots, \texttt{width}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

The set of trees Σ^{T} contains all trees of arbitrary width. Much research also effectively concerns the set of all and only those trees whose width is bounded by some number n. Let $\Sigma^{\mathrm{T}(n)} = \{t \in \Sigma^{\mathrm{T}} \mid \texttt{width}(t) \leq n\}$.

Exercise 19. The *yield* of a tree t, written $\texttt{yield}(t)$, maps a tree to a string of its leaves. For example let t be the tree in (1) in Example 8 above. Then its yield is the string "NP VP NP".

3.2.1 String Exercises

Exercise 20. Let Σ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the in the string. So *addOne* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $6 \cdot (12 \cdot (5 \cdot (\lambda)))$. Using this definition, show *addOne* of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) λ

2. Write the definition of the *timesTwo* of the numbers in the string. So *timesTwo* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $10 \cdot (22 \cdot (8 \cdot (\lambda)))$. Using this definition, show *timesTwo* of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) λ

Exercise 21. Let Σ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the *sum* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

(b) $3 \cdot (2 \cdot (\lambda))$

(c) λ

2. Write the definition of the *product* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) λ

3.2.2 Tree Exercises

Exercise 22. Let Σ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the tree. Using this definition, calculate *addOne* as applied to the trees below.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

2. Write the definition of the function *isLeaf* which changes the nodes of a tree to `True` if it is a leaf node or to `False` if it is not. Using this definition, calculate *isLeaf* as applied to the trees below.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

Exercise 23. Let Σ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the *sum* of the numbers in the tree. Using this definition, show how the sum of the following number trees is calculated.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

2. Write the definition of the *yield* of the numbers in the string. The yield is a string with only the leaves of the tree in it. Using this definition, calculate the yields of the trees below.

   (a) 4[12[ ] 3[ ]]

   (b) 4[12[ ] 3[1[ ] 2[ ]]]

   (c) 4[12[7[ ] 7[6[ ]]] 3[1[ ] 2[ ]]]

Bibliography

Bar-Hillel, Y., M. Perles, and E. Shamir. 1961. On formal properties of simple phrase-structure grammars. *Zeitschrift fur Phonetik, Sprachwissenschaft, und Kommunikationsforschung* 14:143–177.

Büchi, J. Richard. 1960. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly* 6:66–92.

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 113–124. IT-2.

Davis, Martin D., and Elaine J. Weyuker. 1983. *Computability, Complexity and Languages*. Academic Press.

Dolatian, Hossep, and Jeffrey Heinz. 2020. Computing and classifying reduplication with 2-way finite-state transducers. *Journal of Language Modelling* 8:179–250.

Enderton, Herbert B. 1972. *A Mathematical Introduction to Logic*. Academic Press.

Enderton, Herbert B. 2001. *A Mathematical Introduction to Logic*. 2nd ed. Academic Press.

Graf, Thomas. 2011. Closure properties of Minimalist derivation tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 96–111. Heidelberg: Springer.

Graf, Thomas. 2022. Subregular linguistics: bridging theoretical linguistics and formal grammar. *Theoretical Linguistics* 48:145–184.

Graf, Thomas, and Aniello De Santo. 2019. Sensing tree automata as a model of syntactic dependencies. In *Proceedings of the 16th Meeting on the Mathematics of Language*, 12–26. Toronto, Canada: Association for Computational Linguistics.
URL `https://aclanthology.org/W19-5702`

Harrison, Michael A. 1978. *Introduction to Formal Language Theory*. Addison-Wesley Publishing Company.

Hedman, Shawn. 2004. *A First Course in Logic*. Oxford University Press.

Heinz, Jeffrey. 2018. The computational nature of phonological generalizations. In *Phonological Typology*, edited by Larry Hyman and Frans Plank, Phonetics and Phonology, chap. 5, 126–195. De Gruyter Mouton.

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Addison-Wesley.

Huybregts, Riny. 1984. The weak inadequacy of context-free phrase structure grammars. In *Van periferie naar kern*, edited by Ger de Haan, Mieke Trommelen, and Wim Zonneveld, 81–99. Dordrecht, The Netherlands: Foris.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Description*. The Hague: Mouton.

Kaplan, Ronald, and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20:331–378.

Kleene, S.C. 1956. Representation of events in nerve nets. In *Automata Studies*, edited by C.E. Shannon and J. McCarthy, 3–40. Princeton University. Press.

Kobele, Gregory M. 2011. Minimalist tree languages are closed under intersection with recognizable tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 129–144. Berlin: Springer.

Lambert, Dakotah. 2022. Unifying classification schemes for languages and processes with attention to locality and relativizations thereof. Doctoral dissertation, Stony Brook University.
URL `https://vvulpes0.github.io/PDF/dissertation.pdf/`

McNaughton, Robert, and Seymour Papert. 1971. *Counter-Free Automata*. MIT Press.

Partee, Barbara, Alice ter Meulen, and Robert Wall. 1993. *Mathematical Methods in Linguistics*. Dordrect, Boston, London: Kluwer Academic Publishers.

Roark, Brian, and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford: Oxford University Press.

Rogers, Hartley. 1967. *Theory of Recursive Functions and Effective Computability*. McGraw Hill Book Company.

Rogers, James. 1998. *A Descriptive Approach to Language-Theoretic Complexity*. Stanford, CA: CSLI Publications.

Rogers, James, Jeffrey Heinz, Margaret Fero, Jeremy Hurst, Dakotah Lambert, and Sean Wibel. 2013. Cognitive and sub-regular complexity. In *Formal Grammar*, edited by Glyn Morrill and Mark-Jan Nederhof, vol. 8036 of *Lecture Notes in Computer Science*, 90–108. Springer.

Scott, Dana, and Michael Rabin. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 5:114–125.

Shieber, Stuart. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8:333–343.

Sipser, Michael. 1997. *Introduction to the Theory of Computation*. PWS Publishing Company.

Stabler, Edward P. 2019. Three mathematical foundations for syntax. *Annual Review of Linguistics* 5:243–260.

Thomas, Wolfgang. 1982. Classifying regular events in symbolic logic. *Journal of Computer and Systems Sciences* 25:370–376.