Theoretical Computational Linguistics: Finite-state Automata

Jeffrey Heinz

February 11, 2025

Contents

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) $f : X \rightarrow Y$.

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. $\{w \in \Sigma^* : w \textit{ does not contain the string } bb \textit{ on the \{b,c\} tier}\}$.
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. $\{w \in \Sigma^* : \textit{the number of bs is the same as the number of cs in } w\}$.
19. $\{w \in \Sigma^* : \textit{the number of as, bs, and cs is the same in } w\}$.
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)*	$[\![(R\&S)]\!] = [\![R]\!] \cap [\![S]\!]$
• $\overline{\mathrm{R}}$	*(complement)*	$[\![\overline{\mathrm{R}}]\!] = \Sigma^* - [\![R]\!]$

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, among other contributions, provides a written treatment, as does Lambert (2024). 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 $S = \Sigma - T$, and 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 $[T]w$ be the language $S^* I_T(\sigma_1)S^* I_T(\sigma_2)S^* \ldots S^* I_T(\sigma_n)S^*$. 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)*	$\llbracket \langle \sigma_1\sigma_2 \ldots \sigma_n \rangle \rrbracket$	$=$	$\Sigma^*\sigma_1\sigma_2 \ldots \sigma_n\Sigma^*$
• $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	*(unanchored subsequence)*	$\llbracket \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \rrbracket$	$=$	$\Sigma^*\sigma_1\Sigma^*\sigma_2\Sigma^* \ldots \Sigma^*\sigma_n\Sigma^*$
• $\rtimes\langle \sigma_1\sigma_2 \ldots \sigma_n \rangle$	*(left-anchored substring)*	$\llbracket \rtimes \langle \sigma_1\sigma_2 \ldots \sigma_n \rangle \rrbracket$	$=$	$\sigma_1\sigma_2 \ldots \sigma_n\Sigma^*$
• $\rtimes\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	*(left-anchored subsequence)*	$\llbracket \rtimes \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \rrbracket$	$=$	$\sigma_1\Sigma^*\sigma_2\Sigma^* \ldots \Sigma^*\sigma_n\Sigma^*$
• $\ltimes\langle \sigma_1\sigma_2 \ldots \sigma_n \rangle$	*(right-anchored substring)*	$\llbracket \ltimes \langle \sigma_1\sigma_2 \ldots \sigma_n \rangle \rrbracket$	$=$	$\Sigma^*\sigma_1\sigma_2 \ldots \sigma_n$
• $\ltimes\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	*(right-anchored subsequence)*	$\llbracket \ltimes \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \rrbracket$	$=$	$\Sigma^*\sigma_1\Sigma^*\sigma_2\Sigma^* \ldots \Sigma^*\sigma_n$
• $\rtimes \ltimes \langle \sigma_1\sigma_2 \ldots \sigma_n \rangle$	*(anchored substring)*	$\llbracket \rtimes \ltimes\langle \sigma_1\sigma_2 \ldots \sigma_n \rangle \rrbracket$	$=$	$\{\sigma_1\sigma_2 \ldots \sigma_n\}$
• $\rtimes \ltimes \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$	*(anchored subsequence)*	$\llbracket \rtimes \ltimes\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \rrbracket$	$=$	$\sigma_1\Sigma^*\sigma_2\Sigma^* \ldots \Sigma^*\sigma_n$
If $R_1, R_2, \ldots R_n$ are PLEs then so are:				
• $\neg R_1$	*(complement)*	$\llbracket \neg R_1 \rrbracket$	$=$	$\Sigma^* - \llbracket R_1 \rrbracket$
• $*R_1$	*(Kleene star)*	$\llbracket * R_1 \rrbracket$	$=$	$\llbracket R_1 \rrbracket^*$
• $[\sigma_1, \sigma_2, \ldots \sigma_n]R_1$	*(tier max-inv-projection)*	$\llbracket [\sigma_1, \sigma_2, \ldots \sigma_n]R_1 \rrbracket$	$=$	$[\sigma_1, \sigma_2, \ldots \sigma_n] \llbracket R_1 \rrbracket$
• $\wedge\{R_1, R_2, \ldots R_n\}$	*(intersection)*	$\llbracket \wedge \{R_1, R_2, \ldots R_n\} \rrbracket$	$=$	$\bigcap_{1 \leq i \leq n} \llbracket R_i \rrbracket$
• $\vee\{R_1, R_2, \ldots R_n\}$	*(union)*	$\llbracket \vee \{R_1, R_2, \ldots R_n\} \rrbracket$	$=$	$\bigcup_{1 \leq i \leq n} \llbracket R_i \rrbracket$
• $\circ\{R_1, R_2, \ldots R_n\}$	*(concatenation)*	$\llbracket \circ \{R_1, R_2, \ldots R_n\} \rrbracket$	$=$	$\llbracket R_1 \rrbracket \circ \llbracket R_2 \rrbracket \circ \ldots \circ \llbracket R_n \rrbracket$

Theorem 4. $\llbracket CUE \rrbracket \subsetneq \llbracket SFE \rrbracket \subsetneq \llbracket RE \rrbracket = \llbracket GRE \rrbracket = \llbracket PLE \rrbracket$

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*

$$\alpha \to \beta$$

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

-

$$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \to \textbf{\textit{john}} \text{ VP1} \\ \text{VP1} \to \textbf{\textit{laughed}} \\ \text{VP1} \to \textbf{\textit{laughed}} \text{ VP2} \\ \text{VP2} \to \textbf{\textit{and laughed}} \\ \text{VP2} \to \textbf{\textit{and laughed}} \text{ VP2} \end{array} \right\}$$

Example 3. Consider the following grammar G_2:

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

- $\mathcal{N} = \{$S, A, B$\}$; and

-

$$\mathcal{R} = \left\{ \begin{array}{l} \text{S} \to \text{ABS} \\ \text{S} \to \lambda \\ \text{AB} \to \text{BA} \\ \text{BA} \to \text{AB} \\ \text{A} \to \text{a} \\ \text{B} \to \text{b} \end{array} \right\}$$

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

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

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); Rogers and Heinz (2014).

[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 $w = \sigma_1\sigma_2\ldots\sigma_n$:*

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

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*

$$\begin{array}{lll} \textit{(I8)} & (\exists X)[\varphi] & \textit{\textbf{(existential quantification for sets)}} \\ \textit{(I9)} & (\forall X)[\varphi] & \textit{\textbf{(universal quantification for sets)}} \end{array}$$

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.

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

That's it!

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

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

(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 ($\lhd$) so that $[\![x \lhd y]\!]$ is true iff $y = x + 1$ (so y is the next position after x).

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

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.

$$\llbracket FO(\triangleleft)\rrbracket \subsetneq \llbracket F(<)\rrbracket \subsetneq \llbracket MSO(\triangleleft)\rrbracket = \llbracket MSO(<)\rrbracket$$

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	Representation of Order	
Logic	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.

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

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.

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

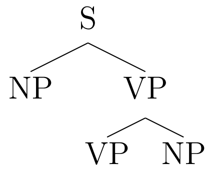
Here are some examples of trees.

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

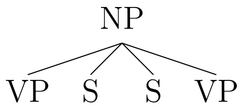
1. S[NP[ ] VP[VP[ ] NP[ ]]]
2. NP[VP[ ] S[ ] S[ ] VP[ ]]
3. NP[NP[NP[ ] VP[ ] S[ ]]]

We might draw these structures as follows.

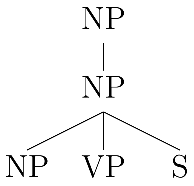
(1)



(2)



(3)



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 $t = a[t_1t_2 \ldots t_n]$ where $a \in \Sigma$ and each t_i is a tree. Then $|t| = 1 + |t_1| + |t_2| + \ldots + |t_n|$.*

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

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

Chapter 4

String Acceptors

4.1 Deterministic Finite-state String Acceptors

4.1.1 Orientation

This section is about deterministic finite-state acceptors for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute the output from some input. This is in contrast to*non-deterministic machines* which can be thought of as pursuing multiple computations simultaneously. The term *acceptor* is synonymous with *recognizer*. It means that this machine solves *membership problems*: given a set of objects X and input object x, does x belong to X? The term *string* means we are considering the membership problem over stringsets. So X is a set of strings (so $X \subseteq \Sigma^*$) and the input x is a string.

4.1.2 Definitions

Definition 14. *A* deterministic finite-state acceptor (DFA) *is a tuple* $(Q, \Sigma, q_0, F, \delta)$ *where*

- *Q is a finite set of states;*
- *Σ is a finite set of symbols (*the alphabet*);*
- *$q_0 \in Q$ is the* initial *state;*
- *$F \subseteq Q$ is a set of* accepting *(*final*) states; and*
- *δ is a function with domain $Q \times \Sigma$ and co-domain Q. It is called the* transition function.

We extend the domain of the transition function to $Q \times \Sigma^*$ as follows. In these notes, the empty string is denoted with λ.

$$\begin{aligned} \delta^*(q, \lambda) &= q \\ \delta^*(q, aw) &= \delta^*((\delta(q, a), w) \end{aligned} \tag{4.1}$$

Consider some DFA $A = (Q, \Sigma, q_0, F, \delta)$ and string $w \in \Sigma^*$. If $\delta^*(q_0, w) \in F$ then we say A *accepts/recognizes/generates* w. Otherwise A *rejects* w.

Definition 15. *The stringset recognized by A is $L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$.*

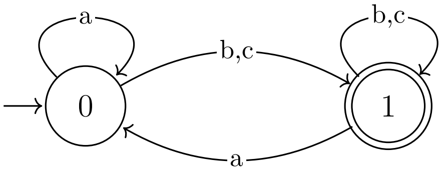
4.1.3 Exercises

Exercise 24. This exercise is about designing DFA. Let $\Sigma = \{a, b, c\}$. Write DFA which express the following generalizations on word well-formedness.

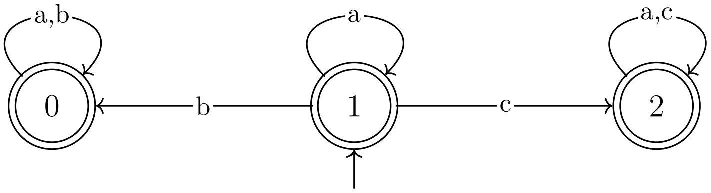
1. All words begin with a consonant, end with a vowel, and alternate consonants and vowels.
2. Words do not contain *aaa* as a substring.
3. If a word begins with *a*, it must end with *c*.
4. Words must contain two *b*s.
5. All words have an even number of vowels.

Exercise 25. This exercise is about reading and interpreting DFA. Provide generalizations in English prose which accurately describe the stringset these DFA describe.

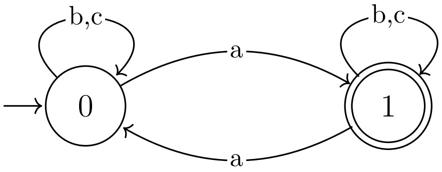
1.



2.



3.



4. Write the DFA in #1-3 in mathematical notation. So what is Q, Σ, q_0, F, and δ?

4.2 Properties of DFA

Note that for a DFA A, its transition function δ may be partial. That is, there may be some $q \in Q, a \in \Sigma$ such that $\delta(q, a)$ is undefined. If δ is a partial function, δ^* will be also. It is assumed that if $\delta^*(q_0, w)$ is undefined, then A rejects w.

We can always make δ total by adding one more state to Q. To see how, call this new state $\diamondsuit$. Then for each $(q, a) \in Q \times \Sigma$ such that $\delta(q, a)$ is undefined, define $\delta(q, a)$ to equal $\diamondsuit$. Every string which was formerly undefined w.r.t. to δ^* is now mapped to $\diamondsuit$, a non-accepting state. This state is sometimes called the *sink* state or the *dead* state.

Definition 16. *A DFA is* complete *if δ is a total function. Otherwise it is* incomplete*.*

It is possible to write DFA which have many useless states. A state can be useless in two ways. First, there may be no string which forces the machine to transition into the state. Second, there may be a state from which no string

Definition 17. *A state q in a DFA A is* useful *if there is a string w such that $\delta^*(q_0, w) = q$ and a string v such that $\delta^*(q, v) \in F$. Otherwise q is* useless*. If every state in A is useful, then A is called* trim*.*

Not all complete DFAs are trim. If there is a sink state, it is useless in the above sense of the word.

Definition 18. *A DFA A is* minimal *if there is no other DFA A' such that $L(A) = L(A')$ and A' has fewer states than A.*

Technically, not all complete DFAs are minimal. If there is a sink state, it is not minimal.

Exercise 26. Consider the DFAs in the exercise 25. Are they complete? Trim? Minimal?

4.3 Some Closure Properties of Regular Languages

A set of objects X is *closed* under an operation $\circ$ if for all objects $x, y \in X$ it is the case that $x \circ y \in X$ too.

We can easily show that the union of any two regular stringsets R and S is also regular. Let $A_R = (Q_R, \Sigma, q_{0R}, F_R, \delta_R)$ be the DFA recognizing R and let $A_S = (Q_S, \Sigma, q_{0S}, F_S, \delta_S)$ be the DFA recognizing S. We can assume A_R and A_S are complete. We assume the same alphabet.

Construct $A = (Q, \Sigma, q_0, F, \delta)$ as follows.

- $Q = Q_R \times Q_S$.
- $q_0 = (q_{0R}, q_{0S})$.
- $F = \{(q_r, q_s) \mid q_r \in F_R \textbf{ or } q_s \in F_S\}$.
- $\delta((q_r, q_s), a) = (q'_r, q'_s)$ where $\delta_R(q_r, a) = q'_r$ and $\delta_S(q_s, a) = q'_s$.

Theorem 9. $L(A) = R \cup S$.

Similarly, the same kind of construction shows that the intersection of any two regular stringsets is regular. Construct $B = (Q, \Sigma, q_0, F, \delta)$ as follows.

- $Q = Q_R \times Q_S$.
- $q_0 = (q_{0R}, q_{0S})$.
- $F = \{(q_r, q_s) \mid q_r \in F_R$ **and** $q_s \in F_S\}$.
- $\delta((q_r, q_s), a) = (q'_r, q'_s)$ where $\delta_R(q_r, a) = q'_r$ and $\delta_S(q_s, a) = q'_s$.

Theorem 10. $L(B) = R \cap S$.

Here are some additional questions we are interested in for regular stringsets R and S.

1. Is the complement of R (denoted $\overline{R}$) a regular stringset?
2. Is $R \backslash S$ a regular stringset?
3. Can we decide whether $R \subseteq S$?
4. Is $R \circ S$ a regular stringset? (Note $R \circ S = \{rs \mid r \in R$ and $s \in S\}$
5. Is R^* a regular stringset? (Note $R^0 = \{\lambda\}$, $R^n = R^{n-1}R$, $R^* = \bigcup_{n \in \mathbb{N}^0} R^n$)

The answers to all of these questions is Yes. With a little thought about complete DFA, the answers to first three follow very easily.

Theorem 11. *If R is a regular stringset then the complement of R is regular.*

Proof (Sketch). If R is a regular stringset then there is a complete DFA $A = (Q, \Sigma, q_0, F, \delta)$ which recognizes it. Let $B = (Q, \Sigma, q_0, F', \delta)$ where $F' = Q \backslash F$. We claim $L(B) = \overline{R}$. □

Corollary 1. *If R, S are regular stringsets then so is $R \backslash S$ since $R \backslash S = R \cap \overline{S}$.*

Corollary 2. *If R, S are regular stringsets then it is decidable whether $R \subseteq S$ since $R \subseteq S$ iff $R \backslash S = \varnothing$.*

Corollary 3. *If R, S are regular stringsets then it is decidable if $R = S$ since $R = S$ iff $R \subseteq S$ and $S \subseteq R$.*

We postpone explaining how and why for the last two questions.

4.4 Non-deterministic Finite-state String Acceptors

4.4.1 Orientation

This section is about non-deterministic finite-state acceptors for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *non-deterministic* means there are potentially many computational paths the machine may follow to compute the output from some input string. As we will see later, *deterministic machines* pursue at most a single computation. The term *acceptor* is synonymous with *recognizer*. It means that this machine solves *membership problems*: given a set of objects X and input object x, does x belong to X? The term *string* means we are considering the membership problem over stringsets. So X is a set of strings (so $X \subseteq \Sigma^*$) and the input x is a string.

4.4.2 Non-deterministic string acceptors

Definition 19. *A* non-deterministic finite-state acceptor (NFA) *is a tuple* $(Q, \Sigma, I, F, \delta)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the alphabet*);*
- $I \subseteq Q$ *is a set of* initial (start) *states;*
- $F \subseteq Q$ *is a set of* accepting (final) *states; and*
- $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$ *is the* transition relation*.*

The symbol ϵ denotes a “free” change of state; that is the state of the system can change without any input being consumed.

There are a couple ways to think about the transition function. One way is to think that when processing a string, there are multiple paths to take. For example, consider the machine below which recognizes the set of strings where every non-empty string must end in a consonant. Given the input string *bb*, when the first *b* is read and the machine is in state 0, it could transition to state 1 or 2. In fact, the epsilon transition from state 0 means that the machine could transition to state 1 without even reading the first symbol of the string. If

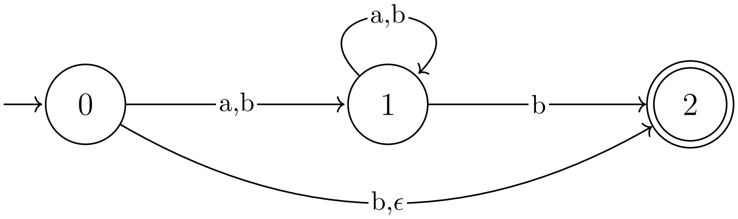


Figure 4.1: An example NFA.

we think of the machine as occupying a single state at any given moment, there are multiple paths that need to be explored. As long as least one them leads from a start state to a final state when reading the whole string, we can say the machine accepts the string. Under this way of thinking, there are potentially many paths that need to be explored and tracked until a successful one is found.

The other way to think about is to think of the machine being in several states simultaneously. In the example above, when the first b is read, we can think of the machine as transitioning from state 0 to both states 1 and 2. In other words, it originally was in state 0 and after reading the b, it is in "state" $\{1, 2\}$. In fact, because of the epsilon transition from state 0, before the first b is read, the machine can be said to be in both states 0 and 2. It is this way of thinking that we use to define the set of strings that NFA recognize, and it also is the intuition behind the fact that stringsets recognizable by non-deterministic finite-state acceptors are the same as those recognizable by DFAs. We will use this same insight to define how to decide whether NFA recognizes an input string or not.

The *epsilon closure* of a set of states $S \subseteq Q$ is the smallest subset S' of Q satisfying these properties:

1. $S \subseteq S'$
2. For all $q, r \in Q$: if $q \in S'$ and $r \in \delta(q, \epsilon)$ then $r \in S'$.

The epsilon closure of Q is denoted $C_\epsilon(Q)$.

δ is a relation and in order to define a "process" function, we use δ to define related functions. First, we define $\delta' : Q \times \Sigma \to \wp(Q)$. Recall that for any set A, $\wp(A)$ denotes the powerset of A, which is the set of all subsets of A.

$$\delta'(q, a) = \{q' \mid (q, a, q') \in \delta\}$$

Next we define $\delta'' : \wp(Q) \times \Sigma \to \wp(Q)$ as follows.

$$\delta''(Q, a) = \bigcup_{q \in Q} \delta'(q, a)$$

Finally, we can explain how a NFA processes strings using the recursive definition below.

$$\begin{aligned} \pi(Q, \lambda) &= Q \\ \pi(Q, aw) &= \pi(C_\epsilon(\delta''(Q, a)), w) \end{aligned} \tag{4.2}$$

Consider some NFA $A = (Q, \Sigma, I, F, \delta)$ and string $w \in \Sigma^*$. If $\pi(C_\epsilon(I), w) \cap F \neq \emptyset$ then we say A *accepts/recognizes/generates* w. Otherwise A *rejects* w.

Definition 20. *The stringset recognized by A is $L(A) = \{w \in \Sigma^* \mid \pi(C_\epsilon(I), w) \cap F \neq \emptyset\}$.*

Exercise 27.

1. Non-determinism makes expressing some stringsets easier because it can be done with fewer states.

   (a) Write an acceptor for the set of strings where the second-to-last letter must be a consonant.

   (b) Write an acceptor for the set of strings where the third-to-last letter must be a consonant.

2. Use the plurality of initial states to show that if R and S are stringsets each recognized by some NFA that the union $R \cup S$ is also recognized by a NFA.

3. Use epsilon transitions to show that if R and S are stringsets each recognized by some NFA that the concatenation RS is also recognized by a NFA.

4.4.3 Closure under concatenation, union, Kleene Star

Theorem 12. $[\![NFA]\!] = [\![RE]\!]$

It is easy to prove that $[\![RE]\!] \subseteq [\![NFA]\!]$. Recall how the REs were defined with base cases and inductive cases.

Exercise 28. Write a NFA which recognizes the same language as each of the RE base cases.

We want to show the following.

1. Given NFA A, B, that there exists a NFA recognizing the $L(A) \circ L(B)$ (closure under concatenation).
2. Given NFA A, B, that there exists a NFA recognizing the $L(A) \cup L(B)$ (closure under union).
3. Given NFA A,that there exists a NFA recognizing the $(L(A))^*$ (closure under Kleene star).

The existence of epsilon transitions make this this easy to show.

1. For closure under concatenation, the NFA recognizing $L(A) \circ L(B)$ is
   - $Q = Q_A \cup Q_B$
   - $\Sigma = \Sigma_A \cup \Sigma_B$
   - $I = I_A$
   - $F = F_B$
   - $\delta = \delta_A \cup \delta_B \cup \{(q, \epsilon, r) : q \in F_A, r \in I_B\}$
2. For closure under union, the NFA recognizing $L(A) \cup L(B)$ is
   - $Q = Q_A \cup Q_B$

- $\Sigma = \Sigma_A \cup \Sigma_B$
- $I = I_A \cup I_B$
- $F = F_B \cup F_B$
- $\delta = \delta_A \cup \delta_B$

3. For closure under Kleene star, the NFA recognizing the $(L(A))^*$.

- $Q = Q_A \cup \{\heartsuit, \spadesuit\}$
- $\Sigma = \Sigma_A$
- $I = \{\heartsuit\}$
- $F = \{\spadesuit\}$
- $\delta = \delta_A \cup \{(\heartsuit, \epsilon, q) : q \in I_A\} \cup \{(q, \epsilon, \spadesuit) : q \in F_A\} \cup \{(\heartsuit, \epsilon, \spadesuit), (\spadesuit, \epsilon, \heartsuit)\}$

We can visualize the argument as follows. Figure 4.2 visualizes two NFA, A and B. Schematically, the initial states are on the left and the final states are on the right. The construction for concatenation is illustrated in Figure 4.3. The construction for union is illustrated in Figure 4.4. The construction for Kleene star is illustrated in Figure 4.5.

Figure 4.2: Two finite state machines with a set of initial states (leftside) and a set of final states (rightside).

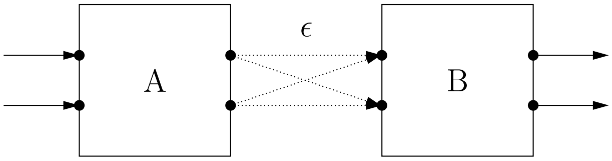


Figure 4.3: Concatenation of two NFAs.

Proving that there is a RE for any NFA is only a little more complicated. It is reviewed in several textbooks such as Sipser (1997) and Hopcroft *et al.* (2001) and we will not review it here.

Since NFA recognize the same languages as REs, we will call this class of languages *regular languages*. Next we turn to DFA.

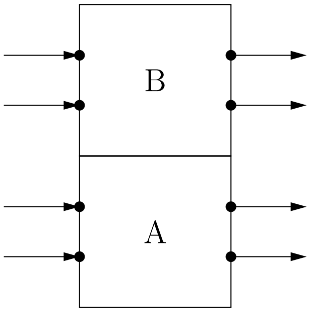


Figure 4.4: Union of two NFAs.

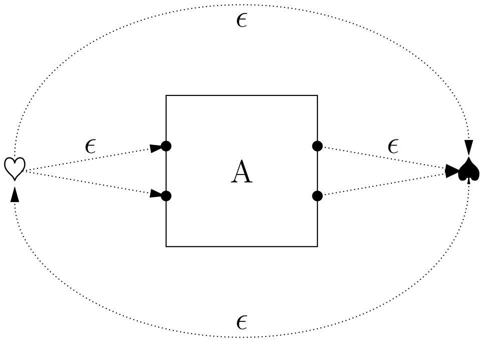


Figure 4.5: Kleene Star of one NFA.

4.5 Determinizing NFA

In this section, we prove the following theorem.

Theorem 13. $[\![NFA]\!]=[\![DFA]\!]$.

Since every DFA is a NFA, it follows that any language recognized by a DFA is also recognized by some NFA. It remains to be shown that for any language recognized by a NFA, there is a DFA that recognizes the same language.

The key idea is one encountered earlier: when processing a string in a NFA, instead of pursuing multiple paths of single states, we can pursue a single path of multiple states. Since there are finitely many states Q in a NFA, the number of sets of multiple states is also finite, and is bounded by the powerset of Q. Thus for any NFA $A = (Q_A, \Sigma_A, I_A, F_A, \delta_A)$, we can construct a DFA $B = (Q, \Sigma, I, F, \delta)$ using *the powerset construction* shown below.

- $Q = \wp(Q_A)$ is a finite set of states;
- $\Sigma = \Sigma_A$ is a finite set of symbols (*the alphabet*);
- $C_\epsilon(I_A)$ is the *initial* state;
- $q \in Q$ is a set of *accepting* state iff $q \cap F_A \neq \emptyset$
- $\delta(q, a) = r$ iff $C_\epsilon(\delta''_A(q, a)) = r$.

Then one can prove $L(A) = L(B)$, and it follows that $[\![NFA]\!] = [\![DFA]\!]$.

Algorithms which determinize non-deterministic finite-state machines do not build the entire powerset. Instead they proceed incrementally beginning with the start state I_A and proceeding through the alphabet. New states are added as they are needed.

Exercise 29. Determinize the NFA in Figure 4.1.

4.6 Minimizing DFA

Here we show that for each regular stringset R, there is a smallest DFA which recognizes R. This DFA is unique (discounting the names of the states).

Consider any DFA A. The idea is that some states are doing the "same work" as other states. Such states are said to be *indistinguishable*. States that are indistinguishable from each other are grouped into blocks. These blocks become the states of the minimal DFA which recognizes the same stringset as A.

Given a DFA A recognizing a stringset R, to find the minimal DFA recognizing R we must do the following:

1. Determine which states of A are indistinguishable.

2. Using this information, construct the minimal DFA.

4.6.1 Identifying indistinguishable states

Consider $A = (Q, \Sigma, i, F, \delta)$. Two states q, r are *distinguishable* in A if there is some string w such that $\delta^*(q, w) \in F$ and $\delta^*(r, w) \notin F$. In other words, if there is a string that causes A to transition from state q to an accepting state but would cause A to transition from state r to a rejecting state then q and r are distinguishable. We say w *distinguishes* q from r.

We can determine whether a distinguishing string w exists for q and r recursively. There are two key observations to see why. First, observe that the empty string λ distinguishes accepting states from rejecting states. Second, if w distinguishes q from r and string u causes A to transition from state q' to q and causes A to transition from state r' to r then it is the case that string uw distinguishes q' from r'. Formally:

Base case: (q, r) is distinguishable if $q \in F$ and $r \notin F$.

Inductive Step: (q', r') is distinguishable if (q, r) is distinguishable and there is $a \in \Sigma$ such that $\delta(q', a) = q$ and $\delta(r', a) = r$.

It is sufficient to check individual symbols in Σ in the inductive step and repeat it until no new distinguishable states are found.

To see why, consider the following. The first iteration checks to see if any 1-long strings distinguish states in A. The second iteration checks to see if any 2-long strings distinguish states in A. Generally, the kth iteration checks to see if any k-long strings distinguish states in A. Importantly, if there is a string w of length k distinguishing q from r in A then there must be a string v of length $k - 1$ and a symbol $a \in \Sigma$ distinguishing $q' = \delta(q, a)$ from $r' = \delta(r, a)$ in A. This justifies why the inductive step can stop iterating if no new distinguishable states are found on the present iteration. It is not possible to find a k-long string distinguishing states if no $(k - 1)$-long string distinguishes any states.

At the end of this process we have a set of distinguishable state-pairs. The state-pairs in A that are not distinguishable are *indistinguishable*.

4.6.2 Building the minimal DFA

Once the indistinguishable states have been identified, a new DFA can be constructed. The indistinguishable state-pairs of A partition its states into *blocks*. A block is just a set of states. States q and r are in the same block only if (q, r) is an indistinguishable state-pair.

The process by which distinguishable states are found ensures that the set of indistinguishable state-pairs is closed under transitivity. In other words, if (q, r) and (r, s) are are indistinguishable state-pairs then so is (q, s). More generally, the *indistiguishable relation* is an equivalence relation satisysfying not only transitivity but also reflexivity and symmetry.

Let B_q denote the block containing state q. Then the minimal DFA M recognizing $L(A)$ is given by:

- $Q_M = \{B_q \mid q \in Q\}$
- $i_M = B_i$

- $F_M = \{B_q \mid q \in F\}$
- $\delta_M(B, a) = B' \in Q_M$ such that $B' \supseteq \{\delta(q, a) \mid q \in B\}$

4.6.3 Example

Identifying indistinguishable pairs of states. This example comes from (Hopcroft *et al.*, 2001, chapter 4). We are going to minimize the automaton shown below.

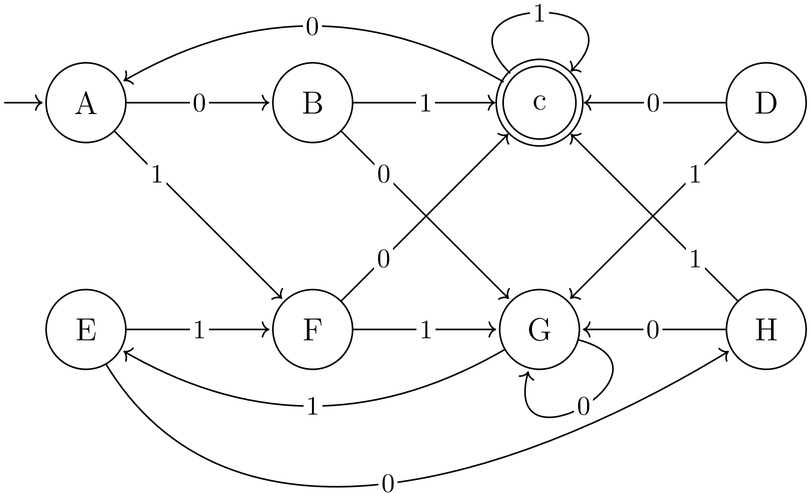


Figure 4.6: A non-minimal automaton (from Figure 4.8 in Hopcroft *et al.* (2001)).

We need to identify distinguishable states. From the base case, each rejecting state is distinguishable from each accepting state with λ.

Distinguishable pairs (Base Case):

$$\{\ \textbf{(A,C), (B,C), (D,C), (E,C), (F,C), (G,C), (H,C)}\ \}$$

A technical note: in addition to (A, C) the set should include (C, A) as well and similarly for the other pairs. However, we leave out these reflexive pairs for readability.

Next we repeatedly apply the inductive case. For each indistinguishable pair (q, r), we ask is there $a \in \Sigma$ such that $(\delta(q, a), \delta(r, a))$ is distinguishable? If so, we add (q, r) to the list of distinguishable pairs. For example, consider the pair (A, B). Since $(\delta(A, 1), \delta(B, 1)) = (F, C)$ and (F, C) is distinguishable, we add (A, B) to the list. On the other hand (A, E) is not added to the list because neither $(\delta(A, 1), \delta(E, 1)) = (F, F)$ nor $(\delta(A, 0), \delta(E, 0)) = (B, H)$ are distinguishable states.

The added ones are shown in bold below.

Distinguishable pairs (Inductive Step 1):

$$\left\{\begin{array}{l} (A,C),(B,C),(D,C),(E,C),(F,C),(G,C),(H,C) \\ \mathbf{(A,B),\ (A,D),\ (A,F),\ (A,H),\ (B,D),\ (B,E),\ (B,F),} \\ \mathbf{(B,G),\ (D,E),\ (D,G),\ (D,H),\ (E,F),\ (E,G),\ (E,H),} \\ \mathbf{(F,G),\ (F,H),\ (G,H)} \end{array}\right\}$$

At this point, only four pairs of states are not yet known to be distinguishable. These are $\{(A,E),(A,G),(B,H),(D,F)\}$. We repeat the inductive step, to see if any new distinguished pairs are discovered. As before (A,E) is not found to be distinguishable. On the other hand, (A,G) is distinguishable now with string 1 and states F and E are now known to be distinguished. In fact, as a result of this step, only (A,G) is added as shown below.

Distinguishable pairs (Inductive Step 2):

$$\left\{\begin{array}{l} (A,C),(B,C),(D,C),(E,C),(F,C),(G,C),(H,C) \\ (A,B),(A,D),(A,F),(A,H),(B,D),(B,E),(B,F), \\ (B,G),(D,E),(D,G),(D,H),(E,F),(E,G),(E,H), \\ (F,G),(F,H),(G,H),\mathbf{(A,G)} \end{array}\right\}$$

The inductive step is repeated again, and this time no new distinguishable states are discovered. Therefore the iteration of the inductive steps terminates.

Distinguishable pairs (Inductive Step 3):

$$\left\{\begin{array}{l} (A,C),(B,C),(D,C),(E,C),(F,C),(G,C),(H,C) \\ (A,B),(A,D),(A,F),(A,H),(B,D),(B,E),(B,F), \\ (B,G),(D,E),(D,G),(D,H),(E,F),(E,G),(E,H), \\ (F,G),(F,H),(G,H),(A,G) \end{array}\right\}$$

Thus, the only indistinguishable pairs are $\{(A,E),(B,H),(D,F)\}$.

Building the minimal automaton. With this information, the states are partitioned into blocks:

$$\Big\{\{A,E\},\{B,H\},\{C\},\{D,F\},\{H\}\Big\}$$

These blocks are the states of the minimal automaton. Since block $\{A,E\}$ contains the start state of the original acceptor, this block is the start state. Since block $\{C\}$ contains a final state of the original acceptor, this block is a final state.

Finally, we calculate the transition function as shown in the diagram below. To illustrate, first Consider the transition from block $\{A,E\}$ upon reading 1. With 1, the original delta function maps state A to F and E to F. There is exactly one block which contains F; this is the block $\{D,F\}$. Hence the minimal machine transitions from state $\{A,E\}$ to $\{D,F\}$ with 1. The other transitions are determined similarly.

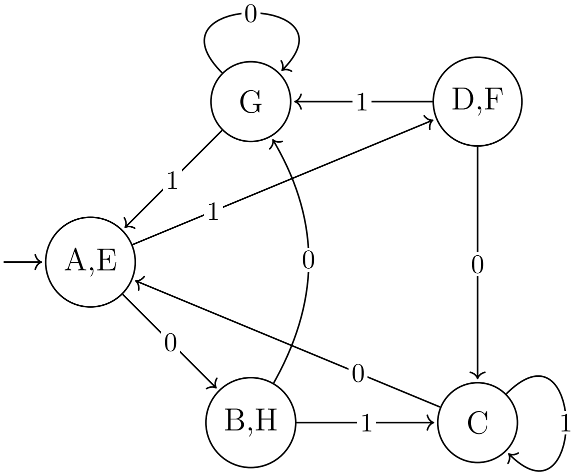


Figure 4.7: The minimal automaton recognizing the same stringset as the one in Figure 4.6.

4.7 Nonregular stringsets

Regular stringsets were defined as those subsets of Σ^* whose membership problem was solvable with a deterministic finite-state acceptor. It was also expressed that finite-state solvable membership problems invoke constant memory. This can be understood to mean that the number of states is constant and does not increase even as the words grow longer and longer.

In this section, we give a few examples of stringsets whose membership problem is not solvable with any DFSA. In each case, I hope to convey that in order to solve the membership problem for any word, the number of states needs to increase as words get longer.

Example 9. Let $\Sigma = \{a, b, c\}$.

1. The set of strings that with each string containing at least as many as as bs. Formally, we can define this set of strings as follows. For each $a \in \Sigma$ and $w \in \Sigma^*$, let $|w|_a$ be the number of times a occurs in w. So $|aabaa|_a = 4$ and $|caca|_c = 2$ and so on. Then the set of strings we are interested in can be expressed as $S_1 = \{w \mid |w|_a \geq |w|_b\}$.

2. The palindrome language. Recall that a palindrome is a word that is the same when written both forwards and backwards. So the palindrome language contains all and only those words that are palindromes. Formally, we can define this set of strings as follows. Let us write w^R to express the reverse of w. So $abac^R = caba$. Then this stringset can be expressed as $S_2 = \big\{wxw^R \mid w \in \Sigma^*, x \in \{\lambda, a, b, c\}\big\}$.

3. The a^nb^n stringset. This is pronounced "a to the n, b to the n". Recall that a^n defines the string with a concatenated to itself n times. So $a^4 = aaaa$ and $c^3 = ccc$ and so on. Note $a^0 = \lambda$. So this stringset can be expressed as $S_3 = \{a^nb^n \mid 0 \leq n\}$.

4.7.1 Exercises

Exercise 30. For each of the above examples, we try to write a deterministic finite-state acceptor. This exercise will help us understand why it is impossible.

1. First write an acceptor for S_1 for words up to size 3. Next try to write the acceptor for words up to size 5. What is happening? What information is each state keeping tracking of?

2. First write an acceptor for S_2 for words up to size 2. Next try to write the acceptor for words up to size 4. What is happening? What information is each state keeping tracking of?

3. First write an acceptor for S_3 for words up to size 2. Next try to write the acceptor for words up to size 4. What about up to size 6? What is happening? What information is each state keeping tracking of?

In each case, the information that the states need to keep track of grows as words get longer. This is the basic insight into why the problem of these stringsets (and many like them) cannot be solved with finite-state acceptors.

4.7.2 Formal Analysis

Formal proofs that these are nonregular exist, based on abstract properies of regular stringsets as discussed in (Sipser, 1997; Hopcroft *et al.*, 2001) and elsewhere.

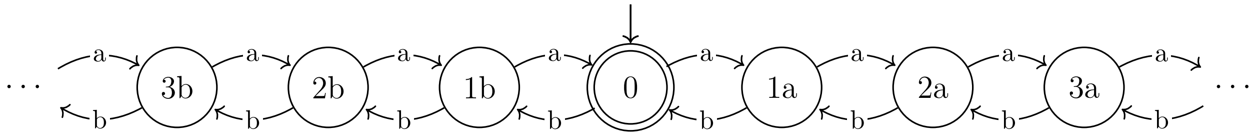
Here is one way to provide a rigorous proof with what we have learned so far.

Recall that when minimizing a DFSA A, we had to determine whether two states did the "same work" or not. We said states did different work, if there was a string that *distinguished* them. A string *distinguishes* state q from r if A would transition to a final state from q but to a non-final state from r (or vice versa). We can use distinguishing strings to see that a DFSA for the above stringsets would have infinitely many states.

Suppose there was a DFSA A for S_1. Let q_0 be the inital state of A and let $q_a = \delta(q_0, a)$. Now we can ask are there any strings which distinguish q from q_a? The answer is Yes. The string b distinguishes them because A must transition to an accepting state from q_a with b, but must transition to a non-accepting state from q_0 with b. Thus q_0 and q_a are distinct states.

We can repeat this reasoning. Let $q_{aa} = \delta(q_a, a)$. Is there a string which distinguishes q_{aa} from q_a? Yes, in fact the string bb distinguishes them because A must transition to an accepting state from q_{aa} with bb, but must transition to a non-accepting state from q_a with bb. So q_a and q_{aa} are distinct states. What about q_{aa} and q_0? They are distinct states too as witnessed by the distinguishing string b. So now we have three distinct states.

More generally, define state q_i to be $\delta^*(q_0, a^i)$. For any numbers n, m with $n > m$, one can show that q_n is a distinct state from q_m. We have to find a distinguishing string for these two states. The string b^n is such a string. The DFSA A must transition to an accepting state from q_n with b^n because the word $a^n b^n$ has at least as many as as bs. However, A transitions to an non-accepting state from q_m with b^n because the word $a^m b^n$ has more bs as as since $n > m$. Thus A must have infinitely many states if it is to accept words of arbitrary length. And we haven't even looked at words beginning with bs yet!



Exercise 31. Present an argument like the one above for S_2 and S_3.

Chapter 5

Tree Acceptors

5.1 Deterministic Finite-state Bottom-up Tree Acceptors

5.1.1 Orientation

This section is about deterministic bottom-up finite-state tree acceptors. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *acceptor* means this machine solves *membership problem*: given a set of objects X and input object x, does x belong to X? The term *tree* means we are considering the membership problem over *treesets*. The term *bottom-up* means that for each node a in a tree, the computation solves the problem by assigning states to the children of a before assigning states to a itself. This contrasts with *top-down* machines which assign states to a first and then the children of a. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

Acceptor is synonymous with *recognizer*. *Treeset* is synonymous with *tree language*.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

5.1.2 Definitions

We will use the following definition of trees.

Definition 21 (Trees)**.** *We assume an alphabet Σ and symbols [ ] not belonging to Σ.*

Base Cases: *For each $a \in \Sigma$, $a[\,]$ is a tree.*

Inductive Case: *If $a \in \Sigma$ and $t_1 t_2 \ldots t_n$ is a string of trees of length n then $a[t_1 t_2 \cdot \ldots t_n]$ is a tree.*

Let Σ^T denote the set of all trees of finite size using Σ. We also write $a[\lambda]$ for $a[\,]$.

It will be helpful to review the following concepts related to functions: *domain, co-domain, image*, and *pre-image*. A function $f : X \to Y$ is said to have domain X and co-domain Y. This means that if $f(x)$ is defined, we know $x \in X$ and $f(x) \in Y$. However, f may not be defined for all $x \in X$. Also, f may not be *onto* Y, there may be some elements in Y that are never "reached" by f.

This is where the concepts *image* and *pre-image* come into play. The image of f is the set $\{f(x) \in Y \mid x \in X, f(x) \text{ is defined}\}$. The pre-image of f is the set $\{x \in X \mid f(x) \text{ is defined}\}$. So the pre-image of f is the subset of the domain of f where f is defined. The image of f is the corresponding subset of the co-domain of f.

With this in place, we can define our first tree acceptor.

Definition 22 (DFBTA)**.** *A* Deterministic Bottom-up Finite-state Acceptor (DFBTA) *is a tuple (Q, Σ_r, F, δ) where*

- *Q is a finite set of states;*
- *Σ is a finite alphabet;*
- *$F \subseteq Q$ is a set of* accepting *(*final*) states; and*
- *$\delta : Q^* \times \Sigma \to Q$ is* the transition function*.* The pre-image of δ must be finite*. This means we can write it down—for example, as a list.*

We use the transition function δ to define a new function $\delta^* : \Sigma^T \to Q$ as follows.

$$\begin{aligned}\delta^*(a[\lambda]) &= \delta(\lambda, a)\\ \delta^*(a[t_1 \cdots t_n]) &= \delta\big(\delta^*(t_1) \cdots \delta^*(t_n), a\big)\end{aligned} \tag{5.1}$$

There are some important consequences to the formulation of δ^* shown here. One is that δ^* is undefined on tree $a[t_1 \cdots t_n]$ if no transition $\delta(q_1 \cdots q_n, a)$ is defined.

(Also, I am abusing notation since δ^* is strictly speaking not the transitive closure of δ.)

Definition 23 (Treeset of a DFBTA)**.** *Consider some DFBTA $A = (Q, \Sigma, F, \delta)$ and tree $t \in \Sigma^T$. If $\delta^*(t)$ is defined and belongs to F then we say A* accepts/recognizes *t. Otherwise A* rejects *t. The treeset recognized by A is $L(A) = \{t \in \Sigma^T \mid \delta^*(t) \in F\}$.*

The use of the 'L' denotes "Language" as treesets are traditionally referred to as *formal tree languages.*

We observe that since the pre-image of δ is finite, there is some n such that for all $(\vec{q}, a, q') \in \delta$, it is the case that $|\vec{q}| \leq n$. In other words, δ is effectively a function from $Q^n \times \Sigma$ to Q. We say δ is n-wide and any DFBTA with an n-wide δ is also called n-wide.

Definition 24 (n-wide Recognizable Treesets)**.** *A treeset is n-*wide recognizable *if there is a n-wide DFBTA that recognizes it.*

Definition 25 (Recognizable Treesets)**.** *A treeset is* recognizable *if there exists $n \in \mathbb{N}$, such that an n-wide DFBTA that recognizes it.*

5.1.3 Notes on Definitions

We have departed a bit from standard definitions. In particular, most introductions to tree automata make use of a particular kind of alphabet called a *ranked alphabet*. A ranked alphabet Σ_r is a finite alphabet Σ with an arity function $ar : \Sigma \rightarrow \mathbb{N}$. We write $\Sigma_r = (\Sigma, ar)$. The idea is that each symbol comes pre-equipped with a number which indicates how many children it has in trees. This is reasonable provided a node's label determines how many children it can have.

Strictly speaking, a ranked alphabet is not a necessary feature of tree automata. There are two substantive reasons to adopt it. First, using it helps ensure that the transition function is finite. (So it can accomplish the same thing as our requirement that the pre-image of δ be finite.). Second, it can help ensure our transition function is total; that is, defined for every element of the alphabet and the possible states of its children.

A ranked alphabet effectively defines a maximum width n for treees where $n = \texttt{max}\{ar(a) \mid a \in \Sigma\}$. Thus a DFBTA defined with a ranked alphabet will always recognize a treeset which is a subset of $\Sigma^{\mathrm{T},n}$.

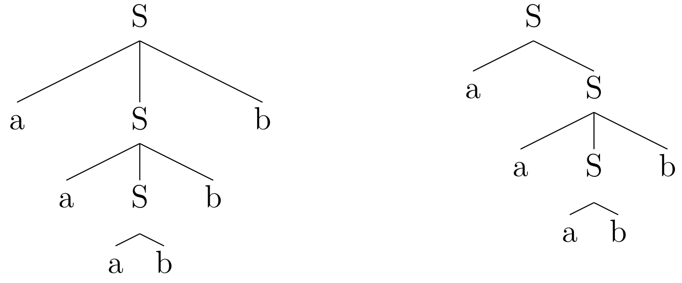
5.1.4 Examples

Example 10. Let $A = (Q, \Sigma, F, \delta)$ with its parts defined as follows.

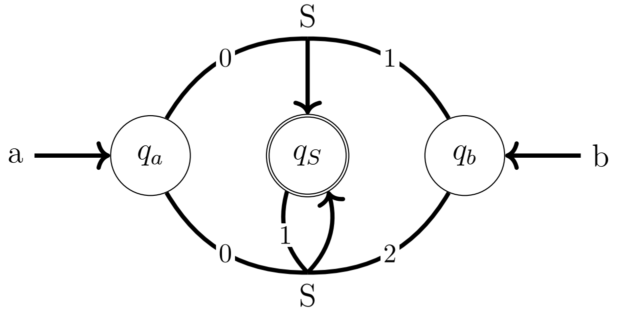
- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $F = \{q_S\}$

- $\delta(\lambda, a) = q_a$
- $\delta(\lambda, b) = q_b$
- $\delta(q_a q_b, S) = q_S$
- $\delta(q_a q_S q_b, S) = q_S$

Let us see how the acceptor A processes the two trees below as inputs.



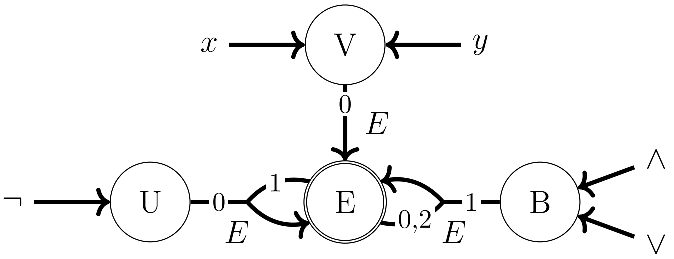
A Lambert graph of this automaton is shown below.



Example 11. The next example regards propositional logic. Propositional logic has one unary operator, negation (¬), and two binary operators: conjunction (∧) and disjunction (∨). We consider the case with two variables $\{x, y\}$. We can define trees of propositional logic as follows.

- x[] and y[] are trees of propositional logic.
- If A and B are trees of propositional logic, then so are ¬ [A], ∧[A,B], and ∨[B,A].

A Lambert graph of the automata recognizing this language is shown below.



5.1.5 Observations

- For every symbol $a \in \Sigma$ which can be leaf in a tree, you will need to define a transition $\delta(\lambda, a)$.
- For every symbol $a \in \Sigma$ which can have n children, you will need to define a transition $\delta(q_1 \cdots q_n, a)$.

5.1.6 Connection to Context-Free Languages

Recognizable treesets are closely related to the derivation trees of context-free languages.

Theorem 14.

- *Let G be a context-free word grammar, then the set of derivation trees of L(G) is a recognizable tree language.*

- *Let L be a recognizable tree language then Yield(L) is a context-free word language.*
- *There exists a recognizable tree language not equal to the set of derivation trees of any context-free language. Thus the class of derivation treesets of context-free word languages is a proper subset of the class of recognizable treesets.*

5.2 Deterministic Top-down Finite-state Tree Acceptors

5.2.1 Orientation

This section is about deterministic top-down finite-state tree acceptors. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *acceptor* means this machine solves *membership problem*: given a set of objects X and input object x, does x belong to X? The term *tree* means we are considering the membership problem over *treesets*. The term *top-down* means that for each node a in a tree, the computation solves the problem by assigning a state to the parent of a before assigning a state to a itself. This contrasts with *bottom-up* machines which assign states to the children of a first and then a. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

Acceptor is synonymous with *recognizer*. *Treeset* is synonymous with *tree language*.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

5.2.2 Definition

Definition 26 (DTFTA)**.** *A* Deterministic Top-down Finite-state Acceptor (DTFTA) *is a tuple* (Q, Σ_r, F, δ) *where*

- Q *is a finite set of states;*
- q_0 *is a initial state;*
- Σ *is a finite alphabet;*
- $\delta : Q \times \Sigma \times \mathbb{N} \to Q^*$ *is the transition function. Note the pre-image of* δ *is necessarily finite.*

The transition function takes a state, a letter, and a number n and returns a string of states. The idea is that the length of this output string should be n. Basically, when moving top-down, the states of the child sub-trees depend on these three things: the state of the parent, the label of the parent, and the number of children the parent has.

We use the transition function δ to define a new function $\delta^* : Q \times \Sigma^T \to Q^*$ as follows.

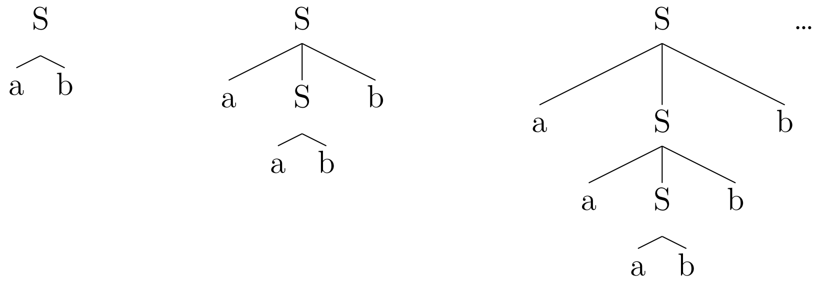
$$\begin{aligned} \delta^*(q, a[\lambda]) &= \delta(q, a, 0) \\ \delta^*(q, a[t_1 \cdots t_n]) &= \delta^*(q_1, t_1) \cdots \delta^*(q_n, t_n) \text{ where } \delta(q, a, n) = q_1 \cdots q_n \end{aligned} \tag{5.2}$$

As before, there are some important consequences to the formulation of δ^*. One is that δ^* is undefined on tree $a[t_1 \cdots t_n]$ if transition $\delta(q, a, n)$ does not return a string from Q^* of length n. (Also, I am abusing notation since δ^* is strictly speaking not the transitive closure of δ.)

Definition 27 (Treeset of a DTFTA)**.** *Consider some DTFTA $A = (Q, \Sigma, F, \delta)$ and tree $t \in \Sigma^T$. If $\delta^*(q_0, t)$ is defined and equals λ then we say A* accepts/recognizes *t. Otherwise A* rejects *t. Formally, the treeset recognized by A is $L(A) = \{t \in \Sigma^T \mid \delta^*(t) = \lambda\}$.*

The use of the 'L' denotes "Language" as treesets are traditionally referred to as *formal tree languages.*

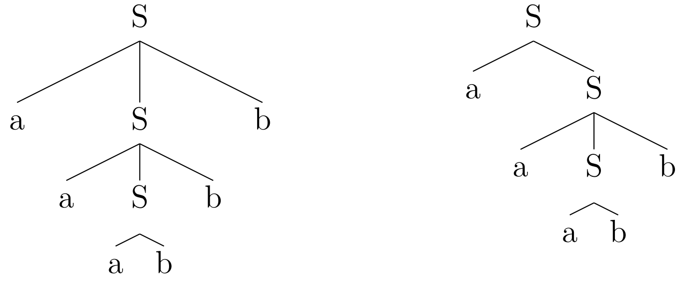
Example 12. Recall the example from last week which generates trees like



- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$

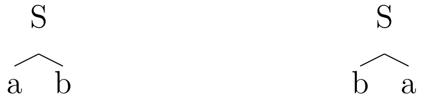
- $\delta(q_a, a, 0) = \lambda$
- $\delta(q_b, b, 0) = \lambda$
- $\delta(q_S, S, 3) = q_a q_S q_b$
- $\delta(q_S, S, 2) = q_a q_b$

Let us see how the acceptor A processes the two trees below as inputs.



Theorem 15. *Every treeset recognizable by a DTFTA is recognizable, but there are recognizable treesets which cannot be recognized by a DTFTA.*

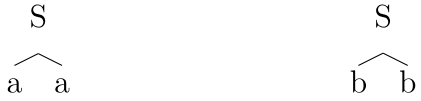
The following example helps show why this is the case. Consider the treeset T containing only the two trees shown below.



This is a recoginiable treeset because the DBFTA below recognizes exactly these two trees and no others.

- $Q = \{q, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$
- $\delta(\lambda, a) = q_a$
- $\delta(\lambda, b) = q_b$
- $\delta(q_a q_b, S) = q_S$
- $\delta(q_b q_a, S) = q_S$

Notice that this DBFTA fails on these two trees.



A DTFTA cannot recognize the trees in T without also recognizing the trees shown immediately above. This is because moving top down there can only be one value for $\delta(q_S, S, 2)$. Suppose it equals $q_1 q_2$. To recognize the first tree, we would also have to makes sure that $\delta(q_1, a, 0)$ and $\delta(q_2, b, 0)$ are defined. Similarly, to recognize the second tree, we would have to makes sure that $\delta(q_1, b, 0)$ and $\delta(q_2, a, 0)$ are defined. But it follows then that the aforementioned trees above are also recognized by this DTFTA. For instance the tree with two a leaves is recognized because both $\delta(q_1, a, 0)$ and $\delta(q_2, a, 0)$ are defined. Thus no DTFTA recognizes T.

5.2.3 Observations

- For every symbol $a \in \Sigma$ which can be leaf in a tree, you will need to define a transition $\delta(q, a, 0) = \lambda$.

- For every symbol $a \in \Sigma$ which can have n children, you will need to define a transition $\delta(q, a, n) = q_1 \cdots q_n$.

5.3 Properties of recognizable tree languages

Theorem 16 (Closure under Boolean operations)**.** *The class of recognizable tree languages are closed under union and intersection.*

The class of n-wide recognizable tree languages are closed under union, intersection, and complementation with respect to the $\Sigma^{\mathrm{T},n}$.

The proofs of these cases are very similar to the ones for finite-state acceptors over strings. For every DBFTA A recognizing a treeset T, it can be made *complete* by adding a sink state and transitions to it. Then product constructions can be used to establish closure under union and intersection. Closure under complement is established the same as before too: everything is the same except the final states are now the non-final states of A.

Theorem 17 (Minimal, determinstic, canonical form). *For every recognizable tree language T, there is a unique, smallest DBFTA A which recognizes T. That is, if DBFTA A' also recognizes T then there at least as many states in A' as there are in A.*

5.4 Connection to Context-Free Languages

Recognizable treesets are closely related to the derivation trees of context-free languages.

Theorem 18.

- *Let G be a context-free word grammar, then the set of derivation trees $D_T(G)$ is a recognizable tree language.*
- *There exists a recognizable tree language not equal to the set of derivation trees of any context-free language. Thus the class of derivation treesets of context-free word languages is a proper subset of the class of recognizable treesets.*
- *Let L be a recognizable tree language then $\mathtt{yield}(L)$ is a context-free word language.*

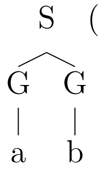
Each of these has a straightforward explanation.

For (1), the recognizable tree language which recognizes $D_T(G)$ for a CFG G can be constructed based on the rules of G. For each symbol a in N, the DBFTA should include $\delta(\lambda, a) = q_a$. And, for each rule $A \to B_1 \cdots B_n$ in R, the DBFTA should include $\delta(q_{B_1} \cdots q_{B_n}, A) = q_A$. That's it!

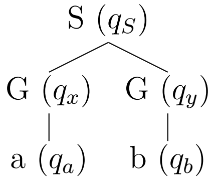
For (2), consider the DBFTA A shown below. The claim is that there is no CFG whose derivation language is exactly this recognizable treeset. The only tree in $L(A)$ is (t1), which is shown below A at left. Tree (t2) shows (t1) with the states A assigns to its subtrees.

$$
(A) \qquad
\begin{array}{rcl}
Q &=& \{q_S, q_a, q_b, q_x, q_y\} \\
\Sigma &=& \{S, G, a, b\} \\
F &=& \{q_S\} \\
\delta(q_x q_y, S) &=& q_S
\end{array}
\qquad
\begin{array}{rcl}
\delta(\lambda, a) &=& q_a \\
\delta(\lambda, b) &=& q_b \\
\delta(q_a, G) &=& q_x \\
\delta(q_b, G) &=& q_y
\end{array}
$$

(t1)

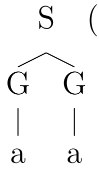


(t2)

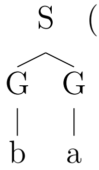


That no CFG can recognize this treeset follows from the fact that such any CFG G which includes the tree above will need to have the following rules: $S \rightarrow GG, G \rightarrow a, G \rightarrow b$. But then this G will not only generate (t1) above but also the derivation trees shown below.

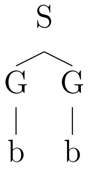
(t3)



(t4)



(t5)



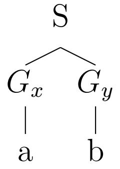
Thus $L(A) \neq D_T(G)$.

This example shows that the states of the DBFTA are more abstract than the labels on the nodes. The reason recognizable tree languages are more expressive than the derivation treesets of CFGs follows from this. The DBFTA uses states q_x and q_y to distinguish the subtrees bearing the label G. But the CFG cannot distinguish these trees in this way.

For (3), observe that we can write a CFG that generates the same stringset as the one above. For instance, we could write a CFG with the rule $S \rightarrow ab$.

More generally, though, we can always write a CFG that puts the state information into the nodes themselves. The trees in the derivation treeset for this CFG would be "structurally the same" as the trees in the recognizable treeset, but the labels on the nodes would be different. So they are not the same trees. In the example above for instance we can write a CFG with rules $S \rightarrow G_x G_y, G_x \rightarrow a, G_x \rightarrow b$. There is only one tree in this CFG's derivation treeset shown below.

(t6)



Importantly, (t6) is not the same as (t1). The inner nodes are labeled differently! So they are different trees. However, they only differ with respect to how the inner nodes are labeled, so it follows that the stringsets obtained by taking the `yield` of these trees are the same. This is the kind of argument used to show that the yield of any recognizable treeset is a context-free language.

Chapter 6

String Transducers

6.1 Deterministic Finite-state String Transducers

6.1.1 Orientation

This section is about deterministic finite-state transducers for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute the output from some input. The term *transducer* means this machine solves *transformation problems*: given an input object x, what object y is x transformed into? The term *string* means we are considering the transformation problem from strings to objects. So x is a string. As we will see, we can easily write transducers where y is a string, natural number, real number, or even a finite stringset! We will also see that DFSAs are a specific case of DFSTs.

However, we first define the output of the transformation to be a string. Then we will generalize it.

6.1.2 Definitions

Definition 28. *A* deterministic finite-state string-to-string transducer (DFST) *is a tuple* $T = (Q, \Sigma, \Delta, q_0, v_0, \delta, F)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the input alphabet*);*
- Δ *is a finite set of ouput symbols (*the output alphabet*);*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in \Delta^*$ *is the* initial string*;*
- δ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \Delta^*$*. It is called the* transition function*. If transition* $(q, a, r, v) \in \delta$ *it means that there is a transition from state* q *to state* r *reading letter* a *and writing string* v*. It will be helpful to refer to the "first" and*

"second" outputs of delta with δ_1 and δ_2 respectively. So for all $(q, a, r, v) \in \delta$, we have $\delta_1(q, a) = r$ and $\delta_2(q, a) = v$. These are are the "state" transition and the "output" transitions, respectively.

- *F is a function with domain Q and co-domain Δ^*. Let's call it the* final function*.*

For each transducer T, we can define a new function "process" $\pi : Q \times \Delta^* \times \Sigma^* \to \Delta^*$ as follows. Ultimately, $\pi(q, v, w)$ processes an input string w letter by letter from a given state q with a given output string v and returns an output string.

$$\begin{aligned} \pi(q, v, \lambda) &= v \cdot F(q) \\ \pi(q, v, aw) &= \pi\big((\delta_1(q, a),\ v \cdot \delta_2(q, a),\ w\big) \end{aligned} \tag{6.1}$$

Consider some DFST $T = (Q, \Sigma, \Delta, q_0, v_0, \delta, F)$ and string $u \in \Sigma^*$. Then $T(u) = \pi(q_0, v_0, u)$. We say T *transforms* u into v.

Definition 29. *The function defined by T is $\big\{(u, v) \in \Sigma^* \times \Delta^* \mid \pi(q_0, v_0, u) = v\big\}$.*

We write $T(u) = v$ iff $(u, v) \in T$.

Definition 30. *A string-to-string function is called* sequential *if there is a DFST that recognizes it.*

(Sequential functions are also often called subsequential functions. The nomenclature is unfortunate and different people have different opinions about it.) The important thing is that they are *deterministic* on the input and one should always check the definitions and not rely on names.

6.1.3 Exercises

Exercise 32. Let $\Sigma = \Delta = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that prefixes *pa* to all words.
2. Write a transducer that suffixes *ing* to all words.
3. Write a transducer that deletes word initial vowels.
4. Write a transducer that voices obstruents which occur immediately after nasals.
5. Write a transducer that deletes word final vowels. So $T(abba) = abb$ and $T(pie) = pi$.
6. Write a transducer that voices obstruents intervocalically.

Note that the transition function and the final function can be partial functions. In this case, the transducer is *incomplete* in the sense it is not defined for all inputs. As before, we will strive to make our sequential transducers describe total functions.

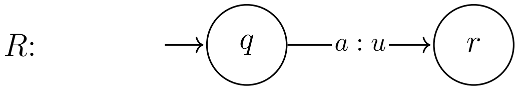
6.2 Some Closure Properties of Sequential functions

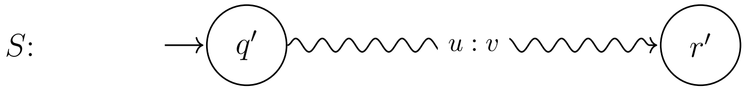
Theorem 19 (Closure under composition)**.** *If f, g are sequential functions then so is $f \circ g$.*

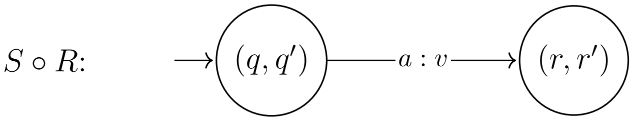
Closure under composition can also be shown using a product construction. The transition function in the product construction is not built however by following simultaneous paths in the two machines, but instead one after another.

Consider two sequential functions R and S and we are interested int he composition $S \circ R$, where R applies to first and S applies to the output of R. We treat the components of the image of the delta function separately with δ_1 and δ_2. Let $T_R = (Q_R, \Sigma, \Delta, q_{R0}, v_{R0}, F_R, \delta_{R1}, \delta_{R2})$ be the DFT recognizing R and let $T_S = (Q_S, \Delta, \Gamma, q_{S0}, v_{S0}, F_S, \delta_{S1}, \delta_{S2})$ be the DFT recognizing S.

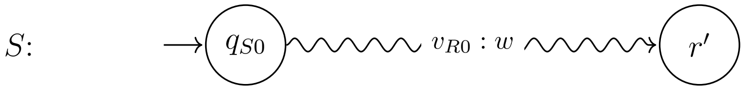
What happens when we are in $q \in Q_R$ and $q' \in Q_S$ and we read the letter a? Well, in T_R we reach state r and write string u. This string u becomes the input to T_S at state q'. It will traverse T_S reaching some state r' and writing some string v. This is shown visually below.





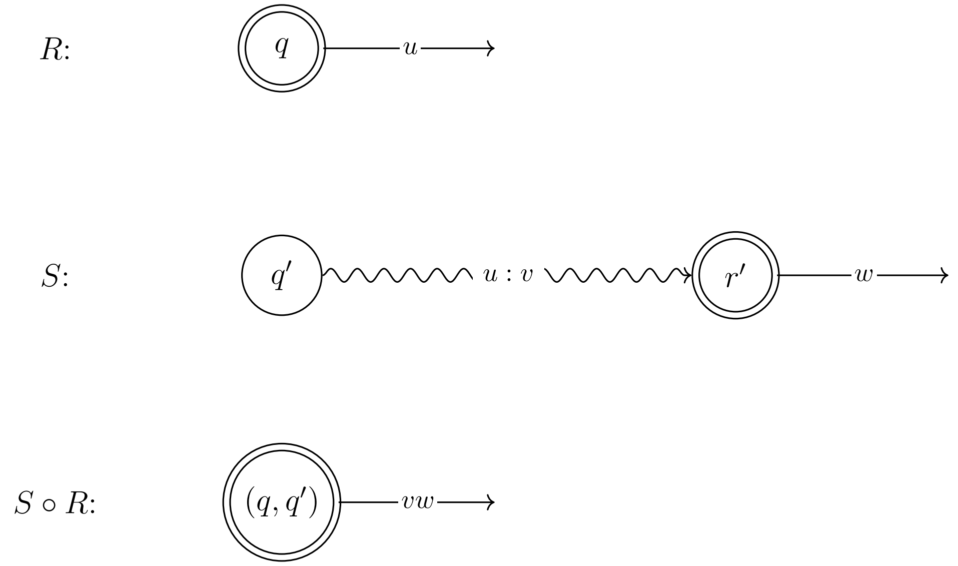


Similarly, what is the initial string v_0 for $T_{S \circ R}$? The initial string for T_R is v_{R0}. This will be processed by T_S from its initial state as shown below writing w and reaching state r'.



Therefore the initial string of $T_{S \circ R}$ will be $v_{S0} \cdot \delta^*_{S2}(q_{S0}, v_{R0})$. And the initial state of $T_{S \circ R}$ will be $(q_{0R}, \delta^*_{S1}(q_{S0}, v_{R0}))$.

The final output functions are handled similarly.



Putting this altogether, we construct $T = (Q, \Sigma, \Gamma, q_0, v_0, F, \delta_1, \delta_2)$ recognizing $S \circ R$ as follows.

- $Q = Q_R \times Q_S$.
- $q_0 = (q_{0R}, \delta^*_{S1}(q_{S0}, v_{R0}))$.
- $v_0 = v_{S0} \cdot \delta^*_{S2}(q_{S0}, v_{R0})$.
- $\delta_1((q, q'), a) = (\delta_{R1}(q, a), \delta^*_{S1}(q', \delta_{R2}(q, a)))$
- $\delta_2((q, q'), a) = \delta^*_{S2}(q', \delta_{R2}(q, a)))$
- $F((q, q')) = \delta^*_{S2}(q', F_R(q)) \cdot F_S(\delta^*_{S1}(q', F_R(q)))$

Theorem 20 (Minimal canonical form)**.** *For every sequential string-to-string function f, it is possible to compute a DFST T such that T is equivalent to f and no other DFST T' equivalent to f has fewer states than T.*

The minimal cananoical form result is due to Choffrut (see his 2003 survey).

Sequential functions are not closed under union. This is because they are functions and so each input has a unique output. I can't find a reference that sequential transducers are closed under intersection. That's a good exercise!

6.3 Generalizing sequential functions with monoids

A *monoid* is a mathematical term which means any set which is closed under some associative binary operation with an identity. So if $(S, *, 1)$ is a monoid then for all $x, y \in S$:

1. is the case that $x * y$ is in S too (Closure under $*$)
2. $(x * y) * z = x * (y * z)$ (Associativity)
3. $1 * x = x * 1 = x$ (1 is the identity)

It is typical to refer to $*$ as "times", "multiplication" or as a product. It is also typical to refer to 1 as the "identity", "unit" or "one".

Σ^* is closed under the binary operation of concatenation. Also the empty string behaves like the identity with respect to concatenation. So $(\Sigma^*, \ \cdot\ , \lambda)$ is a monoid. As we processed the input string, we moved from state to state and updated the ouput value by concatenating strings along the output transitions. We can do the same thing and update the output value using some other product from another monoid.

Example 13. Here are some examples.

1. $(\{\texttt{True}, \texttt{False}\},\ \wedge,\ \texttt{True})$. Boolean values and conjunction. This monoid shows the membership problem is a special case of the transformation problem.
2. $(\mathbb{N},\ +,\ 0)$. Natural numbers and addition. Useful for counting!
3. $([0,1],\ \times,\ 1)$. The real unit interval and multiplication. Useful for probabilities!
4. $(\mathbb{R},\ \times,\ 1)$. All real numbers and multiplication.
5. (FIN, $\cdot$, $\{\lambda\}$) where FIN is the class of finite stringsets and $(\cdot)$ is concatenation of stringsets.
    - FIN $= \{S \mid \exists n \in \mathbb{N}$ with $|S| = n\}$
    - $S_1 \cdot S_2 = \{u \cdot v \mid u \in S_1, v \in S_2\}$.
6. There are many others!

This means we can generalize DFSTs to transducers which output elements from any monoid.

Definition 31. *A* generalized sequential transducer (GST) *is a tuple* $T = (Q, \Sigma, M, q_0, v_0, \delta, F)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the input alphabet*);*
- $(M, *, 1)$ *is a monoid*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in M$ *is the* initial *value;*
- δ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \mu$*. It is called the* transition function*. As before, from this we derive* $\delta_1 : Q \times \Sigma \to Q$ *and* $\delta_2 : Q \times \Sigma \to \mu$ *to be the state and output transition functions.*
- F *is a function with domain* Q *and co-domain* M*. Let's call it the* final function.

The process function π is the same except we replace concatenation of the outputs with the monoid operator $*$.

$$\begin{array}{rcl} \pi(q, v, \lambda) & = & v * F(q) \\ \pi(q, v, aw) & = & \pi\big((\delta_1(q,a),\ v * \delta_2(q,a),\ w\big) \end{array} \tag{6.2}$$

Then, the definition of the function computed by the transducer is identical to what was written formerly.

That's it!

6.3.1 Exercises

Exercise 33. Let $\Sigma = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that counts how many NC (nasal-consonant) sequences occurs in the input word.
2. Write a transducer that optionally voices obstruents which occur immediately after nasals. So for in input like *anta* the output should be the set {*anda, anta*}. (Hint: use the FIN monoid).

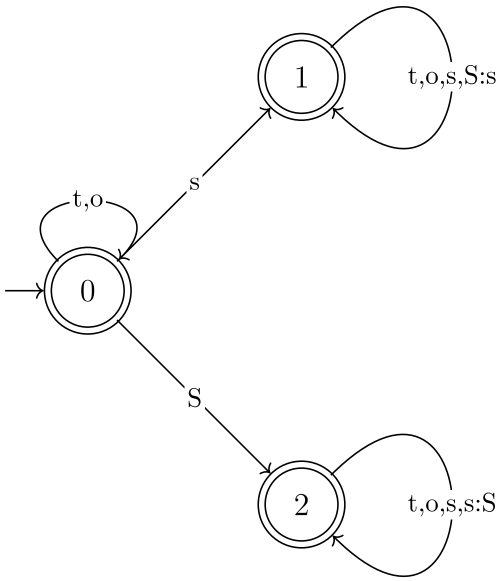
6.4 Learning more

Unfortunately, the material on deterministic transducers has yet to make its way into standard textbooks. Standard textbooks in computer science discuss non-deterministic finite-state transducers if they discuss transducers at all. Within computational linguistics where transducers are widely used, non-determinstic ones are the norm. See, for instance Roche and Schabes (1997); Beesley and Kartunnen (2003); Roark and Sproat (2007) and Jurafsky and Martin (2008a). A notable exception is work by Mehyar Mohri (1997; 2005). The most textbook-like discussion of sequential functions I am aware of comes from Lothaire (2005, chapter 1).

My interest in sequential transducers stems from three interrelated facts. First, they appear sufficient to decribe morpho-phonologial generalizations in natural language (Jardine, 2016) and subsequent discussions. So the extra power that comes with non-deteminsitic transducers appears unnecessary in this domain. Second, sequential transducers have canonical forms (Choffrut's theorem), but non-deterministic ones do not. Third, the class of sequential transducers can be learned from examples, unlike the class of non-deterministic transducers (de la Higuera, 2010, chapter 18).

6.5 Left and Right Sequential Transducers

Below is a sequential transducer for progressive sibilant harmony. This means later sibilants agree in anteriority with the first sibilant in the word. The alphabet here is simply {s,S,t,o} with {s,S} signifying the two classes of sibilants and {t} other consonants, and {o} the vowels. We assume the initial value is λ and for all q the final function maps q to λ. In the diagram, $a : b$ means a is read as input and b is ouput. If there is no colon and only a, then it means a is read as input and a is written as output.



Many examples of sibilant harmony in natural language are not progressive. They are in fact regressive. For instance here are some wonderfully long words in Samala (Chumash) (Applegate 1972). The underlying form is on the left and the surface form is on the right. There are some alternations in the vowels which we ignore here.

/ha-s-xintila-waʃ/	[haʃxintilawaʃ]	‘his former Indian name’
/k-su-kili-mekeken-ʃ/	[kʃuk’ilimekeket͡ʃ]	‘I straigten myself up’
/k-su-al-puj-ʊn-ʃaʃi/	[kʃalpujat͡ʃɨʃi]	‘I get myself wet’
/s-taja-nowon-waʃ/	[ʃtojowonowaʃ]	‘it stood upright’

Exercise 34. Explain why regressive sibilant harmony *cannot* be modeled with sequential transducers.

There is a straightforward way to address this issue. The transducers we are using process strings from left-to-right. However, transducers could also process strings from right-to-left. If we use the transducer above to process the Samala strings from right to left, we can model regressive sibilant harmony. The example below shows /stototooS/↦[StototooS].

$$\begin{array}{ccccccccccccccccccccc}
 & & \text{s} & & \text{t} & & \text{o} & & \text{t} & & \text{o} & & \text{t} & & \text{o} & & \text{o} & & \text{S} & & \\
\leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 3 & \leftarrow & 1 & \leftarrow \\
 & & \text{S} & & \text{t} & & \text{o} & & \text{t} & & \text{o} & & \text{t} & & \text{o} & & \text{o} & & \text{S} & &
\end{array}$$

If a transducer processes the string left-to-right, it is called a *left sequential* transducer. If it processes it right-to-left it is called a *right sequential* transducer.

Theorem 21. *Left sequential functions and right sequential functions are incomparable; that is, there are functions that are both left and right sequential; neither left nor right sequential; left but not right sequential; and right but not left sequential.*

Progressive sibilant harmony is a case in point. It is left sequential but not right sequential. On the other hand, regressive sibilant harmony is right sequential, but not left sequential. The identity function is both left and right sequential. Can you think of a a function which is neither left nor right sequential?

The recursive data structure we are using for strings is inherently left-to-right because the outermost element in the list structure is on the left. If we were to define lists so that the outermost element of the list structure was on the right, it would become natural to process strings right-to-left.

The Haskell implementation of strings is inherently left-to-right because the outermost element in the list structure is on the left. If we were to define lists so that the outermost element of the list structure was on the right, it would become natural to process strings right-to-left.

An easy way to simulate a right sequential transducer with the lists we have in Haskell is to do the following.

1. Implement left sequential transducers as we have done.
2. Before processing a string w, reverse it.
3. Then reverse the output of the transducer.

In other words, `transduce t w` will process the string left-to-right. However, the function `reverse (transduce t (reverse w))` will simulate t processing w right-to-left.

6.6 Non-deterministic Finite-state String Transducers

There are many ways to define non-deterministic finite-state string transducers. We follow Lothaire (2005).

Definition 32. *A* non-deterministic finite-state string-to-string transducer (NFST) *is a tuple* $T = (Q, \Sigma, \Delta, I, F, \delta)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the input alphabet*);*
- Δ *is a finite set of ouput symbols (*the output alphabet*);*
- $I \in Q$ *is a set of* initial *states;*
- $F \in Q$ *is a set of* final *states;*

- *$\delta \subseteq Q \times \Sigma^* \times \Delta^* \times Q$. It is called the* transition relation. *If transition $(q, u, v, r) \in \delta$ it means that there is a transition from state q to state r reading u and writing v.*

Definition 33. *For each transducer T, a* valid path *in T denoted π is a sequence of transitions $(q_0, u_0, v_0, r_0)(q_1, u_1, v_1, r_1) \dots (q_n, u_n, v_n, r_n)$ such that $q_0 \in I, r_n \in F$ and $\forall i$ if $1 \leq i \leq n$ then $(q_i, u_i, v_i, r_i) \in \delta$ and if $i < n$ then $r_i = q_{i+1}$. For each valid path π let its* input projection *be the string $\pi_{in} = u_0 u_1 \dots u_n$ and its* output projection *be the string $\pi_{out} = v_0 v_1 \dots v_n$*

Then the relation recognized by T is

$$\{(u, v) : \exists \textit{ valid } \pi \textit{ in } T, u = \pi_{in}, v = \pi_{out}\}$$

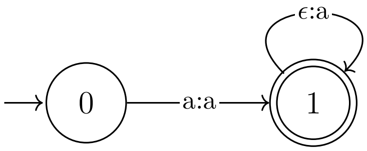
Relations recognized by NFSTs are called *rational relations*. As Lothaire (p. 42) writes. "A transducer is *literal* if for each edge the input label and the output label are letters or the empty word. It is not difficult to show that any transducer can be replaced by a literal one."

Theorem 22. *The pre-image of a rational relation is a regular stringset. The image of a rational relation is a regular stringset.*

Proof For any NFT T, mapping each transition (q, a, b, r) to (q, a, r) yields a NFA which recognizes the input projections of valid paths in T. Similarly, mapping each transition (q, a, b, r) to (q, b, r) yields a NFA which recognizes the output projections of valid paths in T. □

Theorem 23. *Left and right sequential functions are proper subclasses of the rational relations.*

One reason why this is true is that epsilon transitions permit non-deterministic transducers to recognize *infinite relations* not functions. For example, the non-deterministic acceptor below relates input a to every element in $\{a, aa, aaa, \dots\}$.



This is not possible for sequential transducers with the Σ^* monoid or the finite language monoid. However, it should be possible with the "Regular language" monoid where the outputs of the transitions are actually NFA that get "concatenated."

However there are rational *functions* that cannot be described with left or right sequential transducers. An example from Heinz and Lai (2013) is Sour Grapes harmony, which is illustrated in the table below wiht respect to other logically possible harmony patterns (was Table 3, p. 57 in Heinz and Lai (2013)). See Jardine (2016) and McCollum *et al.* (2020) for more discussion regarding the typology of phonological patterns like this.

	w	PH(w)	RH(w)	DR(w)	SG(w)	MR(w)
a.	$/+--/$	$[+++]$	$[---]$	$[+++]$	$[+++]$	$[---]$
b.	$/-++/$	$[---]$	$[+++]$	$[+++]$	$[---]$	$[+++]$
c.	$/---/$	$[---]$	$[---]$	$[---]$	$[---]$	$[---]$
d.	$/-+-/$	$[---]$	$[---]$	$[+++]$	$[---]$	$[---]$
e.	$/+-\boxminus/$	$[++\boxminus]$	$[--\boxminus]$	$[++\boxminus]$	$[+-\boxminus]$	$[--\boxminus]$
f.	$/+\ominus-/$	$[+\ominus+]$	$[-\ominus-]$	$[+\ominus+]$	$[+\ominus+]$	$[-\ominus-]$

Table 6.1: Example mappings of underlying forms (w) given by progressive harmony (PH), regressive harmony (RH), dominant/recessive harmony (DR), sour grapes harmony (SG), and majority rules harmony (MR). Symbols [+] indicates a [+F] vowel and [−] indicates a [−F] vowel where "F" is the feature harmonizing. Symbols [$\boxminus$] and [$\ominus$] are [−F] vowels that are opaque and transparent, respectively.

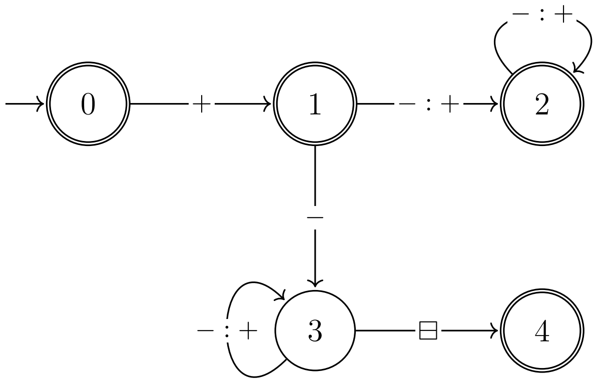


Figure 6.1: A non-deterministic transducer which recognizes a fragment of SG harmony.

Part of a NSFT recognizing Sour Grapes is shown in the Figure above (was Figure 3, p. 58 in Heinz and Lai (2013)).

Finally, the definition above defined rational relations in terms of paths, and not the recursive 'process' function we have used elsewhere. The definition is fine as far as it goes, but it does not tell us how to compute the output for a given input. So given a NFST T how do we compute $T(w)$? As mentioned above, this is more complicated because there may be infinitely many strings which w relates to.

The answer is to use composition ($\circ$). If we want to compute $T(w)$, we first construct a NFST W recognizing the relation $\{(w, w)\}$. Then we compute $T \circ W$. This yields the relation $\{(w, v) : v \in T(w)\}$. Thus the output projection of this transducer is the regular stringset, which is all and only those strings related to w in the rational relation T.

Below is the definiton of composition for literal transducers.

Definition 34. *Let* $T_R = (Q_R, \Sigma, \Delta, I_R, F_R, \delta_R)$ *be the NFST recognizing* R *and let* $T_S =$

$(Q_S, \Delta, \Gamma, I_S, F_S, \delta_S)$ be the NFST recognizing S. Then $T = (Q, \Sigma, \Gamma, I, F, \delta)$ recognizes $S \circ R$ where:

- $Q = Q_R \times Q_S$.
- $I = I_R \times I_S$.
- $F = F_R \times F_S$.
- $\delta = \Big\{ \big((p, r), a, c, (q, s)\big) : (p, a, b, q) \in T_R, (r, b, c, s) \in T_S \Big\}$
  $\cup \Big\{ \big((p, r), a, \lambda, (q, r)\big) : (p, a, \lambda, q) \in T_R, r \in Q_S \Big\}$
  $\cup \Big\{ \big((p, r), \lambda, c, (p, s)\big) : (r, \lambda, c, s) \in T_S, p \in Q_R \Big\}$

Exercise 35.

1. Write a NFST T which deletes sequences of word final consonants. Compute $T(abab)$, $T(abbb)$, $T(baba)$.

2. Write a NFST T which optionally voices obstruents after nasals. Compute $T(anta)$, $T(antanta)$.

Chapter 7

Tree Transducers

7.1 Deterministic Bottom-up Finite-state Tree Transducers

7.1.1 Orientation

This section is about deterministic bottom-up finite-state tree transducers. The term *finite-state* means that the amount of memory needed in the course of computation is independent of the size of the input. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *transducer* means this machine solves *transformation problem*: given an input object x, what object y is x transformed into? The term *tree* means we are considering the transformation problem from trees to trees. The term *bottom-up* means that for each node a in a tree, the computation transforms the children of a node before transforming the node. This contrasts with *top-down* transducers which transform the children after transforming their parent. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentionepd below.

7.1.2 Definitions

Recall the definition of trees with a finite alphabet Σ. All such trees belonged to the treeset Σ^T. In addition to this, we will need to define a new kind of tree which has *variables* in the leaves. I will call these trees *Variably-Leafed*. We assume a countable set of variables X containing variables $x_1, x_2, \ldots$.

Definition 35 (Variably-Leafed Trees)**.**

Base Cases (Σ): *For each $a \in \Sigma$, $a[\,]$ is a tree.*
Base Cases (X): *For each $x \in X$, $x[\,]$ is a tree.*

Inductive Case: *If $a \in \Sigma$ and $t_1 t_2 \dots t_n$ is a string of trees of length n then $a[t_1 t_2 \cdot \dots t_n]$ is a tree.*

Let $\Sigma^T[X]$ denote the set of all variably-leafed trees of finite size using Σ and X.

Note that $\Sigma^T \subsetneq \Sigma^T[X]$. In the tree transducers we write below, the variably-leafed trees will play a role in how outputs are constructed as well as the the intermediate stages of the transformation.

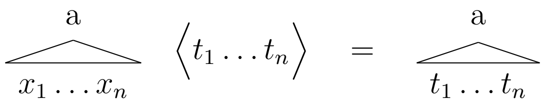
With this definition in place, we can define our first tree transducer.

Definition 36 (DBFTA). *A* Deterministic Bottom-up Finite-state Acceptor (DBFTT) *is a tuple (Q, Σ, F, δ) where*

- *Q is a finite set of states;*
- *Σ is a finite alphabet;*
- *$F \subseteq Q$ is a set of* accepting *(*final*) states; and*
- *$\delta : Q^* \times \Sigma \to Q \times \Sigma^T[X]$ is* the transition function. The pre-image of δ must be finite. *This means we can write it down—for example, as a list.*

  If transition $(\vec{q}, a, r, t) \in \delta$ it means that if the children of node a have been assigned states $\vec{q}$ then the state r will be assigned to a and the tree t will be the output employed at this stage in the derivation. It will be helpful to refer to the "first" and "second" outputs of delta with δ_1 and δ_2 respectively. So for all $(\vec{q}, a, r, v) \in \delta$, we have $\delta_1(\vec{q}, a) = r$ and $\delta_2(\vec{q}, a) = t$. These are are the "state" transition and the "output" transitions, respectively.

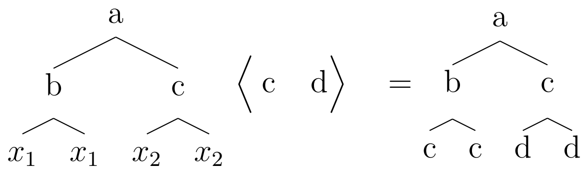
The key to understanding how a tree-to-tree transducer is defined is to understand how variables are used to rewrite and expand output trees. If $t_1 \dots t_n$ is a list of trees and $t_x \in \Sigma^T[X]$ is a variably leafed tree with variables $x_1, \dots x_n$ then let $t_x\langle t_1 \dots t_n\rangle = t \in \Sigma^T$ obtained by replacing each variable x_i in t_x with t_i. Here is a visualization of this notation.



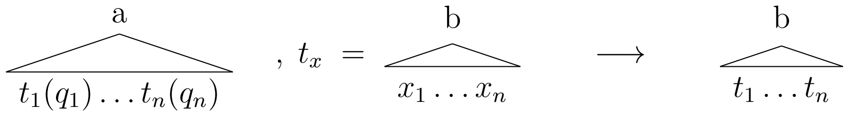
Example 14.

$$a\big[\ b[x_1\ x_1]\ \ c[x_2\ x_2]\ \big]\Big\langle t_1 = c[\ ], t_2 = d[\ ]\Big\rangle = a\big[\ b[c[\ ]\ c[\ ]]\ \ c[d[\ ]\ d[\ ]]\ \big]$$

Visually, we are taking the tree shown below and substituting t_1 for x_1 and t_2 for x_2.



Now we explain how substitution is used in the process of transducing a tree into another. Given $\delta_2(q_1q_2\dots q_n, a) = t_x$ and a tree $a[t_1t_2\dots t_n]$ with states $q_1q_2\dots q_n$, respectively, then the output tree will be the one obtained by replacing each x_i with t_i in t_x. A schematic of this is shown below.



Then we can define a function "process" $\pi : \Sigma^T \rightarrow Q \times \Sigma^T$ which will process the tree and produce its output. It is defined recursively as follows.

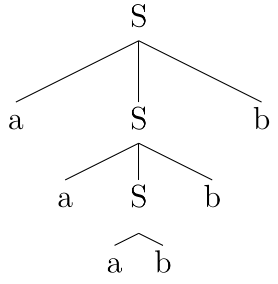
$$\begin{aligned} \pi(a[\,]) &= \big(\delta_1(\lambda, a), \delta_2(\lambda, a)\big) \\ \pi(a[t_1\cdots t_n]) &= (q, t) \text{ where} \\ &\qquad q = \delta_1\big(q_1\cdots q_n, a\big) \text{ and} \\ &\qquad t = \delta_2\big(q_1\cdots q_n, a\big)\langle s_1\cdots s_n\rangle \text{ and} \\ &\qquad (q_1, s_1)\cdots(q_n, s_n) = \pi(t_1)\cdots\pi(t_n) \end{aligned} \tag{7.1}$$

Definition 37 (Tree-to-tree function of a DBFTT)**.** *The function defined by the transducer T is $\big\{(t, s) \mid t, s \in \Sigma^T, \pi(t) = (q, s), q \in F\big\}$. If (t, s) belongs to this set, we say T* transduces *t to s and write $T(t) = s$.*

Example 15. Consider the transducer T constructed as follows.

- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $F = \{q_S\}$
- $\delta_1(\lambda, a) = q_a$
- $\delta_1(\lambda, b) = q_b$
- $\delta_1(q_aq_b, S) = q_S$
- $\delta_1(q_aq_Sq_b, S) = q_S$
- $\delta_2(\lambda, a) = a[\,]$
- $\delta_2(\lambda, b) = b[\,]$
- $\delta_2(q_aq_b, S) = S[x_2x_1]$
- $\delta_2(q_aq_Sq_b, S) = S[x_3x_2x_1]$

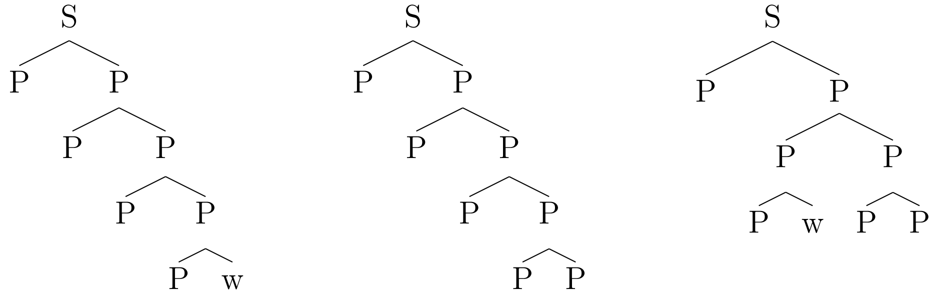
Let us work out what T transforms the tree below into.



Example 16. Consider the transducer T constructed as follows. We let $Q = \{q_w, q_p, q_S\}$, $\Sigma = \{w, P, S\}$, and $F = \{q_S\}$. The transition and ouput functions are given below.

- $\delta(\lambda, P) = q_p, P[\,]$
- $\delta(\lambda, w) = q_w, w[\,]$
- $\delta(q_pq_p, P) = q_p, P[x_1x_2]$
- $\delta(q_wq_p, P) = q_w, P[x_1x_2]$
- $\delta(q_pq_w, P) = q_w, P[x_1x_2]$
- $\delta(q_pq_w, S) = q_S, S[\ w[\,]\ S[x_1x_2]\ ]$
- $\delta(q_wq_p, S) = q_S, S[\ w[\,]\ S[x_1x_2]\ ]$
- $\delta(q_pq_p, S) = q_S, S[x_1x_2]$

Let us work out how T transforms the trees below.



7.2 Deterministic Top-down Finite-state Tree Transducers

7.2.1 Orientation

This section is about deterministic bottom-up finite-state tree transducers. The term *finite-state* means that the amount of memory needed in the course of computation is independent of the size of the input. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *transducer* means this machine solves *transformation problem*: given an input object x, what object y is x transformed into? The term *tree* means we are considering the transformation problem from trees to trees. The term *top-down* means that for each node a in a tree, the computation transforms the node before transforming its children. This contrasts with *bottom-up* transducers which transform the children before transforming their parent. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

7.2.2 Definitions

As before, we use variably leafed trees $\Sigma^T[X]$.

Definition 38 (DTFTT)**.** *A* Deterministic Top-down Finite-state Acceptor (DTFTT) *is a tuple* $(Q, \Sigma_r, q_0, \delta)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite alphabet;*
- $q_0 \in Q$ *is the initial state; and*
- $\delta : Q \times \Sigma \times \mathbb{N} \to Q^* \times \Sigma^T[X]$ *is* the transition function*. As before, we will derive "state" and "output" transitions, and notate them with* δ_1 *and* δ_2*, respectively. So for all* $(q, a, \vec{r}, t) \in \delta$*, we have* $\delta_1(q, a) = \vec{r}$ *and* $\delta_2(q, a) = t$*.*

We also define the "process" function $\pi : Q \times \Sigma^T \to \Sigma^T$ which will process the tree and produce its output. It is defined as follows.

$$\begin{aligned}\pi(q, a[\,]) &= \delta_2(q, a, 0)\\ \pi(q, a[t_1 \cdots t_n]) &= \delta_2(q, a, n)\big\langle \pi(q_1, t_1) \cdots \pi(q_n, t_n)\big\rangle\\ &\quad \text{where } q_1 \cdots q_n = \delta_1(q, a, n)\end{aligned} \tag{7.2}$$

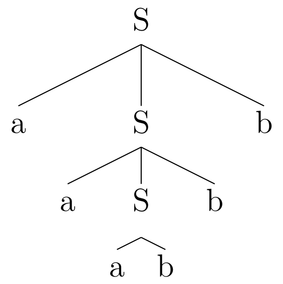
Intuitively, δ_2 transforms the root node into a variably leafed tree. The variables are replaced with the children of the root node. These children are also trees with states assigned by δ_1. Then π transforms each tree-child as well.

Definition 39 (Tree-to-tree function of a DTFTT)**.** *The function defined by the transducer* T *is* $\big\{(t, s) \mid t, s \in \Sigma^T, \pi(q_0, t) = s\big\}$*. If* (t, s) *belongs to this set, we say* M transduces t *to* s *and write* $T(t) = s$*.*

Example 17. Consider the transducer T constructed as follows.

- $Q = \{q, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$
- $\delta_1(q, a, 0) = \lambda$
- $\delta_1(q, b, 0) = \lambda$
- $\delta_1(q_S, S, 3) = q q_S q$
- $\delta_1(q_S, S, 2) = qq$
- $\delta_2(q, a, 0) = a[\,]$
- $\delta_2(q, b, 0) = b[\,]$
- $\delta_2(q, S, 3) = S[x_3 x_2 x_1]$
- $\delta_2(q, S, 2) = S[x_2 x_1]$

Let see how T transforms the tree below.



Exercise 36. Recall the "wh-movement" example from before. Explain why this transformation *cannot* be computed by a deterministic top-down tree transducer.

7.3 Theorems about Deterministic Tree Transducers

Theorem 24 (composition closure)**.** *The class of deterministic bottom-up transductions is closed under composition, but the class of top-down deterministic transductions is not.*

Theorem 25 (Incomparable)**.** *The class of deterministic bottom-up transductions is incomparable with the the class of top-down deterministic transductions.*

This theorem is based on the same kind of examples which separated the left and right sequential functions. Let relations $U = \{(f^n a, f^n a) \mid n \in \mathbb{N}\} \cup \{(f^n b, g^n b) \mid n \in \mathbb{N}\}$ and $D = \{(ff^n a, ff^n a) \mid n \in \mathbb{N}\} \cup \{(gf^n a, gf^n b) \mid n \in \mathbb{N}\}$. U is recognized by a DBFTT but not any DTFTT and D is recognized by a DTFTT but not any DBFTT.

Chapter 8

Strict Locality

8.1 Strictly Local Languages

A language L is k-Strictly Local (SL) if we can determine whether a string x belongs to L by checking each of the k-long substrings in x (Rogers and Pullum, 2011; Rogers *et al.*, 2013; Rogers and Lambert, 2019). If every one is well-formed then $x \in L$. If any k-long substring is ill-formed then $x \notin L$. We can imagine examining each of the local-substructures, checking to see if it is forbidden or not. The whole structure is well-formed only if each local substructure is. For example, consider the string *abab*. If we fix a diameter of 2, we have to check these substrings.

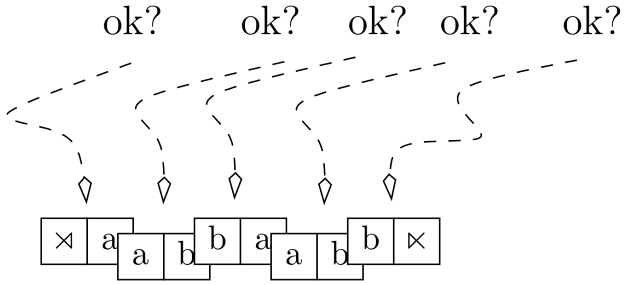


Figure 8.1: Checking substrings of length 2.

Formally, k-SL grammars are defined as follows. For any Σ, define $\mathtt{factor}_k : \Sigma^* \rightarrow \wp(\Sigma^{\leq k})$ as $\mathtt{factor}_k(x) = \{v \in \Sigma^k : \exists u, w \in \Sigma^*, x = uvw\}$ whenever $k \leq |x|$ and let $\mathtt{factor}_k(x) = \{x\}$ otherwise. We lift the domain of $\mathtt{factor}_k$ from strings to set of strings as follows.

$$\mathtt{factor}_k(L) = \bigcup_{x \in L} \mathtt{factor}_k(w)$$

Let $\rtimes, \ltimes$ denote the right and left word boundaries, respectively.

Definition 40. *A k-SL grammar is a subset of*

$$\mathtt{factor}_k\big(\{\rtimes\}\Sigma^*\{\ltimes\}\big)$$

The language of a k-SL grammar G is $L(G) = \{w \in \Sigma^* : \texttt{factor}_k(\rtimes w \ltimes) \subseteq G\}$.

These grammars are sometimes called positive grammars because the grammar is defined as the set of well-formed k-long substrings.

Exercise 37. Define k-SL grammars and languages in terms of forbidden k-long substrings.

8.1.1 Logic

Negative SL Grammars can also be expressed logically as the Conjunctions of Negative Literals. Formally, we define a logical language as follows.

- Base cases: For each $w \in \Sigma^{k-1}$, $\neg \rtimes w$ and $\neg w \ltimes$ are literals. For each $w \in \Sigma^k$, $\neg w$ is a literal. These literals are interpreted as follows.

$$[\![\neg \rtimes w]\!] = \overline{w\Sigma^*}; \quad [\![\neg w \ltimes\ ]\!] = \overline{\Sigma^* w}; \quad [\![\neg w]\!] = \overline{\Sigma^* w \Sigma^*}$$

- Inductive case: if $R_1, \ldots, R_n$ are expressions so is $\bigwedge_{1 \leq i \leq n} R_i$, which is interpreted as $\bigcap_{1 \leq i \leq n} [\![R_i]\!]$.

8.1.2 Suffix Substitution Closure

Abstract characterizations of classes of formal languages are independent of logical formulas, grammars, and automata. They provide laws of *inference* for learning and they provide ways to show certain stringsets do NOT belong to the class.

The theorem below establishes a set-based characterization of SL stringsets independent of any grammar, scanner, or automaton.

Theorem 26 (k-Local Suffix Substitution Closure)**.** *For all* $L \subseteq \Sigma^*$, $L \in$ *SL iff there exists* k *such that for all* $u_1,\ v_1,\ u_2,\ v_2,\ x \in \Sigma^*$ *it is the case that if* $u_1xv_1, u_2xv_2 \in L$ *and* $|x| = k-1$ *then* $u_1xv_2 \in L$.

As illustrated in Figure 8.2, the theorem provides a law which simultaneously provides a basis for *inference* and provides a method for establishing non-SL$_k$ stringsets.

$$\begin{array}{r|c|ll}
u_1 & \sigma_1 \cdots \sigma_{k-1} & v_1 & \in L \\
u_2 & \sigma_1 \cdots \sigma_{k-1} & v_2 & \in L \\
\hline
u_1 & \sigma_1 \cdots \sigma_{k-1} & v_2 & \in L
\end{array}$$

Figure 8.2: Suffix Substitution Closure

Exercise 38.

1. Consider a Strictly 2-Local stringset L which contains the words aa and ab. Using Suffix Substitution Closure, explain what other words must be in L.

2. Consider the constraint $a \ldots a$. Show this is not SL_k for any k.

3. Consider the constraint EVEN-a. Show this is not SL_k for any k.

8.1.3 Cognitive Interpretation

Rogers *et al.* (2013) argue that this Strictly Local languages have a cognitive interpretation as follows.

- Any cognitive mechanism that can distinguish member strings from non-members of a SL_k stringset must be sensitive, at least, to the length k blocks of consecutive events that occur in the presentation of the string.

- If the strings are presented as sequences of events in time, then this corresponds to being sensitive, at each point in the string, to the immediately prior sequence of $k-1$ events.

- Any cognitive mechanism that is not sensitive to the length k blocks of consecutive events that occur in the presentation of the string will be unable to recognize some SL_k stringsets.

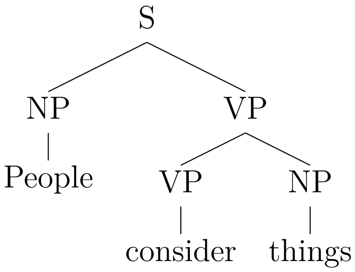
8.1.4 Relationship to Regular Languages

For every regular language L, let A_L be a finite-state acceptor recognizing L. The set of valid paths of A is a 2-SL language. While L is a subset of Σ^*, the path language of A_l is a subset of $(Q \times \Sigma \times Q)^*$. It follows that every regular language $L \in \Sigma^*$ is the homomorphic image of a SL-2 language $P \in \Delta^*$ with $\Sigma \subseteq \Delta$. The homomorphism simply erases the state information in the path language of A_L.

8.2 Strictly Local Tree Languages

A positive k-SL grammar for a string language can be thought of as a set of tiles. The strings in the language are the ones obtained by overlapping these tiles. Similarly, a positive k-SL grammar for a string language is also a set of tiles. However, the tiles now represent treelets of depth k.

For example, consider the tree below



[INSERT FIGURE HERE]

Figure 8.3: Checking treelets of length 2.

We follow the presentation in Ji and Heinz (2020).

8.2.1 Context Free Language Derivation Trees are 2-SL

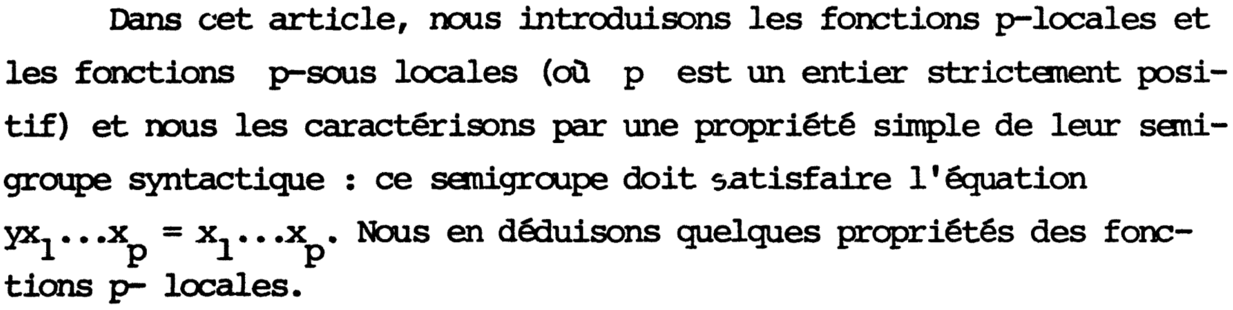
Theorem 27. *Consider any CFG grammar G. The derivation tree language of G is 2-SL.*

Exercise 39. Prove this theorem.

8.3 Strictly Local Functions over Strings

8.3.1 Local functions

Vaysse 1986 studied p-local functions. He writes:



> Dans cet article, nous introduisons les fonctions p-locales et les fonctions p-sous locales (où p est un entier strictement positif) et nous les caractérisons par une propriété simple de leur semigroupe syntactique : ce semigroupe doit satisfaire l'équation $yx_1...x_p = x_1...x_p$. Nous en déduisons quelques propriétés des fonctions p- locales.

8.3.2 Input Strictly Local Functions

The following is an excerpt from Heinz (2018, §6.2.1):

Excerpt Begins.

Input Strictly Local function generalize the notion of Strictly Local stringset. Recall the Strictly Local stringsets are Markovian in nature: the well-formedness of a string can be determined by examining the substrings of length k. Equivalently, this means that the well-formedness of any position in the string can be determined by checking the $k-1$ previous symbols. This is illustrated in Figure 8.4, for the case where $k=2$.

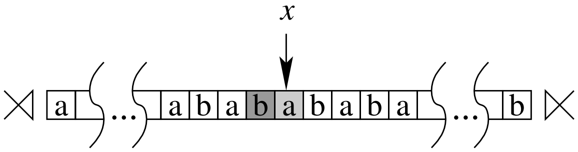


Figure 8.4: A schematic illustrating the Markovian nature of Strictly k-Local stringsets. Each element x of a string belonging to a strictly 2-local stringset depends only on the previous element. In other words, the lightly shaded cell only depends on the darkly shaded cell.

Input Strictly Local functions are similarly Markovian. The idea is that every element in the input string corresponds to a *string* of symbols in the output string. For any input symbol x its output string u will only depend on x and the previous $k-1$ elements of x in the input string. Figure 8.5 illustrates, for the case where $k = 2$.

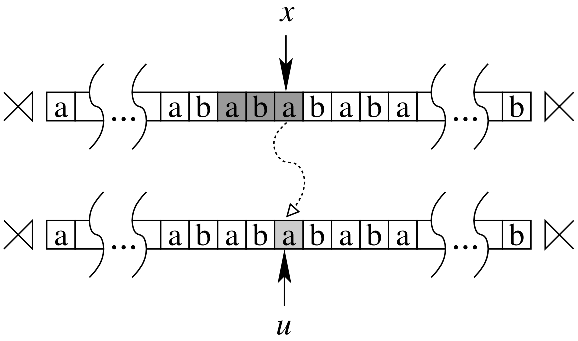


Figure 8.5: A schematic illustrating the Markovian nature of Input Strictly k-Local functions. For every Input Strictly 2-Local function, the output string u of each input element x depends only on x and the input element previous to x. In other words, the lightly shaded cell only depends on the darkly shaded cells.

[...]

(Chandlee, 2014) shows that ISL functions can model a range of local phonological processes, including substitution, insertion, deletion, and synchronic metathesis. More generally, she shows that given a mapping describable with a rule of the form A $\longrightarrow$ B / C __ D where the set of strings in CAD is finite and the rule applies simultaneously then it is ISL for some k.

This result may seem counter-intuitive given the current discussion. A reader may wonder whether, especially given the diagram in Figure 8.5, how ISL functions can model any transformation triggered by any right context at all. As mentioned, every element in the input string corresponds to a *string* of elements in the output. These output strings can be any length, including length zero (the so-called 'empty' string). The option to output the empty string allows the function to 'wait' until it has enough information to decide what to output. But importantly, the amount of input it needs to see to make this decision is *bounded*, by the specified value of k. For example consider regressive nasal place assimilation where underlying /inpa/↦[impa]. Each row in Table 8.1 shows how the output string is determined by each input element x and the input element preceding x. Since the output string at each point is determined by a window whose size is bounded by k, ISL maps are myopic in Wilson's (2003) sense.

element	input	output
preceding	element	string
x	x	u
⋊	i	i
i	n	λ
n	p	mp
p	a	a

Table 8.1: Illustrating why transformations with right contexts can still be ISL. The symbol λ represents the empty string (the string of length zero).

Chandlee also investigated the approximately 5500 phonological processes (from over 500 languages) reported in the P-Base database (v1.95 Mielke, 2008). It was determined that over 95% of these patterns are ISL. Chandlee acknowledges that P-Base ought not be taken as representative of the cross-linguistic distribution of processes that target contiguous versus non-contiguous segments. However, given that it is the most comprehensive collection of processes of which we are aware, she deemed it necessary to survey.

Furthermore, Chandlee (2014); Chandlee *et al.* (2014) also show how ISL functions can be efficiently learned from finitely many examples in the sense of Gold (1967) and de la Higuera (1997). This stands in stark contrast to the class of regular functions which cannot be so learned. Remarkably, Jardine *et al.* (2014) generalize this result to obtain an even more efficient learning algorithm for this class of functions.

Excerpt Ends.

8.3.3 Definiteness

Definition 41. *A deterministic finite-state machine has a k-definite structure whenever*

- $Q = \Sigma^{\leq k-1}$
- $q_0 = \lambda$
- $\delta(q, a) = r$ *iff* $r = \texttt{Suffix}^{k-1}(qa)$.

Theorem 28. *The language of DFA with a k-definite structure is a definite language.*

Definite languages are a subclass of the SL languages. They can be defined by identiying a finite set of permissible suffixes $\{w_1, \ldots w_n\}$. A word belongs to the language only if it ends with one of the w_i.

Theorem 29. *The language computed by a DFT with a k-definite structure with outputs from the Boolean monoid* $(\{\texttt{true}, \texttt{false}\}, \wedge, \texttt{true})$ *are k-SL languages.*

Theorem 30. *The function computed by a DFT with a k-definite structure with outputs from the* $(\Sigma^*, \circ, \lambda)$ *monoid are k-ISL languages.*

Dakotah Lambert was working on algebraic characterizations of various subregular classes when he realized that ISL functions have the definite structure. A reviewer pointed out that the such functions are p-local and pointed us to te Vaysse 1986 article. Vaysse defines p-local functions in terms of windows (like Figure 8.5; see Sakarovitch (2009, p. 61)) and provides an automata-theoretic characterization. Chandlee *et al.* (2014) define ISL functions abstractly using the concept of "tails" (also called "residuals"), and characterize these in terms of finite-state transducers that have the definite-structure.

What are tails and/or residuals? It is a way of dividing a formal language or function into equivalence classes. For a formal language L, the tails of a string w with respect to L is below.

$$T_L(w) = \{v : wv \in L\}$$

In other words it is the set of strings which continue w and obtain a string in L.

Consequently each language L naturally gives rise to an equivalence relation over Σ^*: strings w and w' are tail-equivalent with respect to L provided $T_L(w) = T_L(w')$. The equivalence relation partitions Σ^* into blocks of strings; all the strings in a block are tail-equivalent to each other. This relation is called the Nerode equivalence relation after Nerode who proved that a language is regular (acceptable by a NFA) if and only if the partition induced by L has finitely many blocks. His proof establishes a one-to-one correspondence between the blocks of this partition induced by L and the states of the minimal DFA recognizing L.

Incidentally, we can also define other equivalence relations over strings according to a language L. The contexts of a string w with respect to a language L is below.

$$C_L(w) = \{(u, v) : uwv \in L\}$$

In other words it is the set of pairs of strings (u, v) into which w can be "sandwiched" to obtain a string in L. This also induces an equivalence relation over Σ^*: strings w and w' are tail-equivalent with respect to L provided $C_L(w) = C_L(w')$. This equivalence relation is called the Myhill equivalence relation, after Myhill who proved that the partition induced by this relation has finitely many blocks if and only if L is regular. Algebraic characterizations of formal languages and trasnductions are based on the Myhill relation because the partition forms what is called a congruence, as opposed to the Nerode relation, which is only guaranteed to yield what is called a right congruence. Broadly speaking, the Myhill relation is more "symmetric" than the Nerode relation and therefore more natural from an algebraic perspective.

Both the Nerode and Myhill equivalence relations can be lifted to functions. An important concept in both cases is the *longest common prefix*. For a set of strings S the longest common prefix is the string u such that for all $w \in S$, there exists v such that $w = uv$ (so u

is a prefix of each $w \in S$) and for all other u' such that u' is a prefix of each $w \in S$, we have $|u'| < |u|$.

The Nerode equivalence relation extends naturally to functions by means of the tails of input strings. The set of tails of x in a function f, $T_f(x)$, is defined as follows:

$$T_f(x) = \{\langle y, v\rangle \colon f(xy) = \text{lcp}(f(x\Sigma^*))v\}.$$

Two strings are related iff their tails are equal. Choffrut proved that f induces a finite partition over Σ^* if and only if f is sequential, again by establishing a correspondence between the states of the minimal onward DFT for f and the blocks of this partition.

A two-sided extension generalizes Myhill equivalence. The contexts of x in f are as follows:

$$C_f(x) = \{\langle w, y, v\rangle \colon f(wxy) = \text{lcp}(f(wx\Sigma^*))v\}.$$

Chandlee's definition of k-ISL functions is that two strings w and w' are tail equivalent if and only if w and w' have the same $k-1$ suffix.

Lambert (2022) has more to say about algebraic characterizations of many classes, including definiteness.

8.3.4 Output Strictly Local Functions

For a k-ISL function, the states of the DFA depend on the last $k-1$ symbols read in the input string. In contrast, for a k-ISL function, the states of the DFA depend on the last $k-1$ symbols read in the output string. Why would be interested in this? The following excerpt from Heinz (2018, §6.2.2) explains.

Excerpt Begins.

A notable example of a map that Input Strictly Local functions are unable to model are ones like progressive harmony (PH). Recall that a mapping like $/+ - --/ \mapsto [+ + ++]$ belongs to this map, and more generally for all numbers k, $/+ -^k -/ \mapsto [+ +^k +]$ and $/- -^k -/ \mapsto [- -^k -]$. Such a map cannot be Input Strictly Local for any k. This is because whether the last input element surfaces as $[+]$ or $[-]$ depends on an *input* element which is more than k input elements away.

Chandlee (2014) defines Left and Right Output Strictly Local functions (LOSL and ROSL) to address such maps. These capitalize on the output-oriented nature of many phonological processes (Kisseberth, 1970; Prince and Smolensky, 1993, 2004). They are Markovian like ISL functions, but this time the context is found in the output string, not the input string. Specifically for Left (Right) OSL functions, for any input element x, its output string u will only depend on x and the previous (following) $k-1$ elements of the output string. The idea is that a function is Left or Right, depending on whether the left or right context in the output string matters. Figures 8.6 and 8.7 illustrate Left and Right OSL functions, respectively, for the case where $k = 2$.

Informally, Left and Right OSL functions can be thought of as characterizing the maps one can describe with rewrite rules that apply left-to-right or right-to-left (Howard, 1972)

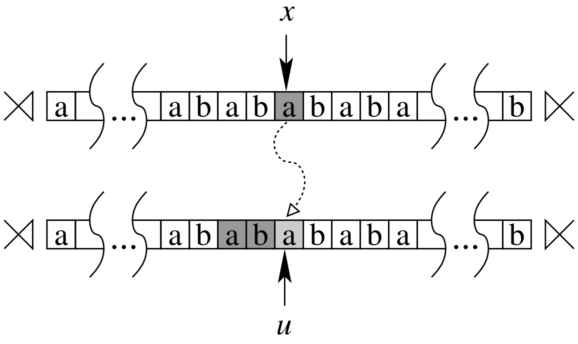


Figure 8.6: A schematic illustrating the Markovian nature of Left Output Strictly k-Local functions. For every Left Output Strictly 2-Local function, the output string u of each input element x depends only on x and the output element previous to u. As before, the lightly shaded cell only depends on the darkly shaded cells.

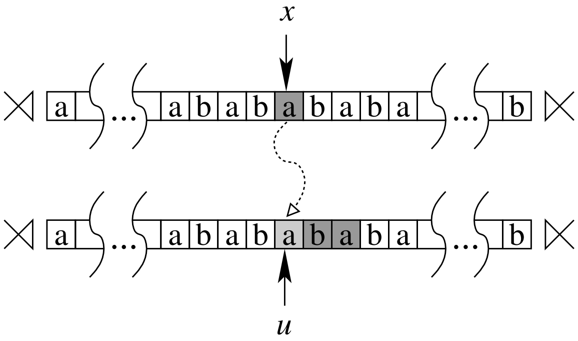


Figure 8.7: A schematic illustrating the Markovian nature of Right Output Strictly k-Local functions. For every Right Output Strictly 2-Local function, the output string u of each input element x depends only on x and the output element succeeding u. As before, the lightly shaded cell only depends on the darkly shaded cells.

(cf. the treatment of rule-application by Kaplan and Kay (1994)). This appears to be approximately correct, though certain details are still being worked out. However, we can say with certainty that the map PH is LOSL and the map RH is ROSL. More generally, such functions capture spreading processes such as progressive and regressive nasal spreading.

Left and Right OSL functions can both be computed by subsequential transducers. For Right OSL functions, the input string must be processed right-to-left by the transducer and the resulting output will then be reversed. See Heinz and Lai (2013) for details.

Excerpt Ends.

8.3.5 Input-Output Strictly Local Functions

Chandlee, Eyraud and Heinz have a never-ending “in progress” paper where we combine the two ideas via a product construction. Consquently, the states ofthe DFT only make distinctions based on both the last $k-1$ symbols on the input string *and* the last $k-1$ symbols written on the output.

8.4 Strictly Local Functions over Trees

Two different definitions of ISL tree transducers exist. One is given by Ji and Heinz (2020) using deterministic bottom-up tree transducers. Their goal was to follow the same line of thinking which informed Chandlee and colleagues. Another approach is given by Graf (2020) using a non-deterministic top-down approach.

Chapter 9

Conclusion

9.1 Conclusion

In this class, we learned about finite-state automata and how they can be used to express linguistic generalizations.

We studied two kinds of linguistic generalizations: representational and transformational. The former asks whether a particular representation is well-formed or not. The latter changes one representation into another. From a computational perspective, the first question corresponds to a *membership problem* and the second one to what we called a *transformation problem.*

Broadly speaking, automata are machines that solve particular classes of these problems. *Recognizers*, also called *acceptors*, solve membership problems. *Transducers* solve transformation problems.

When the automata are finite-state, it means that the memory they require is bounded by a constant, no matter the size of the input.

We studied four major classes of finite-state automata: recognizers and transducers for strings and trees. We implemented them in the programming language Haskell, a fully-typed, functional programming language with lazy evaluation. The programs were short and concise, and the code looked a lot like the math.

In the remainder of this conclusion, I want to emphasize aspects of automata that we were unable to get into in class.

9.1.1 Two-way deterministic automata

The automata we studied are called one-way automata. This means they processed the data structures in one direction. For instance, for strings the automata processed them left-to-right. For trees, it was either top-down or bottom-up. However, people have also studied two-way automata. These automata are allowed to re-read parts of the input string.

A useful analogy is to consider the automata as a scanning device with a “read head” that hovers over elements in the data structure. The diagram below shows such a device reading a string. Here the automata is like a scanner. It scans the string in one direction and

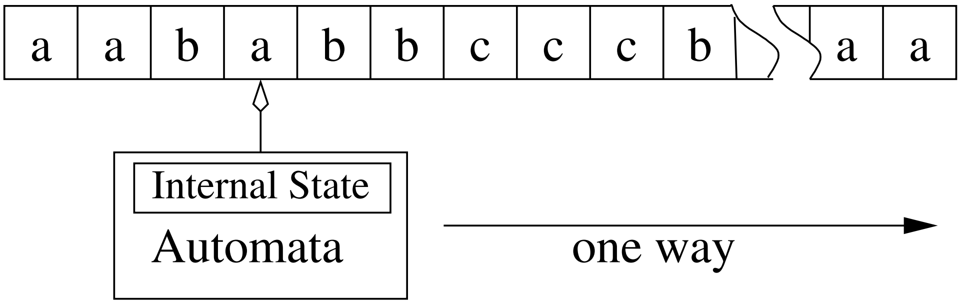


Figure 9.1: One-way automata

its internal state changes in response to new inputs. As Engelfriet and Hoogeboom (1999) explain, “Here the point of view has changed: the input is not fed into the automaton (like money into a coffee machine), but the automaton walks on the input string (like a mouse in a maze).”

The automata below is “Two-way.” This means it can move back and forth along the input tape.

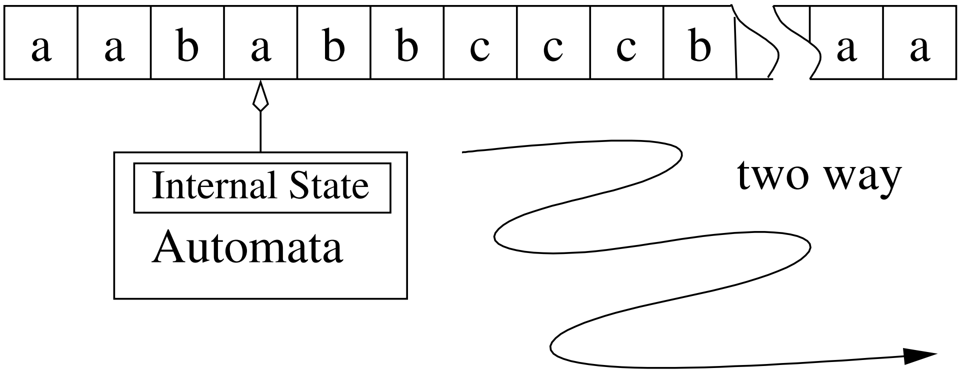


Figure 9.2: Two-way automata

Formally, for deterministic automata, the transition function δ has as its domain $Q \times \Sigma^*$ and as its co-domain $\{\texttt{Left}, \texttt{Stay}, \texttt{Right}\} \times Q$. The values `Left,Stay,Right` indicates whether the scanning device should move one position to the left or right or whether it should stay in its current position. If the scanner goes off the left edge, gets stuck in a loop, it rejects the string. It only accepts the string if it eventually moves off the right edge of the string

It is also possible to define non-deterministic two way automata.

In the case of string recognizers, all combinations are equally expressive.

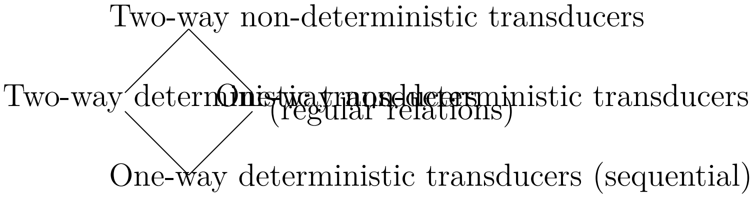
Theorem 31. *The class of stringsets recognized by*

1. *one-way deterministic acceptors,*

2. *two-way deterministic acceptors,*
3. *one-way non-deterministic acceptors, and*
4. *two-way non-deterministic acceptors*

are the same. They are the regular stringsets.

However, when we get to transducers, the picture changes. As mentioned, the deterministic/non-deterministic distinction matters for the transducer case. So does the one-way/two-way distinction. There are four distinct classes (Filiot and Reynier, 2016).


Two-way non-deterministic transducers
Two-way deterministic transducers
One-way non-deterministic transducers (regular relations)
One-way deterministic transducers (sequential)

One reason to be interested in two-way automata is that they provide a natural way to describe reduplication in natural languages! Total reduplication patterns *cannot* be described with one-way string transducers. Your classmate Hossep Dolatian has shown this is the case for over a hundred reduplication patterns in the world's languages and he will present this research at an upcoming linguistics conference at the University of Chicago.

For tree automata, there is an analog of "Two-way." It is called "Walking" and these automata can move from one node to another (up to a parent or down to one of the children). They have been difficult to analyze; see Engelfriet and Hoogeboom (1999).

9.1.2 Natural Language Processing

Finite-state automata are widely used in natural language processing tasks (Roche and Schabes, 1997; Mohri, 1997, 2005; Jurafsky and Martin, 2008b). Many of these models rely on probabilities. Essentially, the transitions of the automata are weighted in the way we considered when the outputs of the sequential transducers were elements of a monoid.

In the domain of syntax, for many years people used context-free grammars or context-sensitive grammars, but a recently some replaced those with weighted tree automata (Knight and May, 2009; Maletti, 2009).

There are many aspects of NLP that this class did not cover since it focused on theoretical and foundational aspects. However, I hope that if you choose to further study NLP, the material you would have learned in this class would help.

9.1.3 Combining Automata and complex generalizations

Why do we care about the closure properties of automata? We care because they tell us that we can build new automata from old ones and that the new ones we build will behave the way we intend. This is especially useful if we have complex linguistic generalizations.

For example, consider a language which places stress on the first heavy syllable in a word if there is a heavy syllable. On words with no heavy syllables, stress goes on the last syllable. Simplifying we can assume that the letters are L and H for “Light” and “Heavy” respectively. Thus we have:

HLL	↦	H́LL
HHH	↦	H́HH
LLH	↦	LLH́
LLL	↦	LLĹ
LLLL	↦	LLLĹ

It may be difficult to write a single automaton for this. However, if we are writing non-deterministic transducer, we can write one transducer for all the words with no heavy syllables (below left) and one transducer for the words with at least one heavy syllable (below right). Each of these is simple to write.

Next, since regular relations are closed under union we can simply take the union of these two machines. That’s it! Furthermore, if there is a deterministic transducer able to describe this pattern, there are algorithms that can determinize it (Mohri, 1997).

More generally, operations like intersection, composition, and concatenation let us describe complex generalizations in terms of simpler parts.

9.1.4 Subregular Classes of Automata

There are many types of regular stringsets, which can be further classified. The figure below shows one classification based on logic. Each of these classes have multiple characterizations in terms of logic, automata, and algebraic properties (McNaughton and Papert, 1971; Thomas, 1982; Rogers and Pullum, 2011; Rogers *et al.*, 2010). Rogers *et al.* (2013) argue

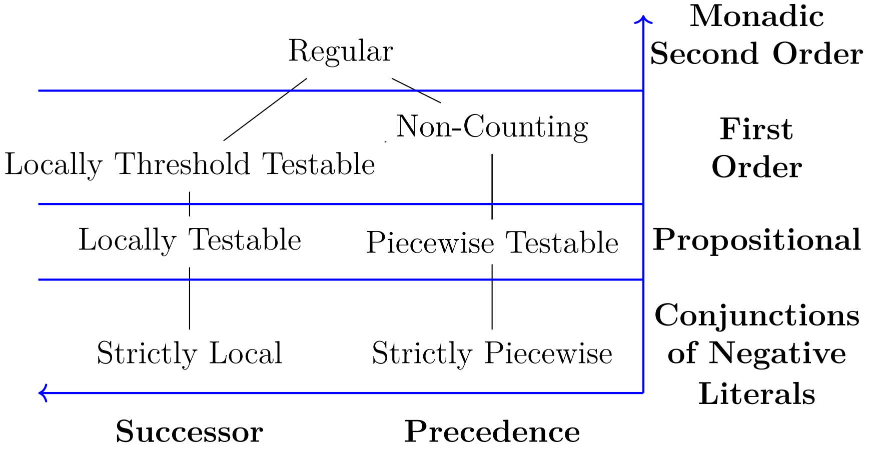


Figure 9.3: Subregular hierarchies of stringsets

that each of these classes correspond to a kind of model of memory with implications for cognition.

Heinz (2010) argues that all human phonotactic generalizations belong to the smallest classes shown here: Strictly Local and Strictly Piecewise generalizations. Heinz and Idsardi (2011, 2013) develop this view in the broader context of language and cognition.

What about subregular string transformations? This is much less well-studied and the University of Delaware has been at the forefront of this with Jane Chandlee's PhD work (Chandlee, 2014; Chandlee *et al.*, 2014, 2015).

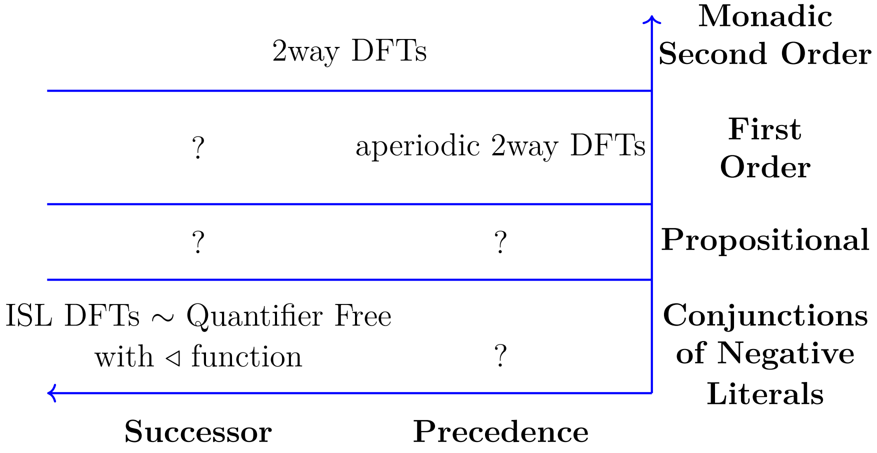


Figure 9.4: Subregular hierarchies of string-to-string functions

Heinz (2018) overviews all of the work up to that point. Lambert (2022) algebraic characterizations yield methods for deciding whether a given language or function (given as a deterministic finite-state machine) belongs to some class. With respect to functions, there is a lot here to explore.

9.1.5 Learning Automata

Finally, there has been a lot of work on how automata can be learned. Here are some of the main results. Readers are referred to the following texts for more information Heinz *et al.* (2015); de la Higuera (2010); Heinz and Sempere (2016).

(Note that "identification in the limit" is a well-known definition for learning dating to Gold's seminal work Gold (1967) even if it is a little controversial (Clark and Lappin, 2011). See also Heinz (2016).)

Theorem 32. *The regular stringsets cannot be identified in the limit from positive data.*

Theorem 33. *The regular stringsets and recognizable treesets can be efficiently identified in the limit from positive and negative examples.*

Theorem 34. *The Strictly k-Local, Strictly k-Piecewise, Locally k-Testable, Piecewise k-Testable, and Locally t-Threshold k-Testable classes (stringsets) each be efficiently identified in the limit from positive and negative data.*

Furthermore, there are concrete senses in which the lower classes are more efficiently learnable than the higher ones. I don't know of work on subregular treesets, but these results should carry through.

Theorem 35. *The class of sequential functions is efficiently identifiable in the limit from positive data.*

Theorem 36. *The class of probability distributions over strings and over trees describable with weighted deterministic recognizers are efficiently approximable from positive data.*

Bibliography

Applegate, R.B. 1972. Ineseño Chumash grammar. Doctoral dissertation, University of California, Berkeley.

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.

Beesley, Kenneth, and Lauri Kartunnen. 2003. *Finite State Morphology.* CSLI Publications.

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

Chandlee, Jane. 2014. Strictly local phonological processes. Doctoral dissertation, The University of Delaware.

Chandlee, Jane, Rémi Eyraud, and Jeffrey Heinz. 2014. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics* 2:491–503.

Chandlee, Jane, Rémi Eyraud, and Jeffrey Heinz. 2015. Output strictly local functions. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, edited by Marco Kuhlmann, Makoto Kanazawa, and Gregory M. Kobele, 112–125. Chicago, USA.

Choffrut, Christian. 2003. Minimizing subsequential transducers: a survey. *Theoretical Computer Science* 292:131 – 143.

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

Clark, Alexander, and Shalom Lappin. 2011. *Linguistic Nativism and the Poverty of the Stimulus.* Wiley-Blackwell.

Comon, H., M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree automata techniques and applications. Available on: `http://tata.gforge.inria.fr/`. Release October, 12th 2007.

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.

Engelfriet, Joost, and Hendrik Jan Hoogeboom. 1999. Tree-walking pebble automata. In *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, edited by Juhani Karhumäki, Hermann Maurer, Gheorghe Păun, and Grzegorz Rozenberg, 72–83. Berlin, Heidelberg: Springer Berlin Heidelberg.

Filiot, Emmanuel, and Pierre-Alain Reynier. 2016. Transducers, logic and algebra for functions of finite words. *ACM SIGLOG News* 3:4–19.

Gold, E.M. 1967. Language identification in the limit. *Information and Control* 10:447–474.

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. 2020. Curbing feature coding: Strictly local feature assignment. In *Proceedings of the Society for Computation in Linguistics 2020*, 224–233. New York, New York: Association for Computational Linguistics.
URL `https://aclanthology.org/2020.scil-1.27`

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. 2010. Learning long-distance phonotactics. *Linguistic Inquiry* 41:623–661.

Heinz, Jeffrey. 2016. Computational theories of learning and developmental psycholinguistics. In *The Oxford Handbook of Developmental Linguistics*, edited by Jeffrey Lidz, William Synder, and Joe Pater, chap. 27, 633–663. Oxford, UK: 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.

Heinz, Jeffrey, Colin de la Higuera, and Menno van Zaanen. 2015. *Grammatical Inference for Computational Linguistics*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool.

Heinz, Jeffrey, and William Idsardi. 2011. Sentence and word complexity. *Science* 333:295–297.

Heinz, Jeffrey, and William Idsardi. 2013. What complexity differences reveal about domains in language. *Topics in Cognitive Science* 5:111–131.

Heinz, Jeffrey, and Regine Lai. 2013. Vowel harmony and subsequentiality. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, edited by Andras Kornai and Marco Kuhlmann, 52–63. Sofia, Bulgaria.

Heinz, Jeffrey, and José Sempere, eds. 2016. *Topics in Grammatical Inference*. Berlin Heidelberg: Springer-Verlag.

de la Higuera, Colin. 1997. Characteristic sets for polynomial grammatical inference. *Machine Learning* 27:125–138.

de la Higuera, Colin. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.

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.

Howard, Irwin. 1972. A directional theory of rule application in phonology. Doctoral dissertation, Massachusetts Institute of Technology.

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.

Jardine, Adam. 2016. Computationally, tone is different. *Phonology* 32:247–283.

Jardine, Adam, Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. 2014. Very efficient learning of structured classes of subsequential functions from positive data. In *Proceedings of the Twelfth International Conference on Grammatical Inference (ICGI 2014)*, edited by Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka, vol. 34, 94–108. JMLR: Workshop and Conference Proceedings.

Ji, Jing, and Jeffrey Heinz. 2020. Input strictly local tree transducers. In *Proceedings of the 14th International Conference on Language and Automata Theory and Applications (LATA 2020)*, edited by A. Leporati, C. Martín-Vide, D. Shapira, and C. Zandron, Lecture Notes in Computer Science, 369–381. Springer.

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

Jurafsky, Daniel, and James Martin. 2008a. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics.* 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Jurafsky, Daniel, and James Martin. 2008b. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics.* 2nd ed. Prentice-Hall.

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

Kisseberth, Charles. 1970. On the functional unity of phonological rules. *Linguistic Inquiry* 1:291–306.

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.

Knight, Kevin, and Jonathan May. 2009. Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*, edited by Manfred Droste, Werner Kuich, and Heiko Vogler, Monographs in Theoretical Computer Science, chap. 14. Springer.

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/`

Lambert, Dakotah. 2024. System description: A theorem-prover for subregular systems: The language toolkit and its interpreter, plebby. In *Functional and Logic Programming*, edited by Jeremy Gibbons and Dale Miller, 311–328. Singapore: Springer Nature Singapore.

Lothaire, M., ed. 2005. *Applied Combinatorics on Words.* 2nd ed. Cmbridge University Press.

Maletti, Andreas. 2009. Minimizing deterministic weighted tree automata. *Information and Computation* 207:1284 – 1299.

McCollum, Adam G., Eric Baković, Anna Mai, and Eric Meinhardt. 2020. Unbounded circumambient patterns in segmental phonology. *Phonology* 37:215–255.

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

Mielke, Jeff. 2008. *The Emergence of Distinctive Features*. Oxford: Oxford University Press.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics* 23:269–311.

Mohri, Mehryar. 2005. Statistical natural language processing. In *Applied Combinatorics on Words*, edited by M. Lothaire. Cambridge University Press.

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

Prince, Alan, and Paul Smolensky. 1993. Optimality Theory: Constraint interaction in generative grammar. Tech. Rep. 2, Rutgers University Center for Cognitive Science.

Prince, Alan, and Paul Smolensky. 2004. *Optimality Theory: Constraint Interaction in Generative Grammar*. Blackwell Publishing.

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

Roche, Emmanuel, and Yves Schabes. 1997. *Finite-State Language Processing*. MIT 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, and Jeffrey Heinz. 2014. *Model-Theoretic Phonology*. Tübingen, Germany. Course taught at the 2014 European Summer School for Logic, Language, and Information (ESSLI).

Rogers, James, Jeffrey Heinz, Gil Bailey, Matt Edlefsen, Molly Visscher, David Wellcome, and Sean Wibel. 2010. On languages piecewise testable in the strict sense. In *The Mathematics of Language*, edited by Christian Ebert, Gerhard Jäger, and Jens Michaelis, vol. 6149 of *Lecture Notes in Artifical Intelligence*, 255–265. Springer.

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.

Rogers, James, and Dakotah Lambert. 2019. Some classes of sets of structures definable without quantifiers. In *Proceedings of the 16th Meeting on the Mathematics of Language*, 63–77. Toronto, Canada: Association for Computational Linguistics.
URL `https://www.aclweb.org/anthology/W19-5706`

Rogers, James, and Geoffrey Pullum. 2011. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information* 20:329–342.

Sakarovitch, Jaques. 2009. *Elements of Automata Theory*. Cambridge University Press. Translated by Reuben Thomas from the 2003 edition published by Vuibert, Paris.

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.