Unifying Classification Schemes for Languages and Processes

With Attention to Locality and Relativizations Thereof

A Dissertation Presented

by

Dakotah Jay Lambert

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Linguistics

Stony Brook University

May 2022

Stony Brook University

The Graduate School

Dakotah Jay Lambert

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Jeffrey Heinz — Dissertation Advisor
Professor, Department of Linguistics and Institute for Advanced Computational Science

Thomas Graf — Chairperson of Defense
Associate Professor, Department of Linguistics

Jordan Kodner
Assistant Professor, Department of Linguistics

James Rogers
Professor Emeritus, Department of Computer Science, Earlham College

This dissertation is accepted by the Graduate School

Eric Wetheimer
Dean of the Graduate School

Abstract of the Dissertation

Unifying Classification Schemes for Languages and Processes

With Attention to Locality and Relativizations Thereof

by

Dakotah Jay Lambert

Doctor of Philosophy

in

Linguistics

Stony Brook University

2022

This dissertation synthesizes research in abstract algebra, computer science, and linguistics to better characterize properties of natural languages. The first contribution is a thorough explanation of how a traditional linguistic treatment of nonlocality (relativized adjacency) ftis into existing computational treatments of sequential patterns. The second is a classification scheme unifying multiple complexity hierarchies that have been proposed in prior literature. The third is an extension of this unified classification scheme to functions and other transformations. The importance of these results lies in their explanatory power. They provide a lower upper bound on the complexity of linguistic generalizations than has been proposed before, and they help to explain how any mechanism, human or machine, can learn these patterns from small amounts of data. Moreover they provide a basis for developing new algorithms for processing language, simplifying the task by taking advantage of their fundamental properties. Because these characterizations are grounded in alegbraic theory, they draw tighter and clearer connections between the formal descriptions of

language used by computer scientists and linguists.

The complexity of a language determines what kinds of mechanisms, cognitive or computational, are required in order to learn it, or in order to verify whether it contains a given form. The classical Chomsky hierarchy partitions languages into four groups by what kinds of computation define them: unrestricted computation (type 0), context-sensitive rewriting (type 1), context-free rewriting (type 2), and regular expressions (type 3). Each is contained in the previous; all type 3 languages are also type 2 languages. There was no type 4, but even still we encounter plenty of patterns that do not need the full power of regular expressions. As a result, many subregular hierarchies have been explored, some defined by restricted applications of formal logic, others defined by the containment of subparts. This dissertation uses algebraic methods to unify some of these subregular hierarchies, as well as to extend their application from languages to functions and relations. With this unified classification scheme, I show that the upper bound on the complexity of linguistic generalizations which govern systematic variation in the pronunciation of related words is lower than previously thought.

Natural language processing tasks can often take advantage of these less-complex classes: when an n-gram model is applicable, it offers a standard and extremely simple mechanism for learning. Under such a model, a word is valid if (and only if) all of its n-long substrings are attested. This is perfectly suitable for capturing some types of linguistic pattern, such as the strict consonant-vowel alternation embodied in some languages. An n-gram model is heavily restricted in that it can only account for local dependencies. Long-distance constraints cannot be adequately described by such a system. One way to extend the system to account for long-distance constraints is to consider what

would be adjacent if only we could ignore irrelevant symbols. In other words, we restrict to some tier of salient symbols. I provide several equivalent characterizations for tier-based extensions of various subregular classes, and discuss how standard n-gram learners can be extended to account for these more powerful classes.

One type of characterization is based on algebraic methods. Strings can be partitioned into equivalence classes based on how the pattern treats them, and the structure of these classes corresponds to the complexity of the pattern. Algebraic properties prove exceptionally useful in combining hierarchies: if one can show that all of the properties required by one class are necessarily satisfied by another, then the second is a subclass of the first. This provides a mechanism to unify the subregular hierarchies that have been proposed over the years, and to prove whether multiple classes are equivalent to one another. Indeed, by applying these methods to functions one can show that the class of total input strictly local functions, well-known in computational phonology, corresponds precisely to the class of order-preserving definite functions, known in theoretical computer science. The two fields have independently found value in the same type of simple function, and the equivalence opens the door for each to learn from what the other has discovered.

The algebraic structure of a regular language or function can be derived from the finite-state machines that represent them. For languages, and for certain types of functions, there is a canonical form from which we can be sure we have found the simplest structure. For other types of functions, no such canonical form is known, but an upper bound on complexity can still be found by examining one or more given implementations. Unfortunately while these classification schemes apply to string languages and string-to-string transformations, tree languages have not received such deep

characterizations. However, the decision procedures for the classes on strings are based, generally, on graph-algorithms. Tree acceptors and transformations more naturally take on hypergraph representations. I propose such a representation here, offering one step toward a more general classification scheme that handles trees as easily as strings.

In summary, this dissertation unifies subregular hierarchies explored by linguists and by theoretical computer scientists using algebraic methods. The unified classification scheme is applied to various patterns in natural language in order to propose a lower upper bound on complexity than ever before. Finally directions for future research, especially regarding extension of these methods to patterns over trees, are discussed. The algorithms described are not merely theoretical. All algorithms discussed in this dissertation are implemented in software.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Numerous types of constraints are used in description of linguistic patterns. I explore the types of patterns definable with a constant memory bound over both strings and trees. With a primary focus on phonology and strings, I discuss how learnability considerations may provide reason to prefer some constraints over others. Methods for extracting constraints are provided, both from data (learning) and from automata (factoring). These may help in finding descriptions extensionally equivalent to a given pattern. Also, if a typologically predictive theory is desired then care must be taken to avoid formalisms that can define all computable functions.

Finite-state machines formalize the notion of a constant memory bound and have been widely used for decades in both computer science (Thompson, 1968) and linguistics. Using different properties of the machines or of the formal languages they represent, we can discover a great deal about languages, including their complexity (Rogers et al., 2012) or how they relate to one another (Clark and Roberts, 1993). Constraint-based analysis have proven particularly useful (Lambert and Rogers, 2019), so I aim to describe both factorization methods, for cases in which one is presented with a structure, and learning methods, for cases in which one is presented with data.

In classification, algebraic characterizations are often used because they can just as easily affirm or deny the inclusion of a language in a class. But also language-theoretic characterizations are useful, because they are machine agnostic. I aim to provide both types of characterizations for some classes which previously have received no such treatment, specifically the tier-based classes and the

(strongly) costrict classes. Further, tree languages and finite-state methods thereon have traditionally been treated as entirely distinct from tree languages, so I aim to unify the analyses.

Acceptors are not everything, however. Often in linguistics we wish to discuss transformations from one structure to another, or more generally to explain how related structures relate. There are several approaches in wide use, including logical transductions (Courcelle, 1994), semiring-based analyses (Lothaire, 2005), and finite-state systems with outputs on the edges (Mohri, 1997). The last of these is really a special case of the second. I aim to show that the semiring-based analysis is too powerful. Instead, I explore a generalization to transducers of the algebraic classification scheme used for acceptors. This generalization has been explored to some extent by Carton and Dartois (2015).

In general, I aim to unify finite-state methods for languages and relations over strings and trees, all while avoiding the issue of unbounded computational power. How do we determine which constraints they satisfy? In this work, I begin this unification by discussing the methods used on string languages and by proposing a unifying graphlike structure for string and tree languages. Rather than discuss the structure of string languages or of tree languages or of relations, I would like to discuss simply the structure of structure.

In short, I am tackling a lack of unity and completeness in the formal description of linguistic patterns. The result of this work will provide a firm mathematical and computational basis for linguistic description and for models of language learning, in addition to improving pedagogy and tools relating to constraint-based descriptions of languages.

1.1 Review of the Literature

Kleene (1956) introduces the regular languages and associates them with finite-state automata. McNaughton and Papert (1971) discuss formal languages built around local dependencies: the strictly local and locally testable classes. These languages have some nice properties: they are efficiently learnable, efficiently testable, and closed under intersection (and thus usable as constraints) (Heinz, 2010b; Rogers et al., 2012). Indeed, these classes were studied for their relative simplicity compared to the regular languages, requiring no modulo-counting mechanism. The locally threshold testable class of Beauquier and Pin (1989) lies between regular and locally testable allowing for counting of local structures. Local constraints can't handle everything, however, and natural language is full of patterns outside of these classes, such as the stress pattern of Yidin (Goedemans et al., 2015; Lambert and Rogers, 2019), the asymmetric sibilant harmony of Sarcee (Heinz, 2010a), or the tone plateauing of Luganda (Hyman and Katamba, 1993). None of these are representable even by locally threshold testable descriptions. They can be analyzed by star-free languages (also known as group-free, locally testable with order, or noncounting), but often a simpler analysis is possible.

All of these difficult patterns invoke long-distance dependencies. The piecewise testable languages of Simon (1975) directly encode this type of dependency. Mirroring the strictly local restriction of the locally testable languages, the piecewise testable class can be restricted to a strictly piecewise class (Rogers et al., 2010, see also Haines, 1969). There is no distinct piecewise threshold testable class; it is equivalent to piecewise testable (Lambert et al., 2021a).

Another way of representing some types of long-distance dependencies, based on popular feature geometry models such as those discussed by McCarthy (1988), is the class of tier-based strictly

local languages described by Heinz et al. (2011). There is no corresponding tier-based strictly piecewise class, because the general precedence relation is not strengthened or weakened by the ability to ignore specific symbols. The naturalness and power of tier-based descriptions has caused them to be widely used and extended (McMullin, 2016; Jardine and McMullin, 2017; Aksënova and Deshmukh, 2018; De Santo and Graf, 2019; Lambert, 2021a). Mayer and Major (2018) even discuss a hypothesis that all phonological patterns are tier-based strictly local, then go on to provide a counterexample. This should come as no surprise; the tone plateauing of Luganda is also outside of this class, unless we assume a representation that removes the unrepresentable long-distance dependencies.

My opinion is that it is a mistake to believe that any given subregular class could act as a universal descriptor. Instead, a focus on classification is important, as each class has a set of associated properties, and that can tell us a lot about the patterns we see. Patterns may be described in several equivalent ways, perhaps belonging to several classes and sharing all of their properties. If given data, or a prosodic description of a pattern, it may be easy to discover counterexamples to some class-specific language-based characterization. For instance, the strictly piecewise languages are closed under deletion (Rogers et al., 2010), so if a language in this class contained a word CVCV, it would necessarily also contain a word CVC. These language-theoretic characterizations provide an easy way to say that a language cannot be in the class, but a guarantee of inclusion may be more difficult to provide. A grammatical-formalism based analysis has the opposite problem: providing a grammar is an easy way to guarantee inclusion, but it may be difficult to prove that no such grammar is possible.

McNaughton and Papert (1971) discuss the use of the syntactic monoid of a language to determine if that language is star-free. This algebraic structure is derived from the canonical acceptor for the language in question. Algebraic procedures for deciding membership in the locally testable class (Brzozowski and Simon, 1973; McNaughton, 1974), the locally threshold testable class (Beauquier and Pin, 1989), and the piecewise testable class (Simon, 1975) also exist. For those classes testable in the strict sense, an algebraic approach finds an issue: an automaton and its complement share a syntactic monoid, but these classes are not closed under complementation. More information is necessary beyond the structure itself, as shown by De Luca and Restivo (1980) for the strictly local class, or by Fu et al. (2011) for the strictly piecewise class. Lambert (2021b) describes a general algebraic characterization for tier-based languages. Indeed, several subregular classes have algebraic decision procedures, including the class defined by the fragment of first-order logic restricted to two variables and general precedence (Thérien and Wilke, 1998) and possibly betweenness (Krebs et al., 2020).

A pattern may be defined by zero or more constraints over the set of all possible words. Rather than classifying a pattern in its entirety, it may be useful to factor it into simple constraints and classify those (Lambert and Rogers, 2019). Leaving the pattern in its factored state can result in a smaller state space and easier processing (Heinz and Rogers, 2013). Rogers and Lambert (2019b) describe a mechanism for extracting some types of subregular constraints from finite-state automata. Lambert (2021b) shows how factoring can be extended to tier-based classes, but a persistent problem plagues tier-based analyses: other constraints easily interfere with the tier selection process (Lambert, 2021a).

As with classification, it may be desired to determine which constraints are satisfied by some data without a neat, complete description of the pattern. This is the learning problem. Gold (1967) proposed several learning frameworks, and one in particular has caught on: learnability in the limit from positive data. The restriction to only positive data both provides a more tightly restricted learning paradigm and, as Yang (2015) discusses, may more accurately reflect the acquisition process of natural language. Most if not all of the subregular classes are learnable in the limit from positive data using some variant of Heinz's (2010b) string extension learning (Lambert et al., 2021a; Lambert, 2021a). One may go so far as to suggest that without an effective learning procedure, a class is useless in description of natural language phenomena. The robustness of the learning in the presence of sparse data may also be of concern.

The discussion so far has dealt only with string languages definable by a state machine of finitely many states. Syntax cannot be described this way. Syntax, like many other hierarchical systems, deals in trees. Rogers (1997) showed that context free grammars, while producing superregular string yields, are only strictly 2-local over trees. Graf (2018) shows that, in at least some sense, the operations fundamental to Minimalist syntax are tier-based strictly local on trees. The question becomes "how do we classify tree languages?" Klein and Manning (2004) and Huang and Chiang (2005) consider state machines based on a chart parse such as a CKY table, where the number of states grows with the size of the input. True finite-state machines also appear; Gécseg and Steinby (1984) and Comon et al. (2007) both detail finite-state analyses wherein a recursive procedure assigns a state to each subtree before deciding the state for the tree as a whole. It seems that none have constructed graph-based analyses of problems with respect to these tree automata, so this problem is explored here. On the topic of trees, algebraic characterizations are indispensable in

classifying string languages. But the syntactic monoids of context free languages are infinite, and may not have such nice properties. Clark (2015) discusses a different algebraic system over strings which classifies the context-free languages, and a few algebraic systems have been proposed for dealing with tree languages (Germain and Pallo, 2000; Steinby, 2015), but it remains unclear how these connect to a subregular hierarchy for trees.

Finite state acceptors can also be extended by associating outputs with their edges in order to describe transformations, sometimes referred to as transductions. If each input symbol is required to be associated with some sequence of output symbols, which are concatenated as computation progresses, then the rational relations are described (Frougny and Sakarovitch, 1993, see also Rabin and Scott, 1959). Deterministic relations (functions) in this class are called "subsequential", and are frequently used in phonology (Chandlee, 2014). Indeed (Chandlee, 2014) describes subclasses of the subsequential functions, input strictly local and output strictly local, to account for certain phonological properties. Tonal phonology and templatic morphology are described by input strictly local functions over multiple tapes (Dolatian and Rawski, 2020; Rawski and Dolatian, 2020), and tier-based functions have been proposed (Burness and McMullin, 2019). Another common approach, which uses semirings to unify acceptors, transducers, weighted automata, and more, is described by Lothaire (2005). This semiring approach essentially augments the state machine with an accumulator in some monoid. I show that the power of such an accumulator is unrestricted. The approach with less freedom in structure yields a more restrictive set of relations. Courcelle (1994) describes logical transductions, which abstract away from the machine. Classes of string languages may be related both to a particular kind of algebraic property and to some particular kind of logical formalism (Rogers and Lambert, 2019a), so the same ought to hold for classes of functions. Unbounded

deletion prevents input strictly local functions from aligning with a kind of quantifier-free logic, but restrictions on functions based on origin information (Bojańczyk, 2014; Bojańczyk et al., 2017) may be useful in closing the gaps.

1.2 Outline of the Dissertation

Chapter 2 introduces the core concepts that will be used throughout this work, including (sub)regular languages, finite-state automata, model theory, and factors. Beyond that, this dissertation is structured such that the remaining chapters can be read as independent works in their own right. Yet it is also designed to promote a unifying perspective. The first part, from here through chapter 5, focuses on thorough exploration of the piecewise-local subregular hierarchy. Classes based on phonological tiers are characterized in a multitude of ways, and the algebraic approach in particular is extended to investigate linguistically relevant functions. The second part, spanning the remaining chapters until the conclusion, ties up loose ends. This includes learning algorithms for tier-based classes, a graphlike structure for tree-automata, as well as finally an argument against one sort of transducer-acceptor unification.

The piecewise-local subregular hierarchy is a partial order of language classes which can be described via the successor or general precedence relations using at most monadic second-order logic. Tier-based languages are defined by the transitive reduct of a particular restriction of the general precedence relation. A subset of the piecewise-local classes form a grid, where one axis describes the types of factors in use and the other describes the kinds of distinctions made. Other classes fit less neatly into such a paradigm. Chapter 3 discusses classification of regular languages into the classes of the piecewise-local subregular hierarchy, including algebraic, automata-theoretic,

language-theoretic, and model-theoretic approaches to the problem. In particular, the complements of strictly languages are newly characterized, and tier-based extensions are provided for all classes of the hierarchy, extending the tier-based strictly local class. Chapter 4 incorporates classes from the computer science literature into the hierarchy, namely those classes based on definability with first-order logic restricted to two variables and several other classes characterized algebraically. Concluding the first part, chapter 5 extends these algebraic notions to functions and begins a process of categorizing the processes that occur in natural language.

While chapter 3 includes brief discussion on extracting constraints from automata, Chapter 6 incorporates discussion on constraint extraction from data (learning). This problem is explored through a generalization of string extension learners, extending the original definition to account for different possible interpretations of grammars, and a learning algorithm for tier-based languages is defined.

There are many ways to represent string acceptors. One might use the potentially infinite set describing the target language, some kind of grammar, or a state machine. These state machines are represented by graphs, but properly superregular languages require infinitely many states. Context free languages can, however, be recognized with finitely many states using a tree acceptor rather than a string acceptor. Chapter 7 introduces strongly directed hypergraphs in order to represent tree acceptors as direct generalizations of string acceptors. The Boolean operations are described in much the same way as the graph-based algorithms for strings. Some decision problems require less structure, and these reductions are described.

Throughout these chapters it is discussed that algebraic structure is immensely useful in complexity classification. Some have expressed interest in describing certain phenomena using semiring-based transducers, which essentially are finite-state acceptors augmented with a monoidal accumulator. Chapter 8 discusses the expressive power of such a structure. First I show that a CKY-style chart parse can be expressed with a monoid. Then I generalize, demonstrating that a run of a Turing machine can also be expressed as a monoid. In order to maintain a truly finite amount of state, one may demand the use of a finite monoid, but this prohibits representing even the identity function. A fundamental question remains: what kinds of restrictions should a monoidal accumulator satisfy? My answer is that the monoidal accumulator is the wrong approach in general, unless care is taken to ensure that the accumulator does not hide too much power.

The final chapter concludes the work by reiterating the goals of a structure-based approach to linguistic analysis and stating directions for future work.

Part I

Chapter 2: Formal Languages and String Acceptors

This chapter introduces formal languages and other concepts that will be used throughout the text, including finite-state automata, finite model theory, and basic abstract algebra. If a language can be associated with some fixed memory bound under which any possible word can be tested for membership, then that language is regular. The piecewise-local subregular hierarchy, which is shown in Figure 2.3 and which will be explored further in Chapter 3, is a collection of classes of regular languages, where each class imposes its own additional constraints.

2.1 Notation

A function from a domain $\mathcal{D}$ to a codomain $\mathcal{D}'$ is written $f : \mathcal{D} \to \mathcal{D}'$. The set of natural numbers is written $\mathbb{N}$. As a special case reflecting traditional notation for sequences, the value of a function $f : \mathbb{N} \to \mathcal{D}'$ at n is written f_n.

2.2 Formal Language Theory

A word is a sequence of symbols drawn from some alphabet, typically written Σ. The set of all possible words is Σ^*, and the set of nonempty words is Σ^+. For reasons that will be discussed in section 2.5, these are sometimes referred to as the **free semigroup** and **free monoid**, respectively, over Σ. A formal language is nothing more than a possibly infinite set of words. Such a set may also be referred to as a pattern or, when considering intersections of multiple patterns, a constraint. Different classes of formal languages are defined based on what mechanisms are needed in order to decide whether a word belongs to the language. A class is a collection of languages, and a

characterization of a class is a property such that all and only those languages that have this property are in the class.

The locally testable (LT) and strictly local (SL) languages discussed by McNaughton and Papert (1971) can be recognized by a fixed-width left-to-right scanner (see also Beauquier and Pin, 1991). Several language classes admit more than one characterization. For example, the locally threshold testable (LTT) class of Beauquier and Pin (1989) can be described as all and only those language recognizable by such a scanner where each possible factor has its attestation count incremented (up to some threshold) each time it is encountered and the acceptability of the word is determined by the collection of counts. Or it could be described as all and only those languages first-order definable with successor Thomas (1982). Each of these characterizations tells us something about the languages in the class.

2.3 Finite Model Theory

Concepts from finite model theory provide a uniform way to describe relational structures and their parts in logical terms (see Libkin, 2004 for a thorough introduction). Applying these concepts to linguistic structure is not a new idea, with applications to syntax by Rogers (1996, 1998) beginning to popularize the approach.

A relational word model consists of a **domain**, $\mathcal{D}$, which is isomorphic to an initial segment $\{1, \ldots, n\}$ of the nonzero natural numbers and represents positions in the word, as well as a collection of relations, $R_i \subseteq \mathcal{D}^{a_i}$, each of which has its own arity a_i.

$$\mathscr{M}(w) = \langle \mathcal{D}; R_i \rangle.$$

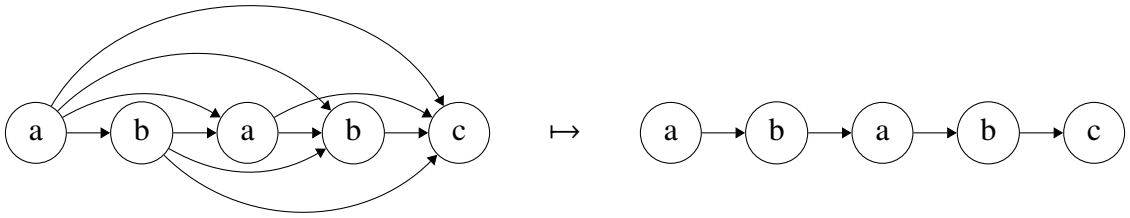


Figure 2.1: Precedence and successor models of "ababc".

Generally we assume that a model consists of at least one **ordering** relation, as well as one or more unary **labeling** relations that partition the domain. Additional relations of any arity are of course permitted. The assumption of a partition is nonrestrictive; one can convert a model whose labeling relations do not form a partition of the domain into a partitioned normal form by using the powerset of these relations instead. One simple example of an ordering relation is that of general precedence ($<$), where $a < b$ if and only if (iff) the domain element a occurs anywhere before b.

The immediate successor relation that defines the local branch of the subregular hierarchy can be derived in first-order logic from general precedence.

$$x \lhd y \triangleq (x < y) \wedge \neg(\exists z)[x < z < y].$$

This is simply the transitive reduction of this general precedence relation. An example word model is shown in Figure 2.1.

Now that we have a notion of a structure, we turn to discussion of contained structures.[1] Following Lambert and Rogers (2020), two concepts are distinguished: a **factor**, which is a connected structure

[1]Because our notion of structural containment is distinct from the standard model-theoretic notion of substructures, we are careful to avoid that term.

contained within a model, and a **window**, which is a structured collection of domain elements from which a factor may be derived. We begin by defining a window.

Given a (homogeneous) relation R of arity $a \geqslant 2$, i.e. $R\colon \mathcal{D}^a$, its a-windows are defined by the set

$$\mathscr{W}_a^R \triangleq \left\{\left\{\langle \overset{i}{x_i}, \overset{i+1}{x_{i+1}}\rangle \colon 1 \leqslant i < a\right\} \colon \langle x_1, \ldots, x_a\rangle \in R\right\}.$$

This effectively turns each tuple in the relation into a sequence of overlapping pairs that represent the edges of a (linear) directed graph version of that tuple. Each node in this graph is labeled by not only the domain element itself, but also an index so that cycles in the structure do not translate into cycles in the window. For instance, if 1 is a domain element and $\langle 1, 1\rangle$ appears in R, the only 2-window in the set of 2-windows that corresponds to this relation is

$$\left\{\langle \overset{1}{1}, \overset{2}{1}\rangle\right\}.$$

Discussing smaller windows is simple. Any connected subgraph of an a-window is also a window, of size equal to the number of nodes it contains.

For windows of size larger than the arity of the relation from which they are defined, we can use an

inductive definition:

$$\mathscr{W}_{k+1}^{R} \triangleq \Big\{ A \cup \langle \overset{j_{a-1}}{x_{a-1}}, \overset{k+1}{x_a} \rangle \colon A \in \mathscr{W}_{k}^{R} \text{ and } \langle x_1, \ldots, x_a \rangle \in R$$
$$\text{and } \{j_1, \ldots, j_{a-1}\} \subseteq \{1, \ldots, k\}$$
$$\text{and } \{\langle \overset{j_i}{x_i}, \overset{j_{i+1}}{x_{i+1}} \rangle \colon 1 \leqslant i < a-1\} \subseteq A$$
$$\text{and } (\exists y, \ell)\big[\langle \overset{j_{a-1}}{x_{a-1}}, \overset{\ell}{y} \rangle \in A \text{ or } \langle \overset{\ell}{y}, \overset{j_{a-1}}{x_{a-1}} \rangle \in A\big]$$
$$\text{and } \big(\forall j_a \in \{1, \ldots, k\}\big)\big[\langle \overset{j_{a-1}}{x_{a-1}}, \overset{j_a}{x_a} \rangle \notin A\big]\Big\}.$$

The conditions on the first line select a k-window and an element of the relation. The second line selects $a-1$ indices. The third line, which is never relevant for a binary relation, ensures that these indices form a path from the first to the last, and that this path is labeled by the appropriate domain elements. The fourth line accounts for binary relations, simply asserting that the selected index corresponds to an appropriate domain element. And finally the fifth ensures that edges in the model may only be repeated in the case of cycles.

In short, for each k-window, we find a linear subgraph (a path) that maps to the first $a-1$ elements of a tuple in R, then add an edge from the final node of this path to a newly constructed node representing the final domain element from that tuple.

The conditions assert that adding this new node does not simply repeat the construction of an already-existing path, while still allowing cycles to be iterated without bound. For example, given the 2-window described previously, the only valid 3-window formed from the same $\langle 1, 1 \rangle$ tuple is

$$\big\{\langle \overset{1}{1}, \overset{2}{1} \rangle, \langle \overset{2}{1}, \overset{3}{1} \rangle\big\}.$$

Both index 1 and index 2 provide valid attachment points for a $\langle 1, 1\rangle$ edge, but this edge has already been followed from index 1, so that attachment is ruled out by the condition on the fifth line. The result of this induction is that a window is a rooted, connected, acyclic graph of indexed domain elements, where the root is the unique node of in-degree zero.

Although only binary relations will be discussed in this work, this definition applies more generally. Consider $R_3 = \langle 1, 2, 3\rangle, \langle 1, 2, 4\rangle, \langle 2, 4, 5\rangle, \langle 3, 4, 5\rangle$. Two pairs can be chained: $\langle 2, 4, 5\rangle$ can overlay the right of $\langle 1, 2, 4\rangle$, or $\langle 1, 2, 4\rangle$ and $\langle 1, 2, 3\rangle$ can be overlaid at their left portions. So the only 4-windows of R_3 are

$$\left\{\overset{1}{1}\overset{2}{,2}\rangle, \langle\overset{2}{2}\overset{3}{,4}\rangle, \langle\overset{3}{4}\overset{4}{,5}\rangle\right\} \text{ and } \left\{\overset{1}{\langle 1}\overset{2}{,2}\rangle, \langle\overset{2}{2}\overset{3}{,4}\rangle, \langle\overset{2}{2}\overset{4}{,3}\rangle\right\} \text{ and } \left\{\overset{1}{\langle 1}\overset{2}{,2}\rangle, \langle\overset{2}{2}\overset{3}{,3}\rangle, \langle\overset{2}{2}\overset{4}{,4}\rangle\right\}.$$

It is possible that an alternative definition requiring less overlap might be preferred. Notice though that the latter two windows are identical graphs when the indices are ignored.

In general, there can be several windows that correspond to the same contained structure. Looking only at the domain elements represented unifies these multiple representations: the **factor at** a window x in the word model m (written $[\![x]\!]_m$) is the restriction of m to the domain elements in x. For instance, consider the signature

$$\mathscr{M}^{\triangleleft} = \langle \mathcal{D}; <, \mathrm{a}, \mathrm{b}, \mathrm{c}\rangle$$

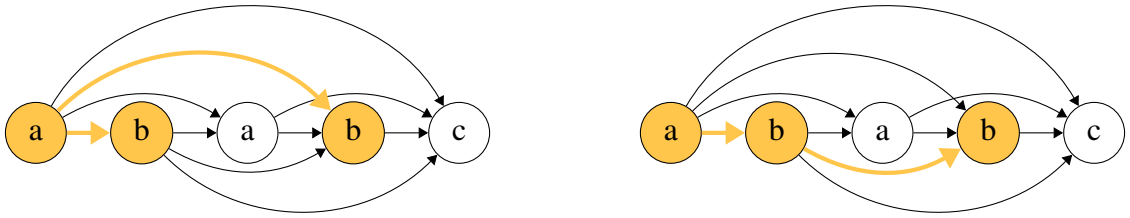


Figure 2.2: Two three-windows of “ababc” for the same factor.

and a word model over this signature

$$m = \Big\langle \{1,2,3,4,5\}; \{\langle x, y\rangle : x, y \in \mathcal{D} \text{ and } x < y\}, \{1,3\}, \{2,4\}, \{5\} \Big\rangle,$$

If we have a window x such that the domain elements included in x are all and only 1, 2, and 4, such as either of those in Figure 2.2, then the factor at x is the corresponding restriction

$$[\![x]\!]_m = m \restriction x = \Big\langle \{1,2,4\}; \{\langle 1,2\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}, \{1\}, \{2,4\}, \varnothing \Big\rangle.$$

The set of all k-factors of a model m is the set

$$\mathscr{F}_k(m) \triangleq \{[\![x]\!]_m : x \in \mathscr{W}_k\}.$$

where the windows are built over the ordering relations of m.

The word models shown to this point have been without explicit indication of domain boundaries. Often, however, we wish to consider models in which these boundaries are explicit, which we call **anchored models**. These can be formed by augmenting a model with new positions, self-related

under all ordering relations, that are labeled "⋊" and "⋉" for head and tail boundaries, respectively. This self-relation allows words shorter than k to be captured by k-windows without special treatment.

One might notice that a model that has a smaller number of domain elements than the arity of its ordering relation might have no factors at all by these definitions. While this is never a problem for anchored models, one might consider alternative constructions when using nonanchored models. The simplest is to construct the factors of the anchored models and then strip away the domain boundaries from the result. In any case, the use of anchored word models will be assumed throughout this text.

Several classes of subregular languages are defined by the model-theoretic means discussed in this section (Rogers et al., 2012; Lambert et al., 2021b). Factors built around the $<$ relation are piecewise factors, and those built around $\lhd$ are local factors. Whether one checks factors in isolation, sets of factors, or multisets of factors determines how finely one may partition the space of languages, and these dimensions form a piecewise-local subregular hierarchy, shown in Figure 2.3, based on that shown by Rogers and Lambert (2019a). The LTT class is defined using multisets of local factors, and TSL is placed off to the side indicating that it does not readily fit in with the rest of the hierarchy. It is defined by applying a strictly local (SL) grammar to a projection, and this projection step allows for more power. The next chapter describes all of these classes, focusing on explaining tier-based classes more generally and allowing us to more firmly locate TSL and newly created extensions thereof.

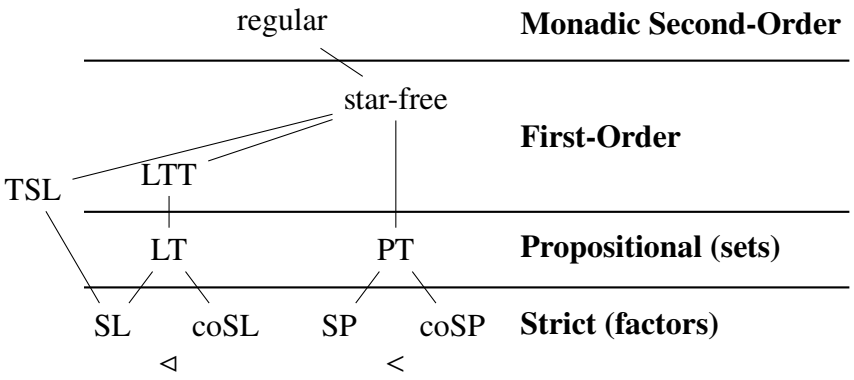


Figure 2.3: The piecewise-local subregular hierarchy.

2.4 Graphs and Finite-State Automata

A finite-state acceptor is a mechanism for deciding whether a structure belongs to a given set by reading the structure in some reasonable order and traversing through a finite set Q of states. Such an acceptor for strings over a finite alphabet Σ consists of its set of states, a transition function $\delta\colon \Sigma \times Q \to Q$, an initial state q_0, and a set $F \subseteq Q$ of accepting states. This is typically written as the five-tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$ (Rabin and Scott, 1959; Sakarovitch, 2009; Rogers et al., 2010). For instance, the following represents the set of strings over the set $\{a, b\}$ that contain an even number of occurrences of a:

$$\mathcal{A} = \langle \{a, b\}, \{q_1, q_2\}, \delta, q_1, \{q_1\} \rangle$$

$$\delta(a, q_1) = q_2$$

$$\delta(a, q_2) = q_1$$

$$\delta(b, q_1) = q_1$$

$$\delta(b, q_2) = q_2$$

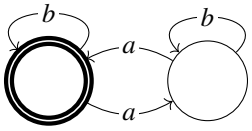


Figure 2.4: A finite-state string acceptor: doubly-outlined states are accepting, and the initial state is marked by the thick outline. All and only those strings which contain an even number of occurrences of a are accepted.

There are constructive proofs that these string acceptors are expressively equivalent to regular expressions and thus recognize all and only regular string languages (McNaughton and Yamada, 1960). Strings are canonically read left-to-right, but this class is closed under reversal so it follows that a right-to-left automaton would be equally expressive. If δ is a function, such an acceptor is called a deterministic finite-state automaton (DFA).

An automaton representing all and only those strings which contain an even number of occurrences of a is shown as a labeled directed graph in Figure 2.4. The alphabet is $\{a, b\}$. Each state of the acceptor is a node, and each element $\langle \sigma, q, r \rangle$ of δ is represented by an edge from q to r labeled σ. The accepting states are marked as such, and the initial state is denoted by an arrow from nowhere. A word w is accepted iff there is some path $q_o \to \cdots \to q_f$ for some $q_f \in F$ whose labels spell out w. For instance, "abba" is accepted by the DFA of Figure 2.4.

2.5 Transition Semigroups and Syntactic Monoids

At the very core, most mathematical structures consist of some set coupled with one or more operations. We will concern ourselves here with only the simplest of structures. A **semigroup** is a pair consisting of a set S and an operation $\cdot$ that satisfies certain special properties, $\langle S, \cdot \rangle$. There are two properties that make this pair a semigroup. First the set must be **closed** under the operation: given two elements x and y from S, it is necessarily the case that $x \cdot y \in S$ as well. Further the

operation must be **associative**: for three elements x, y, and z in S, it is necessarily the case that $(x \cdot y) \cdot z = x \cdot (y \cdot z)$. If the operation is understood from context, we often write xy instead of the full $x \cdot y$.

A **monoid** is a semigroup with another special property. If $\langle S, \cdot \rangle$ is a semigroup and there is some element $e \in S$ such that for all elements $x \in S$ it holds that $e \cdot x = x \cdot e = x$, then this semigroup is a monoid. The special element e is called the **identity**, and is often denoted by 1. For any semigroup S, one can create a monoid S^1 by simply adjoining a new element defined to act as 1.

An object is **free** if the only equations it satisfies are the ones that follow from the definition of that kind of object. That is, in a **free semigroup** one will find that for all x, y, and z it holds that $x(yz) = (xy)z$, but no other equations hold true. For instance, for all x, $x \neq xx \neq xxx \neq \cdots$, and $xy \neq yx$ except when $x = y$. Essentially this lets us think of words in an algebraic way. For a given finite alphabet Σ, the free semigroup over this alphabet is defined as follows:

- For each $x \in \Sigma$, $x \in \Sigma^+$.
- If $x \in \Sigma$ and $w \in \Sigma^+$, then $xw \in \Sigma^+$.
- Nothing else is in Σ^+.

Formally the free semigroup is Σ^+ under concatenation; if u and v are in Σ^+, then $u \cdot v = uv$. The free monoid Σ^* is Σ^+ with an adjoined identity, the empty string.

The **local subsemigroup** of S generated by an element $a \in S$ is the set $aSa = \{asa : s \in S\}$. If a is an idempotent then $aaa = a$ is in this local subsemigroup and $a \cdot asa = asa = asa \cdot a$, so it is a monoid whose identity is its generating idempotent a.

2.5.1 Equivalence Relations

Recall that a formal language is a possibly infinite set of words, in other words, it is some subset of Σ^*. One can in principle discuss this set as it stands, but often it is simpler to think about a smaller structure, the structure induced by considering the ways in which words behave when put together. After defining some notion of behavior, one can construct an equivalence relation between words such that words equivalent iff they have the same behavior under that definition.

For example, Nerode (1958) proposes using acceptable extensions to indicate behavior: for some fixed language L, a is equivalent to b under this notion, $a \overset{N}{\sim} b$, iff for all possible (suffixal) extensions v, it holds that $av \in L$ iff $bv \in L$. The Myhill-Nerode theorem states that a language is regular iff $\overset{N}{\sim}$ induces only finitely many classes of equivalent words, and moreover the equivalence classes correspond precisely to the states in the minimal complete deterministic finite-state automaton representing the language, (Hopcroft and Ullman, 1979).

Consider the language over $\Sigma = \{a, b\}$ represented by Figure 2.5, containing all and only those words which have an ab substring. There are precisely three equivalence classes under $\overset{N}{\sim}$: words that have contained an ab, $[\![ab]\!]$ corresponding to the rightmost state, words that end on a, $[\![a]\!]$ corresponding to the central state, and everything else, $[\![\varepsilon]\!]$ corresponding to the leftmost state. Notice that a and ba are both in $[\![a]\!]$. However $a \cdot a$ and $a \cdot ba$ lie in different classes: $[\![a]\!]$ and $[\![ab]\!]$,

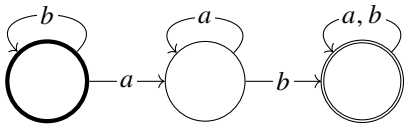


Figure 2.5: "Contains ab" as a string acceptor.

respectively. Unfortunately, the collection of resulting equivalence classes cannot form a semigroup, as concatenation is not well-defined over the classes.

A different, two-sided notion of behavior was introduced by Myhill and reported with attribution by Rabin and Scott (1959). Under this notion, given some fixed language L, two words a and b are equivalent, $a \overset{M}{\sim} b$, iff for all strings u and v in Σ^*, it holds that $uav \in L$ iff $ubv \in L$. Prefixal as well as suffixal extensions are considered. Under this notion, two equivalence classes under $\overset{N}{\sim}$ each split into two new ones under $\overset{M}{\sim}$. The words a and ba behave differently under prefixation by a, so rather than having a class consisting of all words that end on a, we have a class of words that end with a but begin with b which is separate from the class of words that end with a and begin with anything other than b. Similarly, words beginning with b and not containing a are distinct from words neither beginning with b nor containing a, as again a prefixal a will distinguish, say, b from ε.

Theorem 2.1. *The $\overset{M}{\sim}$ relation is compatible with concatenation.*

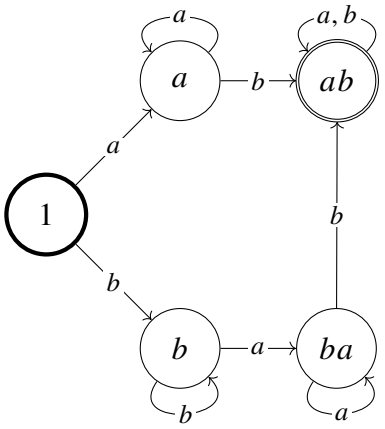
Proof. Let S be a semigroup and T some subset of that semigroup. Further let a, b, x, and y be elements of S such that for all elements u and v of S it holds that $uav \in T$ iff $ubv \in T$ and also it holds that $uxv \in T$ iff $uyv \in T$. That is, $a \overset{M}{\sim} b$ and $x \overset{M}{\sim} y$. Consider $u \cdot ax \cdot v$. This is equal to $u \cdot a \cdot xv$, which is in T iff $u \cdot b \cdot xv \in T$. By reassociating, this is equivalent to $ub \cdot x \cdot v$, which is in T iff $ub \cdot y \cdot v \in T$. That is, $uaxv \in T$ iff $ubyv \in T$ and $ax \overset{M}{\sim} by$. ∎

Essentially what this means is that the equivalence classes themselves form a semigroup structure, and the equivalence class of ε acts as an identity so it is a monoid. Not only is it a monoid, it is the smallest monoid capable of recognizing the language L over which the classes are defined (McNaughton and Papert, 1971). Denote the monoid over $\overset{M}{\sim}$-equivalence classes of Σ^* by $\Sigma^*/\overset{M}{\sim}$. Recognition in this sense means that there is a subset of $\Sigma^*/\overset{M}{\sim}$ such that the union of contained classes is exactly L.

2.5.2 Monoid Construction

McNaughton and Papert (1971) demonstrate that the $\overset{M}{\sim}$ equivalence relation can be constructed directly from a minimal DFA. For each string $x \in \Sigma^*$, construct the function $f_x : Q \to Q$ mapping each state $q \in Q$ to the state r that one reaches by following edges to spell out x from q. Start with the empty string corresponding to the identity function, and extend words by one symbol at a time until no new functions are generated. The result is $\Sigma^*/\overset{M}{\sim}$, often referred to as the **syntactic monoid** of L, or as the **transition monoid** of the automaton from which it was formed. Because this method associates strings to functions from state to state, there are at most Q^Q equivalence classes under $\overset{M}{\sim}$ if Q is the number of classes under $\overset{N}{\sim}$, and Holzer and König (2004) show that this upper bound is indeed reachable.

Applying this construction to the automaton depicted in Figure 2.5, one sees that the empty string is the identity function. Numbering the states 1, 2, 3 from left to right, one writes $\langle 1, 2, 3\rangle$ to denote this function, each position maps to itself. Then a corresponds to $\langle 2, 2, 3\rangle$, as upon reading an a, state 1 transitions to state 2, state 2 remains in state 2, and state 3 remains in state 3. Similarly b corresponds to $\langle 1, 3, 3\rangle$. Each of a and b has created a new function, so they must be extended. We



	1	a	b	ba	ab
1	1	a	b	ba	ab
a	a	a	ab	ab	ab
b	b	ba	b	ba	ab
ba	ba	ba	ab	ab	ab
ab	ab	ab	ab	ab	ab

Figure 2.6: Syntactic monoid of "contains ab".

see that aa corresponds to $\langle 2, 2, 3\rangle$, equivalent to a itself. Similarly bb is equivalent to b itself. But ab and ba are distinct, corresponding to $\langle 3, 3, 3\rangle$ and $\langle 2, 3, 3\rangle$, respectively. Finally we see that $aba = abb = ab$, $baa = ba$, and $bab = ab$, and no new functions are created. The resulting monoid is shown in Figure 2.6 both as a $\overset{M}{\sim}$-minimal acceptor (with equivalence classes in L marked) and as a multiplication table. The graph that underlies the acceptor is the **Cayley graph** of the monoid.

2.6 Conclusions

This chapter introduced the basic concepts from formal language theory, finite model theory, and abstract algebra which will appear throughout the remainder of this work. Alongside this, a well-studied subregular hierarchy was presented. The hierchary presented in Figure 2.3 is motivated by language-theoretic means. Extending this hierarchy is the focus of the following chapters, first demonstrating that TSL forms a natural basis for an entirely new branch of the hierarchy, rather than sitting alone, and then incorporating other well-studied classes of formal language into the picture.

Chapter 3: Classifying and Factoring Tier-Based Extensions of the Subregular Hierarchy

The piecewise-local subregular hierarchy, henceforth referred to as simply **the subregular hierarchy**, has been extensively studied for decades, with the local branch introduced by McNaughton and Papert (1971) and the piecewise branch stemming from Simon (1975). Local constraints are, as the name implies, good at capturing dependencies based on adjacent events, and can do so with even the simplest logics. Even some long-distance dependencies can be captured, such as "A and B do not occur in the same word", but there is no notion of directionality here. Piecewise constraints fall on the other extreme, easily representing certain types of long-distance dependencies but requiring at least first-order logic to be able to refer to adjacent events at all.

In order to more simply state some types of long-distance dependencies, and to account for some that piecewise constraints cannot, a third branch came into existence with the tier-based strictly local (TSL) class imposing adjacency on distant parts of a string (Heinz et al., 2011). A TSL description works by relativizing the concept of adjacency over some subset of the alphabet, referred to as the tier alphabet. Symbols outside this subset are ignored entirely.

Linguistic interest in the TSL class stems from its usefulness in describing long-distance dependencies, especially those that strictly piecewise constraints cannot handle such as blocked harmony patterns (Heinz, 2010a; Heinz et al., 2011; McMullin, 2016). For example, the liquid dissimilation pattern of Latin (Cser, 2010) is a sort of blocked harmony pattern, shown to be 2-TSL by McMullin (2016). This pattern can be shown to be strictly star-free when restricted to the local or piecewise branches of the subregular hierarchy. However, another important consideration for linguistics is learnability,

and the star-free class is not learnable (Gold, 1967). On the other hand, 2-TSL has been shown to be effectively learnable both by humans (McMullin, 2016) and by machines (Jardine and Heinz, 2016), and k-TSL for arbitrary k has been shown to be learnable as well (Jardine and McMullin, 2017).

The main formal result of Heinz et al. (2011) was a language-theoretic proof that TSL is a subclass of star-free. Originally TSL was defined by applying an erasing homomorphism to a string(set), projecting it to strings formed from the tier alphabet, then applying a strictly local filter to the result. This operational perspective is useful in describing the solution, but it can mask some insights. To provide more clarity, Lambert and Rogers (2020) provide an equivalent alternative definition based on model theory and a new class of ordering relations, and from this they develop language- and automata-theoretic characterizations of the class.

The present work extends this further, characterizing not only the TSL class but also relativized variants (introduced here) of the other classes in the subregular hierarchy. These characterizations also go further, including algebraic as well as automata-, language-, and model-theoretic results. This chapter uses these characterizations to extend the subregular hierarchy of Figure 2.3 to a richer scheme shown in Figure 3.1 in which all local classes have an associated tier-based relativization.

We begin in section 3.1 with an overview of the model-theoretic concepts that will be used, then section 3.2 provides definitions as well as model- and language-theoretic characterizations for all of the relativized classes. Section 3.3 provides automata-theoretic characterizations for some of the classes, deferring the rest to section 3.5 in which automata are converted to an algebraic structure. In this latter section, algebraic characterizations are given for each new class, primarily synthesizing classical results and applying them to the new structures. Algebraic characterizations are explored

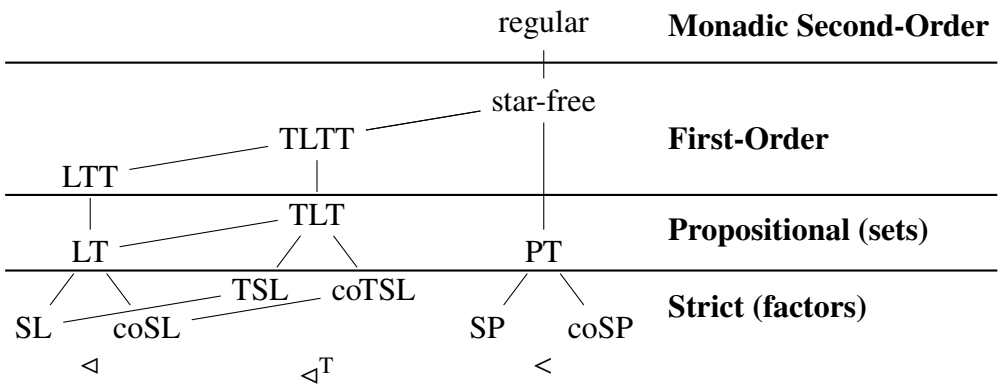


Figure 3.1: The piecewise-local subregular hierarchy with tiers.

even further in chapter 4. Closure properties are proved in section 3.4, using properties of automata to prove that some closures do hold, while using the language-theoretic characterizations provided earlier to demonstrate that some other potential properties do not.

3.1 Model Theoretic Descriptions

In this section we discuss a model-theoretic treatment of relativized adjacency. The relation defined here is then used to define variants of each class of the subregular hierarchy, where the relativized variant of the strictly local class is identical to the tier-based strictly local class of Heinz et al. (2011).

Recall that the successor relation is the transitive reduction of the general precedence relation. Instead of reduction, we might consider restricting the domain to all and only those elements that satisfy a certain predicate φ.

$$x <^{\varphi} y \triangleq \varphi(x) \wedge \varphi(y) \wedge (x < y).$$

This is essentially the general precedence relation on the model's projection to those elements that

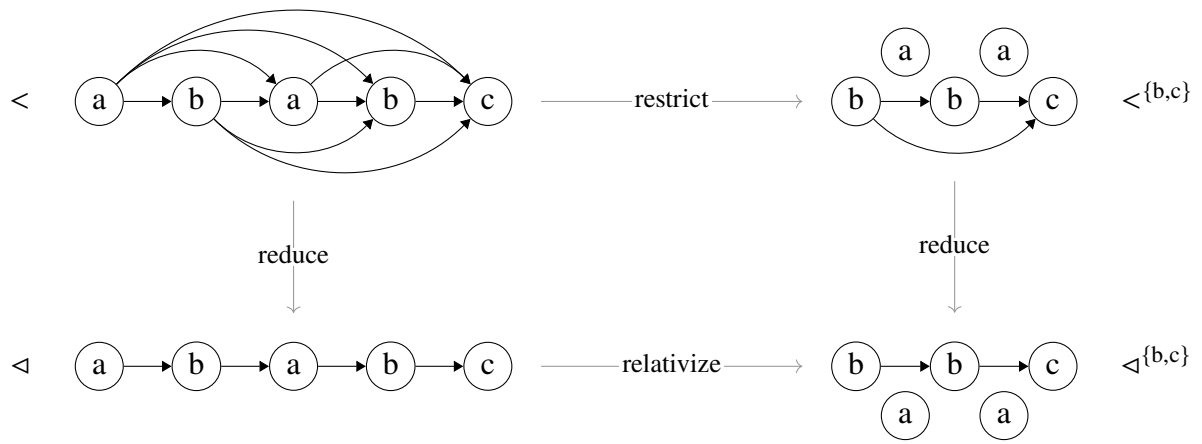


Figure 3.2: Word models for "ababc" using general precedence, immediate successor, and relativized variants of each, showing the relationships among these relations. Domain elements that are not ordered are pulled aside from the structure. In these examples, the alphabet is $\Sigma = \{a, b, c\}$, and the salient symbols for the relativized relations are $T = \{b, c\}$

satisfy φ. We can of course combine these to obtain the reduction of this restriction,

$$x \lhd^{\varphi} y \;\triangleq\; (x <^{\varphi} y) \wedge \neg(\exists z)[x <^{\varphi} z <^{\varphi} y],$$

which defines a relativized successor relation. Given some alphabet Σ and some set of salient symbols $T \subseteq \Sigma$, the predicate $\varphi(x) = T(x) = \bigvee_{\tau \in T} \tau(x)$ results in a relation that acts as if it were the successor relation on the model's projection to T, reminiscent of the original definition of the TSL^{T} class. We refer to this specific kind of relativization as **projective relativization**. The particular definitions used here guarantee that the factors of a model under a projectively relativized relation are exactly those under the corresponding nonrelativized one of the model's projection.

Figure 3.2 shows how the general precedence and immediate successor relations, as well as their relativized variants, relate to one another. These relationships do not only hold for these relations. In fact any binary relation whose transitive closure is antisymmetric may be relativized by taking

the transitive reduction of a restriction of its transitive closure. The antisymmetry requirement ensures uniqueness of the result (Aho et al., 1972). Throughout this text we will consider only projective relativization, though with appropriate choice of φ the more general treatment can be shown to capture the structure-sensitive tier-based strictly local class of De Santo and Graf (2019) or the domain- and interval-based strictly piecewise classes of Graf (2017). One important property specific to projective relativization is that the unordered elements truly have no effect on the ordered ones. Whether a domain element is included in the restriction is decided entirely by the unary relation that labels that point, and so the unordered elements can be freely removed, shuffled, or inserted at any point.

For any projectively relativized relation, we assume for notational convenience that the boundary symbols $\rtimes$ and $\ltimes$ are considered salient if they are present in the model. So rather than writing $\lhd^{\{\rtimes,\mathrm{b},\mathrm{c},\ltimes\}}$ we simply write $\lhd^{\{\mathrm{b},\mathrm{c}\}}$ instead. Figure 3.3 shows an anchored word model for "ababc" under the $\lhd^{\{\mathrm{b},\mathrm{c}\}}$ relation along with all of its nonempty 3-factors. Because the domain boundaries are self-related, assuming the domain elements of "$\rtimes$ababc$\ltimes$" are 1 through 7 in order, the windows

$$\left\{\langle \overset{1}{1}, \overset{2}{1}\rangle, \langle \overset{2}{1}, \overset{3}{3}\rangle\right\} \quad \text{and} \quad \left\{\langle \overset{1}{1}, \overset{2}{1}\rangle, \langle \overset{1}{1}, \overset{3}{3}\rangle\right\}$$

both refer to the relativized prefix "$\rtimes$b" of this string. (Either instance of domain element 1 is a valid attachment point for a $\langle 1, 3\rangle$ edge.) The "a" elements are unordered and do not occur in any factor.

3.2 Language-Theoretic Characterizations

A grammar is some representation of a mechanism by which the membership of a string in a stringset may be decided. A class of grammars is denoted by $\mathbb{G}$. The characteristic function $\mathbb{1} : \mathbb{G} \times \mathscr{M} \to \mathbb{B}$

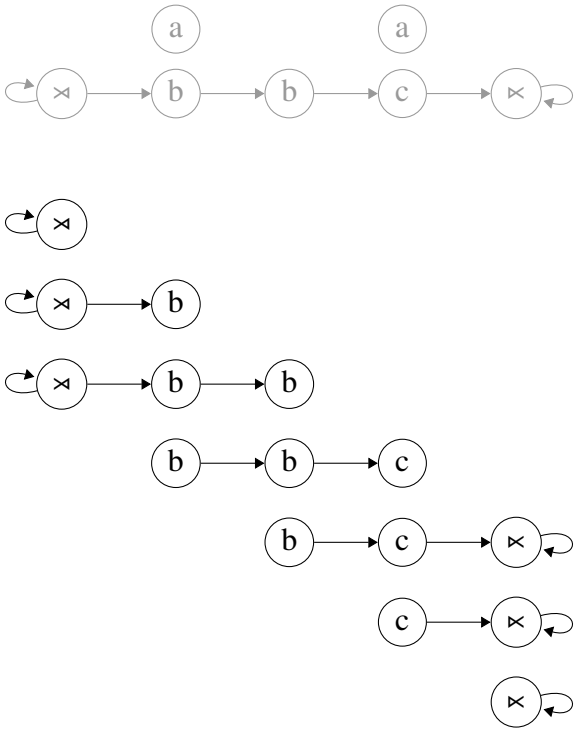


Figure 3.3: All 3-factors of "⋊ababc⋉" under the $\lhd^{\{b,c\}}$ relation. Factors that appear shorter than this are formed from windows that repeat the boundary symbols.

is

$$\mathbb{1}_G(m) \triangleq \begin{cases} \top & \text{if } m \text{ satisfies } G, \\ \bot & \text{otherwise.} \end{cases}$$

Here, $\mathbb{B}$ represents the binary Boolean ring, $\top$ is true, $\bot$ is false. Functions of more than one argument are sometimes written with their first argument as a subscript; $\mathbb{1}_G$ can be thought of as the partial application of the curried form of $\mathbb{1}$. The stringset represented by G is the set of all and only those strings whose models satisfy it:

$$\mathscr{L}(G) \triangleq \left\{w : \mathbb{1}_G\left(\mathscr{M}(w)\right)\right\}.$$

Two grammars G_1 and G_2 are equivalent iff they are extensionally equal, that is, $\mathscr{L}(G_1) = \mathscr{L}(G_2)$.

3.2.1 Strict locality

The original definition of the TSL class from Heinz et al. (2011) was operational. A stringset L is k-TSL$^{\mathrm{T}}$ iff there exists a k-SL grammar such that L contains all and only those strings whose projection to T satisfy this grammar.

A grammar for a k-SL stringset is simply a subset of $\mathscr{F}_k(\Sigma^*)$, that is, a set of k-factors, where an anchored model m satisfies G iff each of its k-factors occurs in G (McNaughton and Papert, 1971):

$$\mathbb{1}_G(m) \;=\; \mathscr{F}_k^{\triangleleft}(m) \subseteq G.$$

Because the symbols not in T never appear in any factors under $<^{\mathrm{T}}$ or $\triangleleft^{\mathrm{T}}$, any two strings with the same projection to T will have the same set of factors under these relations. Specifically, if $\pi_{\mathrm{T}}(m)$ represents the projection of m to T, it holds that m and $\pi_{\mathrm{T}}(m)$ have exactly the same set of factors under these relations. Moreover, if $\mathrm{T} = \Sigma$ it follows by definition that these are equivalent to their nonrelativized analogues. Therefore the only difference between a k-SL grammar and a corresponding k-TSL one is interpretation:

$$\mathbb{1}_G(m) \;=\; \mathscr{F}_k^{\triangleleft^{\mathrm{T}}}(m) \subseteq G.$$

We will see that this is indeed always the case, so from this point on characteristic functions will be given without the relation specified. Using $\triangleleft$ yields the local class, $\triangleleft^{\mathrm{T}}$ its relativization.

With this in mind, we turn to the language-theoretic characterization of the SL class: closure under substitution of suffixes (Rogers and Pullum, 2011, see also De Luca and Restivo, 1980). A stringset

satisfies suffix substitution closure iff there is some k such that for any two strings $w_1 = u_1xv_1$ and $w_2 = u_2xv_2$ where $|x| \geqslant k - 1$, if both w_1 and w_2 are in the set, then so is $w_3 = u_1xv_2$. Lambert and Rogers (2020) provide a similar characterization for the TSL class.

Definition 3.1 (Preprojective suffix substitution closure: PSSC)**.** A stringset L over the alphabet Σ is closed under T-preprojective suffix substitution (T-PSSC) iff there is some natural number k such that for any two strings $w_1 = u_1x_1v_1$ and $w_2 = u_2x_2v_2$ where $\pi_T(x_1) = \pi_T(x_2)$ and $|\pi_T(x_1)| \geqslant k - 1$, if both w_1 and w_2 are in L, then so is $w_3 = u_1x_1v_2$.

Notice that Σ-PSSC is equivalent to standard suffix substitution closure, as the strings are necessarily equal to their projections. On its own though, T-PSSC is not sufficient to characterize TSL. The other necessary condition is that symbols not in T be freely insertable and deletable, as previously discussed for the relativized relations.

Theorem 3.1. *A stringset L over an alphabet Σ is TSL iff there is some subset $T \subseteq \Sigma$ such that symbols not in T are freely insertable and deletable and L is closed under* T-PSSC*.*

Proof. To prove that such a stringset is TSL, suppose there exists such a T. Then by T-PSSC, the projection of L to T is SL. Further, by insertion and deletion closure of non-T symbols we guarantee that w is in L iff its projection to T is. Together, these facts show that L is TSL.

To prove the reverse implication, suppose that L is TSL. Then it is k-TSL^T for some k and T. By definition of TSL, w is in L iff its projection to T is, and so symbols not in T are freely insertable and deletable. Because L is TSL^T, its projection to T is SL and thus satisfies suffix substitution closure. Further since L is closed under insertion of symbols not in T, it is then also closed under T-PSSC. ■

In order to prove that a stringset is TSL, it suffices to provide a grammar. To prove that a stringset cannot be TSL, one can find some set of symbols that are not freely both insertable and deletable, then form strings from those symbols alone that violate PSSC. For instance, a constraint that forbids sequential (not necessarily adjacent) occurrences of "a . . b . . a" is not TSL because neither "a" nor "b" is freely insertable (so they are necessarily in T) and PSSC is violated by the following words:

$$\begin{array}{ll} \mathrm{a}\,\overbrace{\mathrm{b}\,.\,.\,\mathrm{b}}^{k-1}\,\mathrm{b} & (\in) \\ \mathrm{b}\,\mathrm{b}\,.\,.\,\mathrm{b}\,\mathrm{a} & (\in) \\ \hline \mathrm{a}\,\mathrm{b}\,.\,.\,\mathrm{b}\,\mathrm{a} & (\notin). \end{array}$$

3.2.2 Complements

If an SL stringset contains all strings that satisfy a grammar G, the complement of this set is all strings that do not satisfy G. Since a model satisfies G iff all of its factors are in G, it follows that the model does not satisfy G iff it has at least one factor not in G. We can use the grammar of the complemented SL stringset as the grammar for a COSL stringset. The result is a collection of factors, at least one of which is required to appear in every word. The resulting characteristic function then is

$$\mathbb{1}_G(m) \;=\; G \cap \mathscr{F}_k(m) \neq \varnothing.$$

Definition 3.2 (Ideal containment: IC)**.** A stringset L is k-COSL iff L contains all and only those strings w that themselves contain at least one factor $f \in \mathscr{F}_k^{\triangleleft}(w)$ such that every string x that contains that factor is also in L:

$$\left\{x\colon f \in \mathscr{F}_k^{\triangleleft}(x)\right\} \subseteq L.$$

If there exists some k for which L is k-cosl, then L is cosl.

In ic, the factor f is the (not necessarily unique) factor that caused dissatisfaction of the corresponding sl grammar. Then in order to show that a stringset is not k-cosl, it suffices to find a (preferably small) word $w \in L$ and a set of words S such that $\mathscr{F}_k(w) \subseteq \mathscr{F}_k(S)$, yet no member of S is in L. For instance we can show that a constraint that bans occurrence of the substring "ab" is not cosl because ic is violated by the following:

$$\begin{aligned} w &= \quad \mathrm{a}\,.\,.\,\mathrm{a} && (\in) \\ S &= \Big\{ \underbrace{\mathrm{a}\,.\,.\,\mathrm{a}}_{k-1}\ \mathrm{b}\ \underbrace{\mathrm{a}\,.\,.\,\mathrm{a}}_{k-1} \Big\} && (\notin). \end{aligned}$$

The factors of w are $\{{\rtimes}\mathrm{a}^i, \mathrm{a}^i{\ltimes} : 0 \leqslant i < k\}$. Each of these factors occurs in the single word in S, and so the presence of any given factor is not sufficient to guarantee acceptance. This extends trivially to **preprojective ideal containment**.

Definition 3.3 (Preprojective ideal containment: pic)**.** A stringset L is k-cotsl iff there exists some $\mathrm{T} \subseteq \Sigma$ such that L contains all and only those strings w that themselves contain at least one factor $f \in \mathscr{F}_k^{\triangleleft \mathrm{T}}(w)$ such that every string x that contains that factor is also in L:

$$\big\{x : f \in \mathscr{F}_k^{\triangleleft \mathrm{T}}(x)\big\}.$$

And L is cotsl iff it is k-cotsl for some k.

Since the factors of w under $\lhd^{\mathrm{T}}$ are the same as those of its T-projection under $\lhd$, this is equivalent in every way to stating that L is k-coTSL$^{\mathrm{T}}$ iff its T-projection is k-coSL and it is closed under insertion and deletion of symbols not in T. Further the order of complementation and relativization is immaterial, as will become clear in section 3.3.

3.2.3 Local testability

Much as the SL stringsets consist of words containing only permitted factors, the locally testable (LT) ones consist of words whose set of factors is permitted (McNaughton and Papert, 1971). This allows a mechanism to reject "ab" even in the case of accepting "abab", whose 2-factors are {⋊a, ab, b⋉} and {⋊a, ab, ba, b⋉}, respectively. Due to their subset relationship, a mechanism capable only of 2-SL distinctions could not do this, though each of the three other possible combinations of acceptance and rejection of these two strings is possible under 2-SL. For testable stringsets, a grammar is a set of permitted sets of factors, and its characteristic function is

$$\mathbb{1}_G(m) \;=\; \mathscr{F}_k(m) \in G.$$

Rogers and Pullum (2011) state that a stringset is LT iff it is closed under **local test invariance**, where given two strings w_1 and w_2 such that $\mathscr{F}_k^{\lhd}(w_1) = \mathscr{F}_k^{\lhd}(w_2)$, the first is in the set iff the second is as well. This extends trivially to **preprojective local test invariance**.

Definition 3.4 (Preprojective local test invariance: PLTI)**.** A stringset L is TLT iff there exists some $\mathrm{T} \subseteq \Sigma$ and some k such that given two strings w_1 and w_2 such that $\mathscr{F}_k^{\lhd^{\mathrm{T}}}(w_1) = \mathscr{F}_k^{\lhd^{\mathrm{T}}}(w_2)$, the first is in L iff the second is as well.

By the same reasoning employed in discussion of COTSL, this is equivalent in every way to stating that L is k-TLT$^{\mathrm{T}}$ iff its T-projection is k-LT and it is closed under insertion and deletion of symbols not in T.

3.2.4 Threshold testability

The LT class is characterized by sets of factors. But a set is merely a structure that describes each possible element by a Boolean value, whether or not that element is included. One might consider a natural extension of this structure which saturates its count of occurrences not at 1 but at some arbitrary value t. This is exactly what Beauquier and Pin did when defining the **locally threshold testable** (LTT) stringsets in 1989. We denote this generalized structure by

$$\mathscr{F}_{k,t}(m) \triangleq \wr [\![x]\!]_m : x \in \mathscr{W}_k \int_t .$$

For threshold testable stringsets, a grammar is a set of permitted multisets whose characteristic function is

$$\mathbb{1}_G(m) \;=\; \mathscr{F}_{k,t}(m) \in G.$$

The characterization of LTT of course is **local threshold test invariance**, where given two strings w_1 and w_2 such that $\mathscr{F}^{\lhd}_{k,t}(w_1) = \mathscr{F}^{\lhd}_{k,t}(w_2)$, the first is in the set iff the second is as well. This extends trivially to **preprojective local threshold test invariance**.

Definition 3.5 (Preprojective local threshold test invariance: PLTTI)**.** A stringset L is TLTT iff there exists some $\mathrm{T} \subseteq \Sigma$ and some k and t such that given two strings w_1 and w_2 such that $\mathscr{F}^{\lhd^{\mathrm{T}}}_{k,t}(w_1) = \mathscr{F}^{\lhd^{\mathrm{T}}}_{k,t}(w_2)$, the first is in L iff the second is as well.

By the same reasoning employed in discussion of cotsl and tlt, this is equivalent in every way to stating that L is k,t-tltt$^{\mathrm{T}}$ iff its T-projection is k,t-ltt and it is closed under insertion and deletion of symbols not in T.

Under the definitions of windows and factors used in this text, this actually counts prefixes and suffixes of length less than k more than once, since several windows might correspond to the same factor. Importantly, for fixed k the count is consistent for prefixes (suffixes) of a given length, and since there is at most one length-n prefix (suffix) in any valid word model, this overcounting does not affect the possible distinctions.

3.2.5 Piecewise relativizations

Using general precedence instead of successor in the definition of the sl class yields the strictly piecewise (sp) class. Characterized by Rogers et al. (2010) as those stringsets closed under deletion (see also Haines, 1969 for an earlier treatment of stringsets closed under deletion), we can show that a stringset is tsp iff it is sp. In fact, unlike reduction, relativization provides neither more nor less expressive power in precedence-based models.

By definition $<^{\Sigma}$ is equivalent to $<$, thus all nonrelativized stringsets are also trivially relativized ones. More interesting is the reverse. Recall from Figure 3.2 that the relativization of the $<$ relation is in fact merely a restriction, and so $\mathscr{F}_k^{<^{\mathrm{T}}}(m) \subseteq \mathscr{F}_k^{<}(m)$. If a factor occurs on the restriction, then it also occurs in the nonrestricted model. Therefore a full piecewise model must be at least as powerful as its relativization, but since T can be equal to Σ they are in fact equivalent.

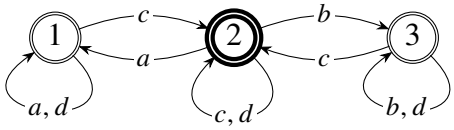
For this same reason, when the factor width is fixed at $k = 1$ all of the projectively relativized classes are equivalent to their nonrelativized analogues.

3.3 Automata

In this section we discuss characterizations of the relativized classes and constructions of their constituent stringsets in terms of deterministic finite-state automata (DFAs).

Recall that a DFA is a directed graph that represents a machine that computes the well-formedness of a string with respect to some regular stringset, and is represented by a five-tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$, Σ an alphabet, where Q is a set of states, $\delta \colon \Sigma \times Q \to Q$ a transition function which represents edges in the graph, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of accepting states. A DFA is **complete** iff δ is total. A word w is accepted by the DFA iff there is some path $q_0 \to \cdots \to q_f$ for some $q_f \in F$ (an **accepting path** from q_0) whose labels spell out w.

Let $\sim$ represent the equivalence relation over states in Q under which $q_1 \sim q_2$ iff for all $u \in \Sigma^*$ there is an accepting path from q_1 labeled u whenever such a path exists from q_2 and vice-versa. This is **Nerode equivalence**. By the Myhill-Nerode theorem (Nerode, 1958), one can construct a **minimal** DFA from a given one by replacing each state in Q by its Nerode-equivalence class, the element of $Q/\sim$ that contains it. A minimal DFA might have a unique **nonaccepting sink**, a state q from which there are no accepting paths for any string. A **canonical** DFA is one that is minimal and has had its nonaccepting sink (if any) removed.



$$\mathbb{C}\mathrm{T} = \bigcap\big\{\{a, d\}, \{c, d\}, \{b, d\}\big\} = \{d\}$$

Figure 3.4: A canonical DFA for which d is a nonsalient symbol.

3.3.1 Characterizations

Every relativized class discussed in this text is closed under insertion and deletion of symbols not in T. In other words, such symbols provide no information regarding the well-formedness of a word. It follows then that from a given state q, the state reached by following an edge labeled by such a symbol must be in the same Nerode-equivalence class as q itself. Thus in a canonical DFA, the nonsalient symbols are exactly those that form self-loops on all states simultaneously.

$$\mathbb{C}\mathrm{T} = \bigcap_{q \in Q} \{\sigma \in \Sigma \colon \delta_\sigma(q) = q\}.$$

In other words, these are exactly the symbols σ such that the set of fixed points of δ_σ is the entirety of Q. Figure 3.4 shows a canonical DFA that represents a TSL stringset in which d is not a salient symbol. The fixed points for a, b, c, and d are $\{1\}$, $\{3\}$, $\{2\}$, and $\{1, 2, 3\}$, respectively.

Given a canonical automaton for a language L, the automaton for its T-projection is formed by restricting δ to $\Sigma \setminus \mathbb{C}T$. As discussed previously, with this choice of T a stringset is in the relativized variant of a given class iff its T-projection is in that class. The T found this way is in fact the smallest set for which this holds, though some stringsets might permit several possible values for T. For example, the stringset Σ^* is TSL^P for every $P \subseteq \Sigma$.

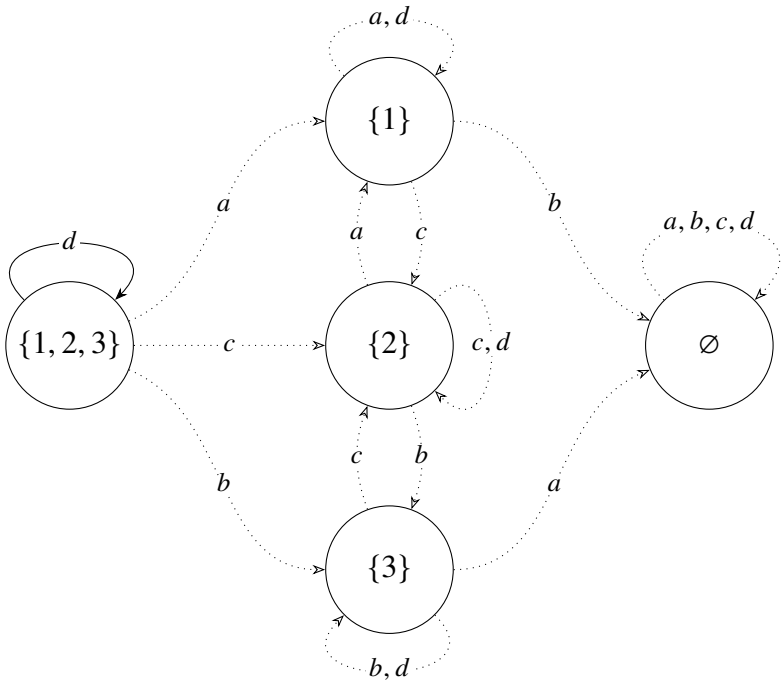


Figure 3.5: The powerset graph that corresponds to the DFA of Figure 3.4, with the cycle marked that proves this stringset is not SL. Notice that if d is removed, there is no such cycle.

Once the projection has been found, any of the numerous existing methods for determining class membership can be used. Most of these tests are based on the algebraic interpretation of the syntactic semigroup corresponding to the automaton, which will be discussed further in section 3.5. However, for the SL class we can use a result of Edlefsen et al. (2008, see also Caron, 1998). Given a canonical DFA $A = \langle \delta, q_0, F \rangle$, construct its **powerset graph** by defining $\delta^{\mathscr{P}} \colon \Sigma \times \mathscr{P}(Q) \to \mathscr{P}(Q)$:

$$\delta^{\mathscr{P}}_{\sigma}(S) \triangleq \{\delta_{\sigma}(s) \colon s \in S\}.$$

The stringset represented by A is SL iff the graph formed by $\delta^{\mathscr{P}}$ contains no cycles that iterate a node whose label contains two or more elements. A powerset graph of a non-SL stringset is shown in Figure 3.5 with the offending path marked.

Edlefsen et al. (2008) also provide a more efficient algorithm in terms of pairs of states. However, the powerset graph construction also allows extraction of a grammar for the target stringset from the automaton itself (Rogers and Lambert, 2019b). Since a stringset is coSL iff its complement is SL, this serves as an automata-theoretic method to decide membership in that class as well. Thus by projecting an automaton to the appropriate tier alphabet, not only can we show that the target stringset is TSL or coTSL, but we can also obtain a canonical grammar for this stringset if it is.

3.3.2 Constructions

The projection mechanism of section 3.3.1 is invertible. Given a DFA representing a stringset L whose alphabet is some $\mathrm{T} \subseteq \Sigma$, the preprojection of L to Σ is given by adding self-edges on each symbol $\sigma \in \Sigma \setminus \mathrm{T}$ to every state. If the subregular class of L is known, then the result lies in the corresponding relativized class.

However, the purpose of these projectively relativized classes is to represent those stringsets in which only certain symbols are salient. It would be meaningful then to employ **alphabet-agnostic automata** (AAA). Such automata test for the occurrence of a factor within a target string of unknown or unspecified alphabet by augmenting the set of symbols relevant to the factor with a **wildcard** symbol ⍰, much like Beesley and Karttunen (2003). For our purposes, we consider a wildcard that matches all and only those symbols not already listed in the alphabet, like the @ of Hulden (2009). The empty language is represented by a single state which is non-accepting, bearing a self-loop labeled ⍰, which is the only member of the alphabet. The universal acceptor is identical, except its single state is accepting.

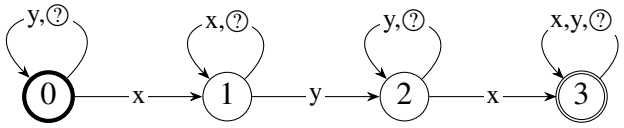


Figure 3.6: Canonical AAA for the factor "xyx" under $<$.

Factors under $<$ (piecewise factors) are the simplest to construct. Given a factor $f = \sigma_1 \dots \sigma_n$ under this relation, the AAA is defined in essentially the same way that Rogers et al. (2010) form a DFA:

$$Q = \{0, \dots, n\}$$

$$\Sigma = \{\sigma_1, \dots, \sigma_n, ⍰\}$$

$$q_0 = 0$$

$$F = \{n\}$$

$$\delta(\sigma, q) = \begin{cases} q+1 & \text{if } q < n \text{ and } \sigma = \sigma_{q+1}, \\ q & \text{otherwise.} \end{cases}$$

An example is shown in Figure 3.6. Note that since ⍰ is by definition never included in the factor, edges on this symbol are always self-loops. In other words, this construction provides a clear picture as to why SP (and extensions thereof) and TSP (and extensions) should be identical.

For factors under $\lhd$ (local factors), any of the common approaches to substring-matching suffice, including that of Knuth et al. (1977). However, a naïve method shown here demonstrates some properties of AAA. Fully-anchored factors of the form $f = \rtimes\sigma_1 \dots \sigma_n\ltimes$ look like piecewise factors,

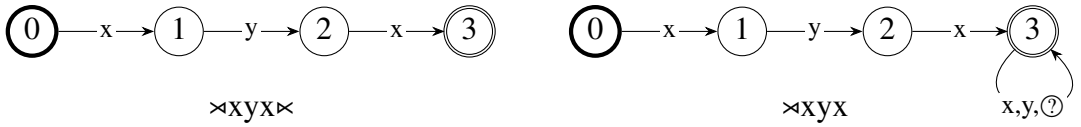


Figure 3.7: Canonical automata for head-anchored factors.

except edges to a non-accepting sink $\perp$ replace the self-loops:

$$Q = \{\perp, 0, \ldots, n\}$$

$$\delta(\sigma, q) = \begin{cases} q+1 & \text{if } q < n \text{ and } \sigma = \sigma_{q+1} \\ \perp & \text{otherwise.} \end{cases}$$

For head-anchored but not tail-anchored factors, the difference is that $\delta(\sigma, n) = n$ rather than $\perp$. Figure 3.7 shows a canonical (trimmed) AAA constructed for each of "⋊xyx⋉" and "⋊xyx".

These automata can then be concatenated after a universal acceptor to represent tail-anchored and free factors, as in Figure 3.8, but this concatenation requires an extra step compared to standard DFA operations: the two inputs to any binary operation must be made **compatible**. Two AAA are compatible iff they have the same alphabet. Since ⍰ represents any symbol not already in the alphabet, a symbol σ is added by placing new edges labeled by σ in parallel with any existing ⍰ edges. Thus to make two AAA compatible, their alphabets should be extended in this way to the union of their individual alphabets. Two compatible automata can be combined using standard DFA operations, treating ⍰ as just another symbol.

To fix the alphabet of an AAA to a specific set Σ, simply extend it as necessary and then remove any ⍰ edges. Relativizing an automaton over some tier alphabet T is a process of fixing its alphabet to

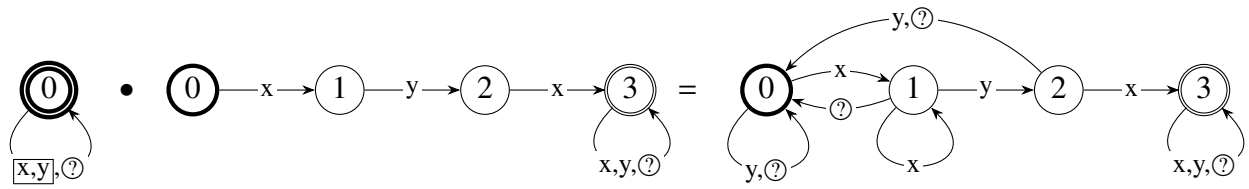


Figure 3.8: Canonical AAA for the free factor "xyx" under $\triangleleft$ constructed via concatenation. The boxed x,y in the first operand are symbols that needed to be added for compatibility.

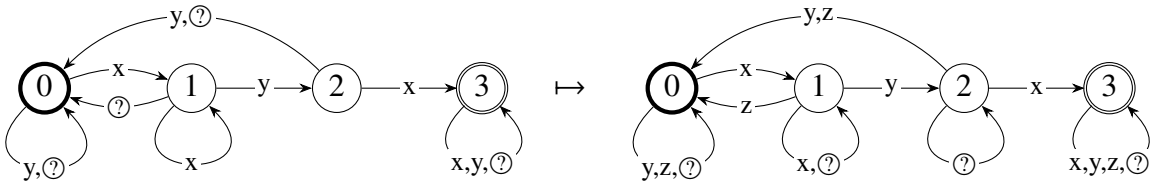


Figure 3.9: Relativizing "xyx" to T = {x, y, z}: Add 'z' in parallel to any existing ? edges, remove those ? edges, then add new ? edges as self-loops everywhere.

T, then adding ? edges back in as self-loops on every state. For example, Figure 3.9 shows how this process modifies the factor of Figure 3.8 to consider only 'x', 'y', and 'z' salient.

3.4 Closure Properties

In this section we constructively prove some closure properties of relativized classes via their automata-theoretic characterizations, and use the language-theoretic ones to provide counter-examples to other closure properties.

3.4.1 Products

The intersections and unions of automata are both formed from **the product construction**. Given two automata $A = \langle \delta, q_0, F \rangle$ and $A' = \langle \delta', q_0', F' \rangle$, one constructs the product

$$A \otimes_{op} A' \triangleq \left\langle \delta^{\otimes}, \langle q_0, q_0' \rangle, F^{op} \right\rangle,$$

where the new transition function is defined pointwise:

$$\delta_\sigma^{\otimes}(\langle q, q' \rangle) \triangleq \left\langle \delta_\sigma(q), \delta'_\sigma(q') \right\rangle.$$

The set of accepting states is

$$F^{op} \triangleq \left\{ \langle q, q' \rangle : q \in F \ op \ q' \in F' \right\},$$

where *op* is "and" for intersection or "or" for union. If σ labels a self-loop on every state of both operands, then this product construction guarantees that the same will hold in the result. By construction then, if $\complement\mathrm{T}$ is the set of nonsalient symbols for A and $\complement\mathrm{T}'$ for A', then $\complement\mathrm{T}^{\otimes} \subseteq \complement\mathrm{T} \cap \complement\mathrm{T}'$ is a set for this product.[1] Notably, if A and A' represent stringsets in a relativized class where $\mathrm{T} = \mathrm{T}'$ and the underlying class is closed under union (intersection), the result of the union (intersection) of A and A' remains in the same relativized class with the same set of salient symbols.

Theorem 3.2. *The* $\textsc{tlt}^{\mathrm{T}}$ *and* $\textsc{tltt}^{\mathrm{T}}$ *classes for fixed* T *are closed under both union and intersection.*

Proof. Because LT and LTT are each closed under both union and intersection, the product construction guarantees that $\textsc{tlt}^{\mathrm{T}}$ and $\textsc{tltt}^{\mathrm{T}}$ for fixed T are as well. ∎

For the same reasons, the same holds for $\textsc{tsl}^{\mathrm{T}}$ under intersection and $\textsc{cotsl}^{\mathrm{T}}$ under union.

Note that these closures rely on equality of the sets of salient symbols. Consider the TSL stringset whose projection to $\{\mathrm{a}, \mathrm{b}\}$ contains no "ab" factor and the TSL stringset whose projection to $\{\mathrm{a}, \mathrm{c}\}$

[1] Since the result is not necessarily canonical, a larger $\complement\mathrm{T}^{\otimes}$ (thus a smaller T) may also exist.

contains no "aca" factor. In the intersection, none of "a", "b", or "c" is freely insertable, so each must be salient no matter what Σ is. Then even though the two strings

$$\overbrace{\mathrm{c}\,..\,\mathrm{c}}^{k}\,\mathrm{b}\,\overbrace{\mathrm{c}\,..\,\mathrm{c}}^{k}\,\mathrm{a}\,\overbrace{\mathrm{c}\,..\,\mathrm{c}}^{k} \quad (\in)$$

$$\mathrm{c}\,..\,\mathrm{c}\;\mathrm{a}\;\mathrm{c}\,..\,\mathrm{c}\;\mathrm{b}\;\mathrm{c}\,..\,\mathrm{c} \quad (\notin)$$

have exactly the same k-factors and exactly the same counts for each, the first is in the intersection while the second is not. This is a violation of PLTTI (see page 40). Thus the intersection of these two stringsets is not even TLTT, and so by containment it cannot be TLT or TSL. For a more direct proof that this intersection is not TSL, consider the following violation of PSSC:

$$\mathrm{ac}\,\overbrace{\mathrm{c}\,..\,\mathrm{c}}^{k-1}\,\mathrm{ca} \quad (\in)$$

$$\mathrm{b}\;\mathrm{c}\,..\,\mathrm{c}\;\mathrm{b} \quad (\in)$$

$$\mathrm{ac}\;\mathrm{c}\,..\,\mathrm{c}\;\mathrm{b} \quad (\notin).$$

In this example, the result of PSSC is a string whose projection to $\{\mathrm{a}, \mathrm{b}\}$ contains an ab factor, which should be forbidden.

3.4.2 Complements of automata

To find the complement of a complete minimal DFA, simply invert the notion of acceptance. That is, map $\langle \delta, q_0, F \rangle$ to $\langle \delta, q_0, Q \setminus F \rangle$.

Theorem 3.3. *A regular stringset L and its complement can be defined by expressions over the same tier of salient symbols as one another.*

Proof. Because L is regular, it can be represented as a complete minimal DFA, and this DFA will be associated with some set of salient symbols. The complement operation does not affect the transition function δ, so the set of self-loops in the result is exactly the same as that in the input. It follows then that the complement of L has the same set of salient symbols as L itself. ∎

Then if the underlying class is closed under complementation (as is the case for LT and LTT) the corresponding relativized variant is so closed as well. Moreover, since a relativization is formed by merely adding self-loops everywhere, the order of relativization and complementation is immaterial. The two operations cannot interfere with one another.

3.4.3 Some non-closures

Having shown that, for fixed T, TLT^{T} and TLTT^{T} are closed under all Boolean operations and TSL^{T} and coTSL^{T} are closed under intersection and union, respectively, we now show that TSL^{T} is not closed under union or complement, and that coTSL^{T} is not closed under intersection or complement.

Consider two $\text{TSL}^{\{\text{a,b}\}}$ stringsets: one which bans the occurrence of "ab" on the projection to $\{\text{a}, \text{b}\}$, and another which bans the occurrence of "ba" on this projection. The union of these two stringsets allows the occurrence of either "ab" or "ba", but not both. Then the following is a violation of PSSC:

$$\frac{\begin{array}{ll} \text{ab}\overbrace{\text{b}\,..\,\text{b}}^{k-1} & (\in) \\ \text{b}\,..\,\text{b ba} & (\in) \end{array}}{\text{ab b}\,..\,\text{b ba} \quad (\notin).}$$

The complement of the first of these, that "ab" must occur on the projection to $\{a, b\}$, is also not TSL. The following is a violation of PSSC:

$$\begin{array}{ll} \overbrace{\mathrm{a}\,..\,\mathrm{a}}^{k-1}\,\mathrm{ab} & (\in) \\ \mathrm{ab}\ \mathrm{a}\,..\,\mathrm{a} & (\in) \\ \hline \mathrm{a}\,..\,\mathrm{a} & (\notin). \end{array}$$

Now consider two COTSL$^{\{a,b\}}$ stringsets, one which requires that some "ab" occurs on the projection to $\{a, b\}$, and another that requires that some "ba" occurs on this projection. Their intersection (requiring both to occur) is not COTSL, as the following violates PIC:

$$\begin{aligned} w &= \mathrm{a}\,\overbrace{\mathrm{b}\,..\,\mathrm{b}}^{k-1}\,\mathrm{a} \quad (\in) \\ S &= \{\mathrm{a}\,\mathrm{b}\,..\,\mathrm{b}, \\ &\qquad \mathrm{b}\,..\,\mathrm{b}\,\mathrm{a}\} \quad (\text{each } \notin), \end{aligned}$$

since the collective factorset of S is a superset of the factors of w. And of course, the complement of the first of these stringsets, banning occurrences of "ab" on the $\{a, b\}$-projection, is also not COTSL, because, as shown in section 3.2.2, the stringset of the projection is not COSL.

From this we have shown that TLTT^{T} and TLT^{T} are closed under all Boolean operations, while TSL^{T} and COTSL^{T} are closed only under intersection and union, respectively. We have also shown that intersection and union closures hold only when both stringsets have the same set of salient symbols.

3.5 Algebra

Many of the algorithms that decide whether a given DFA represents a stringset from a particular class actually make use of the **syntactic semigroup** associated with the DFA. Given a complete minimal DFA $A = \langle \delta, q_0, F \rangle$, recall that $\delta \colon \Sigma \times Q \to Q$ can be viewed as $\hat{\delta} \colon \Sigma \to (Q \to Q)$ and define $\gamma \colon \Sigma^* \to (Q \to Q)$ as follows:

$$\gamma(w) \triangleq \begin{cases} \gamma(v) \circ \hat{\delta}(\sigma) & \text{if } w = \sigma v \text{ for some } \sigma \in \Sigma \text{ and } v \in \Sigma^* \\ \text{id} & \text{otherwise.} \end{cases}$$

Here, 'id' refers to the identity function. The syntactic semigroup is then the semigroup under flipped composition of the following functions:

$$S(A) \triangleq \left\{ \gamma(w) \colon w \in \Sigma^+ \right\}.$$

Then if $a = \gamma(u)$ and $b = \gamma(v)$, we have $ab = b \circ a = \gamma(v) \circ \gamma(u) = \gamma(uv)$. Since $\gamma(u)\gamma(v) = \gamma(uv)$, the semigroup operation is a homomorphism.

The star-free stringsets are those whose syntactic semigroup is finite and has no nontrivial subgroups (Pin, 1997).

Recall from section 3.3 that the set of nonsalient symbols, $\complement\mathrm{T}$, is the set of symbols that form self-loops on every state of a DFA. Translated to the algebraic domain, these are all and only those symbols σ for which $\gamma(\sigma) = \text{id}$. Sometimes we denote id by 1. If $1 \in S$, then we also say S is a monoid.

Lemma 3.1. *If S is the syntactic semigroup of a star-free stringset and $a, b \in S$ are elements such that $ab = 1$, then $a = b = 1$.*

Proof. Suppose a and b are elements of S such that $ab = 1$, and that S is the syntactic semigroup of a star-free stringset. Then $1 = ab = (a(ab))b$, and by continuing this process we see that $a^n b^n = 1$ for all n. Schützenberger (1965) proves that because S is star-free, there exists some m such that $a^m = a^{m+1}$. Then we have $1 = a^m b^m = a^{m+1} b^m = a a^m b^m = a1 = a$, and by a similar argument we see that $b = 1$ as well. Therefore $a = b = 1$. ∎

This means that if S corresponds to a star-free stringset, then every w such that $\gamma(w) = \mathrm{id}$ is composed entirely of symbols from $\complement\mathrm{T}$. We now define the **projected subsemigroup** as

$$X(A) \triangleq \left\{\gamma(w) \colon w \in \mathrm{T}^+\right\}.$$

Theorem 3.4. *If S corresponds to a star-free stringset, then the projected subsemigroup X of S is equal to $S \setminus \{1\}$.*

Proof. Suppose $s \in S$. Then $s = \gamma(w)$ for some $w \in \Sigma^*$, and by definition if w is the string $\sigma_1 \dots \sigma_n$ then $s = \gamma(\sigma_n) \circ \cdots \circ \gamma(\sigma_1)$.

Suppose $s \neq 1$. Because whenever $\sigma \in \complement\mathrm{T}$ it holds that $\gamma(\sigma) = 1$, it follows that $\gamma(w) = \gamma\big(\pi_{\mathrm{T}}(w)\big)$. By definition, if $|\pi_{\mathrm{T}}(w)| \neq 0$ then $s \in X$ as well. If the length of this projection were 0, then $s = \gamma\big(\pi_{\mathrm{T}}(w)\big)$ would be 1, and since by assumption $s \neq 1$, this cannot be. Therefore, $s \in X$.

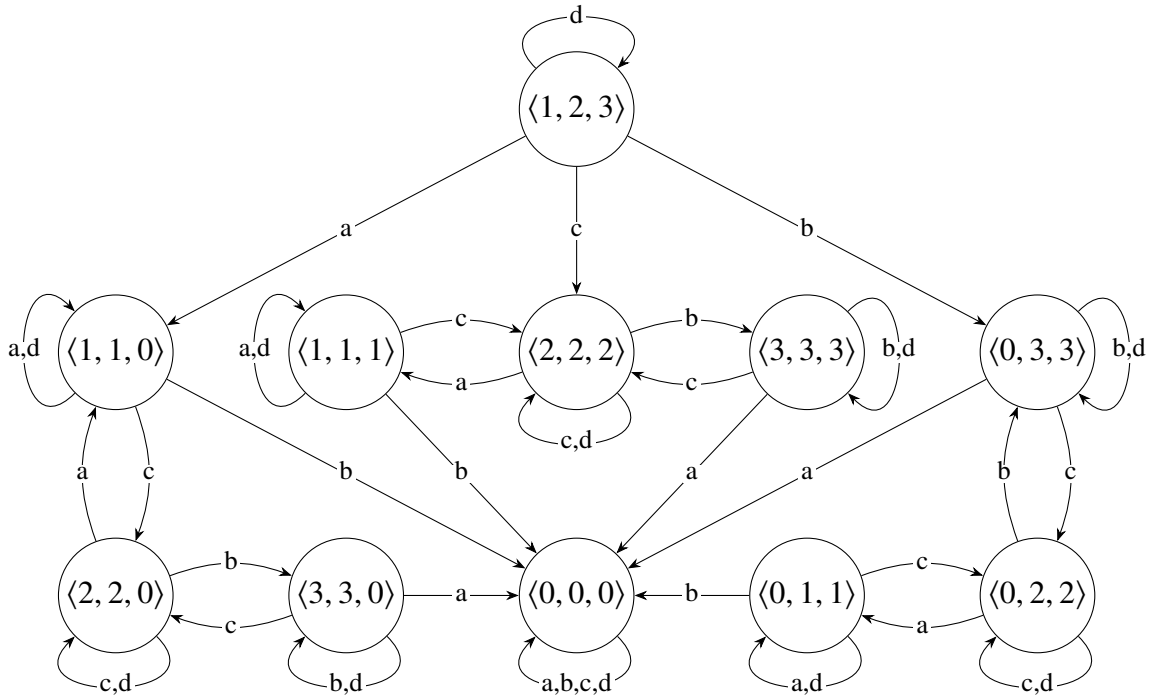


Figure 3.10: The syntactic semigroup corresponding to the DFA of Figure 3.4. Each function is represented by the image of $\langle 1, 2, 3\rangle$. (Each tuple additionally has a fourth entry that is always labeled 0 and is omitted here for brevity.) The corresponding projected subsemigroup lacks the identity as well as every edge labeled d.

On the other hand, suppose $s = 1$. Then by Lemma 3.1, we have $\{\sigma_1, \ldots, \sigma_n\} \subseteq \complement\mathrm{T}$. Then $w = \sigma_1 \ldots \sigma_n$ is not in T^* and by definition is excluded from X.

Thus if S is star-free, X contains all and only the nonidentity elements of S. ∎

Because S and X are finite and generated by Σ or T, respectively, we can visualize them as edge-labeled directed graphs just like a DFA. If A is the DFA shown in Figure 3.4 on page 43 augmented with a nonaccepting sink labeled 0, then S is the semigroup shown in Figure 3.10.

3.5.1 Strictly local stringsets and their complements

The SL and coSL classes (and their relativizations) cannot be characterized algebraically solely on the basis of their syntactic semigroups. Because the syntactic semigroup is generated exclusively from the transition function δ of a DFA, an automaton and its complement are associated with the same semigroup. This fact has been noted previously by McNaughton (1974). However, if information describing whether each state is accepting or rejecting (state parity) is retained then an algebraic characterization is possible.

De Luca and Restivo (1980) provide a characterization based on Schützenberger's (1975) concept of a **constant**. Let A be a subset of some semigroup S, and let $c \in S$. Then c is a constant for A if for all $p, q, r, s \in S \cup 1$ it holds that whenever $pcq \in A$ and $rcs \in A$ it follows that $pcs \in A$. If S is freely generated from some finite alphabet, De Luca and Restivo prove that A is SL iff all sufficiently long words of S are constants for A. This is equivalent to the suffix-substitution closure discussed in section 3.2.1. Of course if all such words are instead constants for the complement of A, then A is coSL rather than SL.

In this case, A is the stringset being tested, the language generated by the syntactic semigroup under observation. If instead these properties hold for the projected subsemigroup, then the stringset is TSL or coTSL, respectively. But again, no algorithm can distinguish a stringset and its complement given only an unadorned syntactic semigroup.

3.5.2 Locally testable stringsets

The characterization for LT and therefore TLT is due to Brzozowski and Simon (1973). First, note that an element x of a semigroup S is called **idempotent** iff $x = xx$. An **idempotent semigroup**

is a semigroup such that all of its elements are idempotent. Given the syntactic semigroup S of a stringset L, Brzozowski and Simon proved that L is LT iff for all idempotent elements e of S it holds that the subsemigroup eSe is a commutative idempotent monoid. Note that the monoid requirement is immaterial, as eSe will always be a semigroup with identity. Namely, for any $s \in S$, we see that $eee \cdot ese = (ee)(ee)se = eese = ese$ and similarly $ese \cdot eee = ese$, so eee (which is itself simply e) is the identity.

Theorem 3.5. *A stringset L is TLT iff for all idempotent elements e of X, the projected subsemigroup of its syntactic semigroup, it is the case that eXe is a commutative idempotent semigroup.*

This holds because the projected subsemigroup is equivalent to the syntactic semigroup of the projection, and a stringset is TLT iff its projection is LT by definition.

3.5.3 Locally threshold testable stringsets

The characterization for LTT and therefore TLTT is due to Beauquier and Pin (1989). They define a variety (collection) of semigroups $\mathbf{V}$ to contain all and only those aperiodic semigroups S such that if e and f are idempotent, $p = es_1f$, $q = fs_2e$, and $r = es_3f$, it holds that $pqr = rqp$. They then prove that a language L is LTT iff its syntactic semigroup S is a member of $\mathbf{V}$. By expansion, this means that for all idempotents e, f of S and for all $s_1, s_2, s_3 \in S$, it holds that $es_1fs_2es_3f = es_3fs_2es_1f$.

The aperiodicity requirement is meaningful, as there do exist stringsets that satisfy the latter requirement without even being star-free. Consider the set of strings of even length over a unary alphabet, $(aa)^*$. Its syntactic semigroup consists of two elements, 1 and a, such that $aa = 1$. It

necessarily holds that $es_1fs_2es_3f = es_3fs_2es_1f$ under any instantiation of these variables, because this is a commutative monoid.

Theorem 3.6. *Let L be a stringset and X be the projected subsemigroup of its syntactic semigroup. Then L is* TLTT *iff for all idempotent elements* e, f *of X, and for all* s_1, s_2, s_3 *in X, it holds that* $es_1fs_2es_3f = es_3fs_2es_1f$.

This holds because the projected subsemigroup is equivalent to the syntactic semigroup of the projection, and a stringset is TLTT iff its projection is LTT by definition.

3.6 Conclusions

We introduced several new classes to the piecewise-local subregular hierarchy building from the model-theoretic characterization of TSL by Lambert and Rogers (2020) and noting that projective relativization does not affect the expressivity of classes based on general precedence. We provided model-, language-, and automata-theoretic as well as algebraic characterizations for each new class. This establishes a clearer notion of relativized adjacency based on the salience of individual symbols, and informs linguistic theory as it pertains to long-distance dependencies in phonology. The newly established classes are integrated into the piecewise-local subregular hierarchy in Figure 3.1.

The topic of learning is not covered in this chapter. But we would be remiss to ignore that the unifying concept of relativized adjacency provides a straightforward mechanism to extend known learning results for the TSL class to TLT and TLTT (see McMullin, 2016; Jardine and Heinz, 2016; Jardine and McMullin, 2017). This applies even to the online TSL learning mechanism described in Chapter 6.

A Haskell library[2] containing the automata-theoretic and algebraic decision algorithms has been created. With this one can construct regular and subregular stringsets from factors, or import OpenFST-format automata, and determine which, if any, subregular classes the result occupies. The factor-based constructors make use of the alphabet-agnostic automata described in section 3.3.2.

The class formed by intersecting multiple TSL constraints over different tier alphabets, MTSL (De Santo and Graf, 2019), is not explored here. Under this model-theoretic view, the relativized successor relation $\lhd^{\mathrm{T}}$ is parameterized by the alphabet over which it is relativized, so different instantiations of this relation are as different as the $\lhd$ and $<$ relations. With this in mind, future work in understanding these interactions will shed light on the strictly piecewise-local class discussed in Rogers and Lambert (2019a), and vice-versa. Moreover, Aksënova and Deshmukh (2018) discuss attested interactions of multiple tiers; one might ask which kinds of combinations (if any) allow intersection-closure to be maintained. Counterexamples exist for each type of interaction (disjoint, overlapping, and subset configurations), but perhaps some smaller portion of the constraint space may combine more readily.

The characterizations provided here of the relativized variants of the subregular classes rely heavily on the fact that $\lhd^{\mathrm{T}}$ and $<^{\mathrm{T}}$ are the results of a projective relativization. Another direction for future work then would be accounting for arbitrary nonprojective relations, which would improve our understanding of De Santo and Graf's structure-sensitive TSL class and its (threshold) testable extensions, or Graf's (2017) domain- and interval-based strictly piecewise classes.

Finally, some of the subregular classes have been explored in application to trees (Rogers, 1997), and some extended to transducers (Chandlee, 2014; Ji and Heinz, 2020). It would be interesting to

[2]Software available at `https://hackage.haskell.org/package/language-toolkit`

see how relativization applies to these cases as well.

Chapter 4: Monoid Varieties and a Subregular Spiral

This chapter uses algebraic fundamentals to construct a subregular hierarchy that subsumes the complement-closed classes of the pieecwise-local subregular hierarchy discussed in previous chapters and incorporates in a unified manner several other classes of formal languages known from the field of theoretical computer science. These latter classes are characterized based on definability in various fragments of various types of formal logic. For instance, The Büchi-Elgot-Trakhtenbrot theorem states that regular languages correspond precisely to those definable in monadic second-order logic with successor or general precedence (Büchi, 1960; Elgot, 1961; Трахтенброт, 1962). Further, connections between first-order logic and star-free languages have long been established (Thomas, 1982), so any fragment of these will necessarily produce some subclass of the full class. This chapter incorporates discussion of the fragments of quantifier-free first-order logic described by Rogers and Lambert (2019a) as well, among others, the two-variable restrictions discussed by Thérien and Wilke (1998) and Krebs et al. (2020) to construct a full unified hierarchy.

We begin by discussing Green's relations (Green, 1951) and forming a basic classification hierarchy from first principles founded upon those relations. From there, more classes are constructed by either adding restrictions to find subclasses or removing restrictions to find superclasses. Once the basic hierarchy is in place, we discuss a general type of constraint relaxation that duplicate the entire hierarchy, and then duplicate it once more by introducing relativized adjacency. The resulting hierarchy subsumes the well-known subregular classes and provides a simple mechanism for extension.

4.1 Green's Relations and a Basic Hierarchy

In abstract algebra, James A. Green is known for his relations that characterize the structure of a semigroup. He first defines $a \mathrel{\mathscr{R}} b$ iff $aS^1 = bS^1$, $a \mathrel{\mathscr{L}} b$ iff $S^1a = S^1b$, and $a \mathrel{\mathscr{J}} b$ iff $S^1aS^1 = S^1bS^1$ (Green, 1951). Two others are defined from these: $a \mathrel{\mathscr{H}} b$ iff both $a \mathrel{\mathscr{L}} b$ and $a \mathrel{\mathscr{R}} b$, and $a \mathrel{\mathscr{D}} b$ iff there exists some c such that $a \mathrel{\mathscr{L}} c$ and $c \mathrel{\mathscr{R}} b$. Each of these is an equivalence relation: reflexive ($a \sim a$), symmetric ($a \sim b \leftrightarrow b \sim a$), and transitive (if $a \sim b$ and $b \sim c$ then $a \sim c$). A monoid is **trivial under** an equivalence relation $\sim$ iff $a \sim b$ implies that $a = b$. It is immediately apparent from their definitions that if $a \mathrel{\mathscr{R}} b$ or if $a \mathrel{\mathscr{L}} b$ then $a \mathrel{\mathscr{J}} b$. One perhaps less obvious implication is as follows.

Lemma 4.1. *If $a \mathrel{\mathscr{D}} b$ then $a \mathrel{\mathscr{J}} b$.*

Proof. Suppose $a \mathrel{\mathscr{D}} b$. Then there exists some element c such that $a \mathrel{\mathscr{L}} c$ and $c \mathrel{\mathscr{R}} b$. Then $c \in S^1a$ and there exists an s such that $c = sa$. And also $b \in cS^1$ and there exists a t such that $b = ct = sat$. Then $b \in S^1aS^1$. By a similar argument, $a \in S^1bS^1$, and therefore $a \mathrel{\mathscr{J}} b$. ∎

And less obvious still is Green's own Theorem 3:

Theorem 4.1 (Green, 1951)**.** *If every element $a \in S$ has finite order, then $\mathscr{J} = \mathscr{D}$.*

Specifically this means that for any finite semigroup, $\mathscr{J}$ and $\mathscr{D}$ coincide. Because this work is focused solely on the syntactic semigroups of regular languages, which are necessarily finite (Rabin and Scott, 1959), these relations will be used interchangeably, choosing whichever is more convenient in a given proof.

There are some aspects of the multiplication table for a semigroup that can be gleaned from the equivalence of elements under Green's relations. One useful result is as follows:

Lemma 4.2 (Miller and Clifford, 1956)**.** *For any idempotent e and any element a of S, if $e \mathscr{R} a$ then $ea = a$ and if $e \mathscr{L} a$ then $ae = a$.*

Proof. Suppose $a, e \in S$ such that $ee = e$. If $a \mathscr{R} e$, then $a \in eS^1$. In other words, there is some s such that $a = es$. By expansion, $ea = ees = es = a$. And by a similar argument, $ae = a$ if $a \mathscr{L} e$. ∎

A special relationship exists between elements related to products that include them.

Lemma 4.3. *In a finite semigroup, if $s \mathscr{J} sx$ then $s \mathscr{R} sx$ and if $s \mathscr{J} xs$ then $s \mathscr{L} xs$.*

Proof. Let s and sx be $\mathscr{J}$-related elements of a semigroup S. Then clearly $s \cdot x = sx$ and $sx \in sS^1$. Because $s \mathscr{J} sx$, there exist u and v such that $s = usxv$. Then $s = u \cdot usxv \cdot xv = \cdots = u^\omega s(xv)^\omega$. By idempotence then, $s = u^\omega u^\omega s(xv)^\omega = u^\omega s$. But now $sx \cdot v(xv)^{\omega-1} = u^\omega sx \cdot v(xv)^{\omega-1} = u^\omega s(xv)^\omega = s$. It follows that $s \mathscr{R} sx$. A similar argument shows that if $s \mathscr{J} xs$ then $s \mathscr{L} xs$. ∎

Corollary 4.1. *If $a \mathscr{J} a^2$ then $a \mathscr{H} a^2$.*

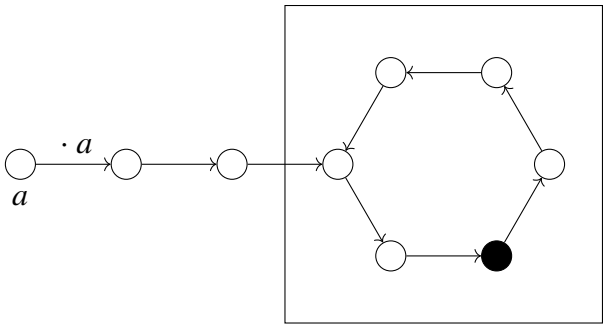


Figure 4.1: Every element a of a finite semigroup S eventually generates a group.

Another property concerns subgroups. Consider the subsemigroup generated by an element a. That is, the elements a, $a^2 = aa$, $a^3 = aaa$, etc. In a finite semigroup, there are of course only finitely many such elements. In other words, there exist some positive natural numbers m and n such that $a^m = a^{m+n}$. Then a^{mn} is idempotent[1] and the powers of a along the cycle back to this element form a group (a monoid in which every element x has an inverse x^{-1} where $xx^{-1} = x^{-1}x = 1$). For instance in Figure 4.1, $m = 4$, $n = 6$, and a^{24} (the highlighted element) is indeed idempotent. Denote this element a^ω.

Lemma 4.4. *If x and y are elements of a group, then $x \mathscr{H} y$.*

Proof. Let x and y be elements of a group. Then $x = y \cdot y^{-1}x$ and $y = x \cdot x^{-1}y$, and it follows that $x \mathscr{R} y$. By a similar argument, $x \mathscr{L} y$. By definition then, $x \mathscr{H} y$. ∎

A semigroup has only trivial subgroups iff it is $\mathscr{H}$-trivial. These groups are precisely the $\mathscr{H}$-classes. An $\mathscr{H}$-trivial semigroup is **aperiodic**. We can construct a five-class hierarchy of finite monoids from Green's relations and the universal relation under which all elements are related. Denote the class of monoids trivial under the latter by **1**, and those trivial under one of Green's relations by that

[1] This holds for every multiple of n that is greater than m.

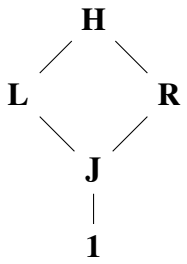


Figure 4.2: A basic five-class hierarchy below star-free.

relation's letter in boldface. The resulting hierarchy is shown in Figure 4.2. Note that Schützenberger (1965) shows that a language is star-free iff it is aperiodic, so this hierarchy is centered below star-free.

Given the Cayley table of a finite semigroup, one can easily construct the equivalence classes under Green's relations. If the rows associated with two elements a and b contain the same set of elements (including the element itself), then $a \mathscr{R} b$. Similarly if their columns contain the same set of elements, then $a \mathscr{L} b$. Recall that $\mathscr{H}$ and $\mathscr{D}$ are derived directly from these, and that for a finite semigroup $\mathscr{J} = \mathscr{D}$. One can construct a so-called "egg-box" diagram for a finite semigroup by placing each $\mathscr{D}$-class into a block, subdivided into a grid whose rows are $\mathscr{R}$-classes and whose columns are $\mathscr{L}$-classes. These blocks are then arranged into a Hasse diagram by their $\leqslant_J$ preorder: if $S^1aS^1 \subsetneq S^1bS^1$ then the block containing a is considered to be less than b. It is impossible to climb to a higher $\mathscr{J}$-class — multiplying on the left or right will leave the class unchanged or strictly smaller. Intuitively, falling in the $\leqslant_{\mathscr{J}}$ order results from discovery of a salient factor (Colcombet, 2011). Idempotent elements, those x where $xx = x$, are denoted by a star in the diagram.

Consider the monoid shown in Figure 4.3, the syntactic monoid of the language in which all words contain an a that is not followed by a $b \ldots c$ subsequence over the alphabet $\Sigma = \{a, b, c, d\}$. At left

	1	a	b	c	ab	bc	cb
1	1	a	b	c	ab	bc	cb
a	a	a	ab	a	ab	bc	ab
b	b	a	b	bc	ab	bc	bc
c	c	a	cb	c	ab	bc	cb
ab	ab	a	ab	bc	ab	bc	bc
bc	bc	a	bc	bc	ab	bc	bc
cb	cb	a	cb	bc	ab	bc	bc

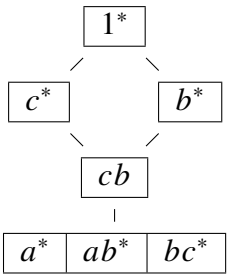


Figure 4.3: The syntactic monoid corresponding to the language in which all words contain an a not followed by $b \ldots c$. The symbol d is in $[1]$.

is its Cayley table; one notes that the a row contains the following set of elements: $\{a, ab, bc\}$, exactly the same set as the ab and bc rows. That is, these elements are in the same $\mathscr{R}$-class (and thus the same $\mathscr{D}$-class). The remaining four elements have distinct rows: 1 has the entire monoid, b has $\{a, b, ab, bc\}$, c has $\{a, c, ab, bc, cb\}$, and cb has $\{a, ab, bc, cb\}$. Each column is distinct, so no two elements share an $\mathscr{L}$-class. We see that cb is the only element that is not idempotent; it squares to bc and thus does not get a star. The two-sided ideal of 1 is the entire monoid, firmly situated at the top of our partial order by subset. That of c is $\{a, c, ab, bc, cb\}$, of b is $\{a, b, ab, bc, cb\}$, of cb is $\{a, ab, bc, cb\}$, and finally that of a is $\{a, ab, bc\}$.

This diagram immediately tells us about the complexity of the language it represents. There are distinct elements that share a block, a $\mathscr{J}$-class, so the represented language is not $\mathscr{J}$-trivial. Specifically they share a row, an $\mathscr{R}$-class, so it is also not $\mathscr{R}$-trivial. However, no two elements share a column within a block, so the language is $\mathscr{L}$-trivial (and thus $\mathscr{H}$-trivial).

Consider now the language over $\Sigma = \{a, b, c, d\}$ shown in Figure 4.4 where every word begins with a, and the first consonant, if any, is b. This language is $\mathscr{R}$- and $\mathscr{H}$-trivial, as no elements share a row, but it is not not $\mathscr{L}$- or $\mathscr{J}$-trivial, as some elements do share a column.

	1	a	b	c	ab
1	1	a	b	c	ab
a	a	a	ab	c	ab
b	b	b	b	b	b
c	c	c	c	c	c
ab	ab	ab	ab	ab	ab

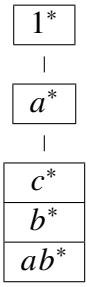


Figure 4.4: The syntactic monoid corresponding to the language in which all words begin with a and whose first consonant is b. The symbol d is in $[c]$.

In both of the examples discussed so far, one might notice that every element in a $\mathscr{D}$-class containing an idempotent is itself idempotent. This is not a coincidence. First, some terminology. An element a of a semigroup S is **regular** iff there exists some element x such that $axa = a$. A theorem of von Neumann (1936) reprinted by Green (1951) and Miller and Clifford (1956) is that the following are equivalent:

1. a is regular,

2. $a \mathscr{L} e$ for some idempotent e, and

3. $a \mathscr{R} e$ for some idempotent e.

Proof. (1 $\Rightarrow$ 2) If a is regular then there exists some x such that $axa = a$. Then $xaxa = xa$. $x \cdot a = xa$ and $a \cdot xa = a$, so $a \mathscr{L} xa$.

(2 $\Rightarrow$ 1) Let $a \mathscr{L} e$ for some idempotent e. Then $xa = e$ for some x and by lemma 4.2 it holds that $ae = a$. Then $a = ae = axa$ and thus a is regular.

(1 $\Leftrightarrow$ 3) A similar argument suffices. ∎

	1	a	b	c	ba	ab
1	1	a	b	c	ba	ab
a	a	a	ab	c	ab	ab
b	b	ba	b	b	ba	ab
c	c	a	c	c	a	ab
ba	ba	ba	ab	b	ab	ab
ab	ab	ab	ab	ab	ab	ab

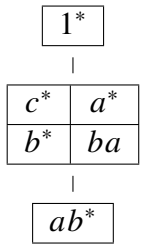


Figure 4.5: Egg-box for "contains ab".

An immediate implication of this is that within a given $\mathscr{D}$-class, if there is any idempotent at all then there is at least one in each row and column of that block in the egg-box diagram. Such a $\mathscr{D}$-class is itself called **regular**, and a class lacking an idempotent is **irregular**. It is not the case in general that every regular element is idempotent, as evidenced by the language depicted in Figure 4.5 over $\Sigma = \{a, b, c\}$ in which all words contain an ab substring. Notice that ba is regular but not idempotent.

Lemma 4.5. *S is an $\mathscr{L}$-trivial semigroup iff then $b(ab)^\omega = (ab)^\omega$ for any two elements a and b.*

Proof. ($\Rightarrow$) Let S be $\mathscr{L}$-trivial and let a and b be elements of S. By aperiodicity, $a \cdot b(ab)^\omega = (ab)^\omega$, and $b \cdot (ab)^\omega = b(ab)^\omega$. In other words, $(ab)^\omega \mathrel{\mathscr{L}} b(ab)^\omega$. By $\mathscr{L}$-triviality, it follows that $(ab)^\omega = b(ab)^\omega$.

($\Leftarrow$) Let S be a finite semigroup such that $b(ab)^\omega = (ab)^\omega$ for all elements a and b, and let $a \mathrel{\mathscr{L}} b$. Then there exist elements s and t such that $sa = b$ and $tb = a$. Then $b = sa = stb = \ldots = (st)^\omega b = t(st)^\omega b = tb = a$. Because this holds for any $\mathscr{L}$-related elements, S is $\mathscr{L}$-trivial. ∎

A similar argument yields the following.

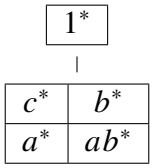


Figure 4.6: Egg-box of "begins with a and ends with b".

Lemma 4.6. *S is an $\mathscr{R}$-trivial semigroup iff $(ab)^\omega a = (ab)^\omega$ for any two elements a and b.*

Theorem 4.2. *If S is an $\mathscr{L}$- or $\mathscr{R}$-trivial semigroup, then all regular elements are idempotent.*

Proof. Let S be $\mathscr{L}$-trivial and let a and b be elements of S such that $aa = a$ and $a \mathrel{\mathscr{D}} b$. Then there exists some c such that $a \mathrel{\mathscr{L}} c$ and $c \mathrel{\mathscr{R}} b$. By $\mathscr{L}$-triviality, it follows that $a = c$ and thus $a \mathrel{\mathscr{R}} b$. Then by lemma 4.2, $ab = b$. Moreover, there exists some s such that $bs = a$. $a = a^\omega = (bs)^\omega = s(bs)^\omega = sa$. Then $bb = bab = bsab = aab = ab = b$ and b is idempotent. It follows that all regular elements are idempotent. A similar argument would show the same if S were $\mathscr{R}$-trivial. ∎

4.2 A First Expansion: $\mathbb{D}\mathbf{A}$

Figure 4.5 showed an aperiodic semigroup in which there exists a regular element that is not idempotent. But there are languages that do have this property but are neither $\mathscr{L}$- nor $\mathscr{R}$-trivial. Consider the language over $\Sigma = \{a, b, c\}$ of Figure 4.6 in which all words begin with a and end with b, shown only as its egg-box. It is neither $\mathscr{L}$- nor $\mathscr{R}$-trivial, but every element is idempotent. Thus it makes sense to include a new class, above $\mathscr{L}$ and $\mathscr{R}$ and below $\mathscr{H}$ as shown in Figure 4.7, containing all and only those aperiodic monoids whose regular elements are idempotent. The following theorem motivates the naming of this class.

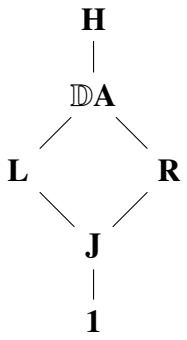


Figure 4.7: The basic hierarchy augmented with $\mathbb{D}\mathbf{A}$.

Theorem 4.3. *If S is a finite aperiodic semigroup then all of its regular elements are idempotent iff its regular $\mathscr{D}$-classes are semigroups.*

Proof. ($\Rightarrow$) Let S be a finite aperiodic semigroup in which all regular elements are idempotent, and let a and b be $\mathscr{D}$-related regular elements of S. Notice that a and b are idempotent. Then there exists some c such that $a \mathscr{L} c$ and $c \mathscr{R} b$. In this case we have that $cc = c$ and by lemma 4.2 $ac = a$ and $bc = c$. Then $a = ac = abc$. So $a = 1 \cdot ab \cdot c$ and $ab = 1 \cdot a \cdot b$. In other words, $a \mathscr{J} ab$ and because S is finite it follows that $a \mathscr{D} ab$. The regular $\mathscr{D}$-class is closed under the semigroup operation and is thus a subsemigroup.

($\Leftarrow$) Let S be a finite aperiodic semigroup whose regular $\mathscr{D}$-classes are subsemigroups of S, and let a be regular. Then $a \mathscr{J} a^2$ by the subsemigroup property, and thus by Corollary 4.1, $a \mathscr{H} a^2$. But S is aperiodic and thus $\mathscr{H}$-trivial, so $a = aa$. This holds for any regular element, so all regular elements are idempotent. ∎

This $\mathbb{D}\mathbf{A}$ class has been widely studied, with Tesson and Thérien (2002) even calling it a "jewel" in their detailed survey of the class for the many properties that characterize it alongside the many

fragments of logic that it describes. If $\text{FO}^k[<]$ denotes the subset of first-order logic with general precedence restricted to at most k different variables, then $\mathbb{D}\mathbf{A} = \text{FO}^2[<]$ (Thérien and Wilke, 1998). Pin and Weil (1997) show that this class also characterizes the class of languages definable in $\text{FO}[<]$ with at most one block of existential quantifiers and one block of universal quantifiers, in either order. And Etessami et al. (1997) show that $\mathbb{D}\mathbf{A}$ is also equivalent to a fragment of temporal logic, the fragment of unary temporal logic allowing only the "eventually in the future" ($\diamondplus$) and "eventually in the past" ($\diamondminus$) operators, $\text{TL}[\diamondplus, \diamondminus]$.

4.3 Piecewise Testable Languages and Subclasses

A language is piecewise testable (PT) iff there is some k for which it is defined by a Boolean combination of k-subsequences. Simon (1975) characterizes the piecewise testable languages as exactly those whose syntactic monoids are $\mathscr{J}$-trivial. That is, **J** corresponds to PT. Recall that a language and its complement share a structure. The languages piecewise testable in the strict sense (strictly piecewise) discussed by Rogers et al. (2010) then cannot be characterized by their semigroups alone. This point is discussed further in the conclusion of this chapter.

However, let us consider the subclass of **H** containing only those monoids which are commutative. Denote this class **Acom**, for aperiodic and commutative. By commutativity, it follows that any permutation of a word behaves the same as the word itself. Namely, one can sort its letters under some ordering imposed on Σ, and so long as the quantities are unchanged, the resulting word is identical in the eyes of the language. Further by aperiodicity, there exists for each symbol some threshold t_a such that $a^{t_a} = a^{t_a+1}$. Denote by t the least common multiple of these thresholds. Any span longer than length t of a single symbol can be compressed to a span of length t, either in the

original word or in its sorted counterpart. That is, if a language L has a syntactic monoid in **Acom** then two words w and v are $\overset{M}{\sim}$-related if they have the same multiset of symbols saturating at a count of t, that is, where counts higher than t are reduced to t itself. It is easy to argue that such a language must be in **J**, as the appearance of some symbol a at least k times is simply the k-long subsequence consisting entirely of a, and any language in **Acom** is a Boolean expression over these. The algebraic proof is just as trivial.

Theorem 4.4. **Acom** $\subsetneq$ **J**

Proof. Let S be a monoid in **Acom**, and let a and b be elements of S. Then there exist elements s, t, u, and v such that $sat = b$ and $ubv = a$. Then by commutativity, both $ast = b$ and $buv = a$, so $a \mathrel{\mathscr{R}} b$, and both $sta = b$ and $uvb = a$, so $a \mathrel{\mathscr{L}} b$. Then $a \mathrel{\mathscr{H}} b$ and by aperiodicity it follows that $a = b$. ∎

There are monoids in **J** that are not in **Acom**, such as that shown in Figure 4.8. Notice that $ab \neq ba$ and therefore the monoid is not commutative. Thus the inclusion is proper. An even smaller class arises when we fix the threshold to a count of one. Rather than require the general $a^{\omega+1} = a^{\omega}$, we fix $a^2 = a$. In other words, this is the restriction to monoids both idempotent and commutative, exactly the class of **semilattices**.

Theorem 4.5. *M is a semilattice iff $M \in$ **J** and $a^2 = a$ for all $a \in M$.*

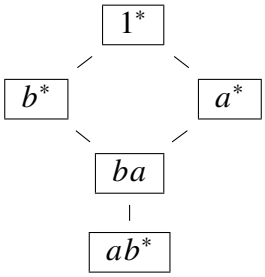


Figure 4.8: Egg-box for "contains $a \ldots b$".

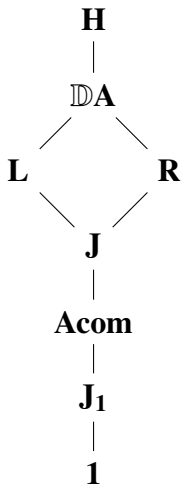


Figure 4.9: The hierarchy augmented with **Acom** and $\mathbf{J_1}$.

Proof. The forward direction is trivial, so we focus here on the converse. Let M be an idempotent $\mathscr{J}$-trivial monoid. Then $xy = xyxy$ and $yx = yxyx$, and therefore $xy \mathrel{\mathscr{J}} yx$. By $\mathscr{J}$-triviality, $xy = yx$. Now M is idempotent and commutative, a semilattice. ∎

Denote this smaller class $\mathbf{J_1}$, for "$\mathscr{J}$-trivial and idempotent". The language L in which all words contain two or more instances of a is in **Acom** but not in $\mathbf{J_1}$, as $aa \in L$ but $a \notin L$ and thus $a^2 \neq a$. Adding these new classes into our hierarchy results in Figure 4.9.

4.4 Equations, Varieties, and a Cloned Hierarchy

A semigroup (monoid) variety in the sense of Birkhoff is a collection **V** of semigroups (monoids) closed under the taking of subsemigroups (submonoids), under homomorphic images, and under products. A **pseudovariety** in the sense of Eilenberg and Schützenberger (1976) is like a variety, except that the semigroups (monoids) are assumed to be finite, and only finite products are considered. An important consequence of this is that any class of formal languages characterized by a (pseudo)variety is closed under (finitely many applications of) the Boolean operations. A class is a (pseudo)variety iff it is defined by equations in terms of variables that must apply for any instantiation of those variables, essentially equations with implicit universal quantifiers (Eilenberg and Schützenberger, 1976; Howie, 1995). As we are considering here only finite structures, the word "variety" will be used generally to mean both "variety" and "pseudovariety".

The classes we have discussed thus far are all defined not just by equations, but by a finite collection of equations. Each is a monoid variety. Lemmas 4.5 and 4.6 provide the equations for **L** and **R**, and one notes that $\mathbf{J} = \mathbf{L} \cap \mathbf{R}$, the collection of structures that are both $\mathscr{L}$- and $\mathscr{R}$-trivial. **H** is defined by aperiodicity. As discussed in the previous section, **Acom** is aperiodicity and commutativity, and $\mathbf{J_1}$ is the special case of **Acom** where $x^\omega = x$. And **1** is of course $x = y$. All that remains is to find a set of equations for $\mathbb{D}\mathbf{A}$, which either includes or implies that of **H**. An equation is provided below, and the resulting identities summarized in Table 4.1.

Theorem 4.6 (Tesson and Thérien, 2002)**.** *A monoid is in* $\mathbb{D}\mathbf{A}$ *iff whenever* $e^2 = e$ *and* $e = usv$ *it holds that* $ese = e$.

Table 4.1: Equations defining our basic hierarchy.

Class	Description	Equations
1	Trivial	$[\![x = y]\!]$
$\mathbf{J_1}$	Semilattices	$[\![x^2 = x,\ xy = yx]\!]$
Acom	Aperiodic and Commutative	$[\![x^{\omega+1} = x^\omega,\ xy = yx]\!]$
J	$\mathscr{J}$-trivial	$[\![y(xy)^\omega = (xy)^\omega = (xy)^\omega x]\!]$
L	$\mathscr{L}$-trivial	$[\![y(xy)^\omega = (xy)^\omega]\!]$
R	$\mathscr{R}$-trivial	$[\![(xy)^\omega x = (xy)^\omega]\!]$
$\mathbb{D}\mathbf{A}$	Regular elements idempotent	$[\![(xyz)^\omega y(xyz)^\omega = (xyz)^\omega]\!]$
H	Aperiodic	$[\![x^{\omega+1} = x^\omega]\!]$

Proof. ($\Rightarrow$) Suppose $M \in \mathbb{D}\mathbf{A}$ and let $e = usv$ such that $e^2 = e$. Then $e = (usv)^\omega$. This is $\mathscr{J}$-related to the idempotent $f = (svu)^\omega$; $sv \cdot e \cdot u = f$ and $u \cdot f \cdot sv = e$. Because $M \in \mathbb{D}\mathbf{A}$, $e \ \mathscr{J}\ ef$ and by lemma 4.3 it follows that $e \ \mathscr{R}\ ef$. More specifically, $ef = (usv)^\omega(svu)^\omega = (usv)^\omega s \cdot vu(svu)^\omega$. If $eft = e$, then $es \cdot vu(svu)^\omega t = e$. And of course $e \cdot s = es$, so $e \ \mathscr{R}\ es$. Further, by Theorem 4.3, es is idempotent. By lemma 4.2, $ese = e$.

($\Leftarrow$) Let M be a monoid such that for all elements s and all idempotents e such that $e = usv$ for some u and v, $ese = e$. Notice that by hypothesis, for any x it holds that $x^\omega x x^\omega = x^\omega$, so $x^\omega x x^\omega x = x^\omega x$, and thus $x^{\omega+1}$ is idempotent, but x^ω is the unique idempotent power of x, so $x^{\omega+1} = x^\omega$ and thus $M \in \mathbf{H}$. Further, let $e \ \mathscr{J}\ s$ such that $e^2 = e$. Then $ese = e$, and by right-multiplying by s we have $eses = es$. Because $es = e \cdot s \cdot 1$, by hypothesis it holds that $es \cdot s \cdot es = es$. By right-multiplying by e we have $essese = ese = e$. In other words, $e = e \cdot ss \cdot ese$, which means $MeM \subseteq MssM$. But $MeM = MsM$, and thus $MsM \subseteq MssM \subseteq MsM$, and $s \ \mathscr{J}\ ss$. By Corollary 4.1, $s^2 \ \mathscr{H}\ s$, and thus $s = s^2$. This argument holds for any regular element, so $M \in \mathbb{D}\mathbf{A}$. ∎

Corollary 4.2. *$M \in \mathbb{D}\mathbf{A}$ iff $(xyz)^\omega y(xyz)^\omega = (xyz)^\omega$ for all x, y, and z.*

Recall that the local subsemigroup of S generated by a is aSa, and that the local subsemigroup generated by an idempotent is a submonoid. If $\mathbf{V}$ is a monoid variety, we can construct a semigroup variety $\mathbb{L}\mathbf{V}$ where for every idempotent e, the submonoid eSe is in $\mathbf{V}$. The equations for the new semigroup variety are precisely those of the monoid variety wherein each variable is wrapped in α^ω for some unused variable α. This ensures that elements refer to elements of eSe, where $e = \alpha^\omega$. For example, a semigroup is locally a semilattice iff $(\alpha^\omega x \alpha^\omega)^2 = \alpha^\omega x \alpha^\omega$ and $\alpha^\omega x \alpha^\omega y \alpha^\omega = \alpha^\omega y \alpha^\omega x \alpha^\omega$. Because varieties are closed under taking of subsemigroups, $\mathbf{V} \subseteq \mathbb{L}\mathbf{V}$. Further, because every monoid contains 1, if a monoid is in $\mathbb{L}\mathbf{V}$ then it must be in $\mathbf{V}$. But in general there may be semigroups S that are not monoids in $\mathbb{L}\mathbf{V}$ where S^1 is not in $\mathbf{V}$.

For example consider the language of words containing an ab substring shown in Figure 4.5 on page 68. The identity 1 is the equivalence class of the empty string, ε, and this class is singleton; no other element occupies this class. That is, 1 is not in the syntactic semigroup $\Sigma^+/\overset{M}{\sim}$. The remaining idempotents are a, b, c, and ab (everything but ba). Note that xS is the row of the Cayley table containing x, and so xSx is each of these elements right-multiplied by x. Let us consider each idempotent's local subsemigroup in turn. The local subsemigroup generated by a is $\{a, ab\}$, because $\{a, ab, c\} \cdot a = \{a, ab\}$. Recall that a is necessarily the identity of this resulting submonoid, and $ab = abab$. The result is an idempotent, commutative monoid: a semilattice. The local subsemigroup generated by b is $\{b, ab\}$ and that of c is $\{c, ab\}$, also semilattices. Finally the local subsemigroup generated by ab is $\{ab\}$, trivial, and thus also a semilattice. All are semilattices, so this pattern is in $\mathbb{L}\mathbf{J_1}$ despite not being in $\mathbf{J_1}$ itself or even in $\mathbb{D}\mathbf{A}$.

Let us prove one simple theorem before constructing a fuller hierarchy. The remaining classes

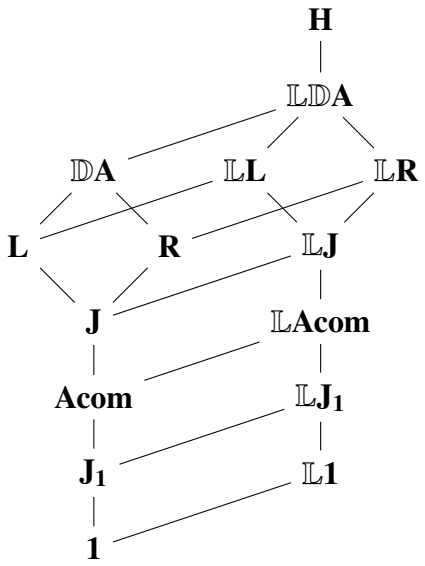


Figure 4.10: The hierarchy cloned to include local varieties.

will then be shown to be distinct by means of separating examples. The memberships and nonmemberships have been verified by `plebby`, software which will be discussed in Chapter 9.

Theorem 4.7. $\mathbb{L}\mathbf{H} = \mathbf{H}$.

Proof. Let S be a finite semigroup such that S^1 is not in $\mathbf{H}$. Then there exists some a and natural numbers m and n such that $a^m = a^{m+n}$ and $a^m \neq a^{m+1}$. Then a^{mn} is idempotent and the identity of the subgroup generated by a^{mn+1}, a nontrivial subgroup of $a^{mn}Sa^{mn}$. Thus this particular local subsemigroup is not $\mathscr{H}$-trivial, and $S \notin \mathbb{L}\mathbf{H}$. ∎

To show that these classes are separated, we will begin at the top and work downward. For each class $\mathbb{L}\mathbf{V}$, we explore one language which is $\mathbb{L}\mathbf{V}$ but not in any lower class. To show incomparability between the branches, we first find a monoid M in the lowest superclasses of $\mathbf{V}$ that is not in $\mathbb{L}\mathbf{V}$,

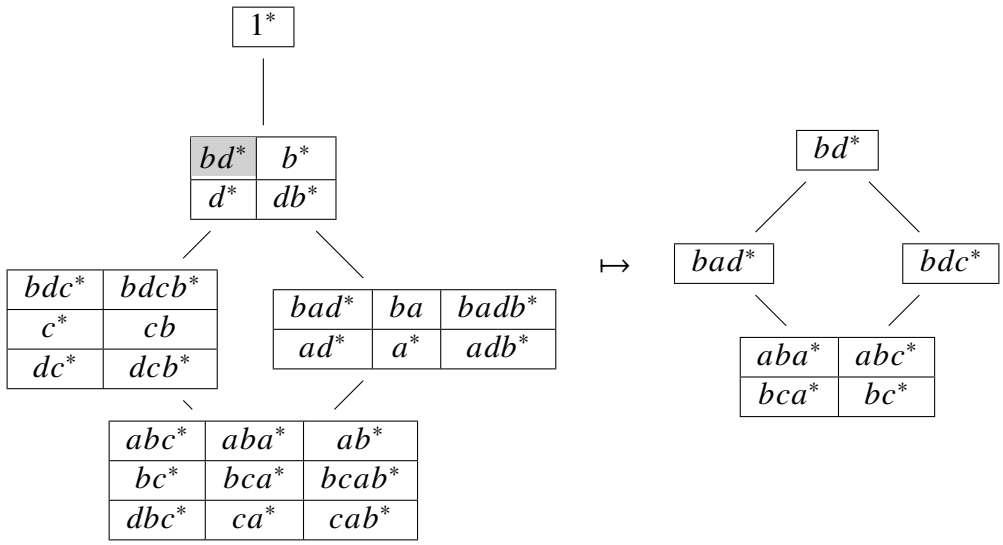


Figure 4.11: Egg-box for "contains a not preceding bc and contains c not following ab".

and then find a semigroup $S \in \mathbb{L}\mathbf{V}$ such that $S^1 \notin \mathbb{D}\mathbf{A}$, and therefore not in any of the lower classes. We end with a surprising containment.

4.4.1 Locally $\mathbb{D}\mathbf{A}$

Consider the language over $\Sigma = \{a, b, c, d\}$ of Figure 4.11 in which all words contain an a that is not followed by a bc substring anywhere later in the word and also contain a c not preceded by an ab substring anywhere earlier in the word. The idempotent bd is highlighted, and its local subsemigroup placed on the right. The full monoid S^1 is not in $\mathbb{D}\mathbf{A}$, because cb and ba are regular but not idempotent, and the local subsemigroup generated by bd is neither $\mathscr{L}$- nor $\mathscr{R}$-trivial, so $\mathbb{L}\mathbb{D}\mathbf{A}$ is a proper superclass of each of $\mathbb{D}\mathbf{A}$, $\mathbb{L}\mathbf{L}$, and $\mathbb{L}\mathbf{R}$.

Thérien and Wilke (1998, see also Krebs et al., 2020) show that this class characterizes precisely the set of languages definable in $\text{FO}^2[<, \lhd]$, the superset of $\text{FO}^2[<]$ where immediate-successor is available. Rather, they show that $\mathbb{D}\mathbf{A} * \mathbb{L}\mathbf{1}$ characterizes this class, and Almeida (1996) shows that

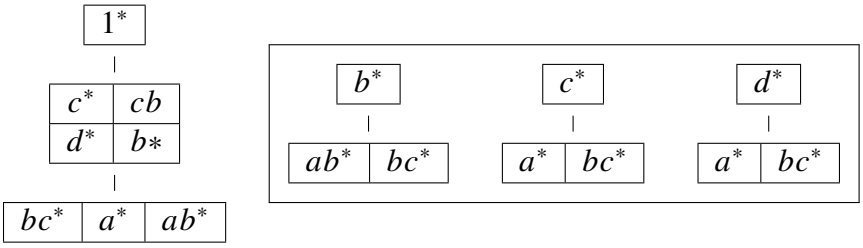


Figure 4.12: A language in $\mathbb{L}\mathbf{L}$ separating this class from $\mathbb{L}\mathbf{R}$ and $\mathbb{D}\mathbf{A}$.

$\mathbb{D}\mathbf{A} * \mathbb{L}\mathbf{1} = \mathbb{L}\mathbb{D}\mathbf{A}$. This class also characterizes precisely the fragment of unary temporal logic using $\oplus$ ("next") and $\ominus$ ("previously") in addition to $\diamondplus$ and $\diamondminus$, $\mathrm{TL}[\diamondplus, \diamondminus, \oplus, \ominus]$ (Etessami et al., 1997).

4.4.2 Locally $\mathscr{L}$- or $\mathscr{R}$-Trivial

The language separating $\mathbb{L}\mathbb{D}\mathbf{A}$ from $\mathbb{D}\mathbf{A}$ is a conjunction of two constraints: that all words contain an a nowhere followed by a bc, and that all words contain a c nowhere preceded by an ab. Let us consider only the first of these. That is, consider the language over $\Sigma = \{a, b, c, d\}$ where all words contain an a not followed by bc. This language is shown in Figure 4.12 alongside each of its nontrivial idempotent-generated local subsemigroups.

But $\mathbb{D}\mathbf{A}$ contains languages $\mathbb{L}\mathbf{L}$ does not, such as the language over $\Sigma = \{a, b, c, d\}$ shown in Figure 4.13 in which the first instance of $\{a, b\}$, if any, is a and the last, if any, is b. That is, when projected to the $\{a, b\}$ tier, a word is empty or it begins with a and ends with b. I have used c in place of 1 to mark the identity in order to indicate that $c \in [\varepsilon]$, that $1 \in \Sigma^+/\overset{M}{\sim}$. This monoid is not in $\mathbf{L}$, and because the identity is present, the entire monoid is an idempotent-generated local subsemigroup (generated by 1), so this is also not $\mathbb{L}\mathbf{L}$. But every element (and in particular every regular element) is idempotent, so it is in $\mathbb{D}\mathbf{A}$. This process of describing a pattern on some tier in order to ensure that the entire monoid is an idempotent-generated local subsemigroup is generally applicable.

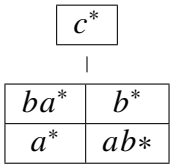


Figure 4.13: A language in $\mathbb{D}\mathbf{A}$ but not $\mathbb{L}\mathbf{L}$, proving incomparability

Theorem 4.8. *If* $\mathbf{V_1} \subsetneq \mathbf{V_2}$ *are monoid varieties, then* $\mathbb{L}\mathbf{V_1} \subsetneq \mathbb{L}\mathbf{V_2}$.

Proof. We proceed by construction. Let M be a monoid in $\mathbf{V_2} - \mathbf{V_1}$. Adjoin a new symbol σ which acts as the identity. Then $(\Sigma \cup \{\sigma\})^+/\overset{M}{\sim}$ is the same monoid, and the entire monoid is an idempotent-generated local subsemigroup. Then there is a local subsemigroup in $\mathbf{V_2} - \mathbf{V_1}$, and the resulting structure cannot be in $\mathbb{L}\mathbf{V_1}$. ■

The other component of our example $\mathbb{L}\mathbb{D}\mathbf{A}$ language, that all words contain a c nowhere preceded by an ab substring, can be verified to be in $\mathbb{L}\mathbf{R}$ but not $\mathbb{L}\mathbf{L}$ or $\mathbb{D}\mathbf{A}$. Thus we conclude that the topmost classes of the cloned hierarchy are distinct and incomparable as depicted.

4.4.3 Locally $\mathscr{J}$-Trivial

Using Theorem 4.8 of the previous section, one can construct a monoid in $\mathbf{L}$ or $\mathbf{R}$ such that the tier-based preprojection of the associated language is not $\mathbb{L}\mathbf{J}$. Moreover, the language of Figure 4.8 on page 73 is (locally) $\mathscr{J}$-trivial but not $\mathbb{L}\mathbf{Acom}$, as c is the identity and the entire monoid is an idempotent-generated subsemigroup which is not in $\mathbf{Acom}$.

The dot-depth hierarchy, introduced by Cohen and Brzozowski (1971) and later shown to be infinite when the alphabet consists of more than a single letter by Brzozowski and Knast (1978), offers a measure of how many levels of concatenation are required to express a star-free language. As noted

by Knast (1983), Simon (of piecewise-testable fame) conjectured that the languages of dot-depth at most one were characterized by $\mathbb{L}\mathbf{J}$. A language L is of dot-depth at most one iff there are some natural numbers j and k such that for all strings u and v in Σ^* it holds that if u and v share the same prefix and suffix of length $k-1$ and have the same j-tuples of k-factors, then either both $u \in L$ and $v \in L$ or both $u \notin L$ and $v \notin L$. A word w contains a j-tuple of k-factors (a $\langle j, k \rangle$-factor) $\langle x_1, \ldots, x_j \rangle$ iff for each i from 1 to j it holds that $w = u_i x_i v_i$ for some $u_i, v_i \in \Sigma^*$ where the elements of the sequence $\langle u_1, \ldots, u_j \rangle$ are of strictly increasing length.

This is reminiscent of the generalized subsequence languages discussed by Heinz (2010a), sometimes referred to as piecewise-locally testable (PLT) (Rogers and Lambert, 2019a). These languages are also characterized by containment of j-tuples of k-factors, except that containment is defined slightly differently. A word w contains the tuple $\langle x_1, \ldots, x_j \rangle$ in this sense iff $w = u_0 x_1 u_1 \ldots x_j u_j$ for some $u_0, \ldots, u_j \in \Sigma^*$. This latter class has been characterized logically as all and only those languages definable by Boolean combinations of factors that include both the successor and general precedence relations.

Knast (1983) shows that dot-depth one is precisely characterized by a variety $\mathbf{J} * \mathbb{L}\mathbf{1}$, where the $*$ refers to a specific sort of combination that will not be discussed further in this chapter. However, it is a theorem that $\mathbf{V} * \mathbb{L}\mathbf{1} \subseteq \mathbb{L}\mathbf{V}$ for any variety $\mathbf{V}$ (Straubing, 1985), and in this case the containment is proper (Knast, 1983). If the generalized subsequence languages and the languages of dot-depth at most one are distinct classes, then $\mathbb{L}\mathbf{J}$ properly contains both.

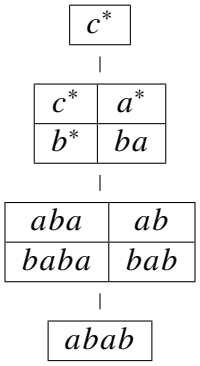


Figure 4.14: A language in $\mathbb{L}$**Acom** but not $\mathbb{L}\mathbf{J_1}$ or $\mathbb{D}\mathbf{A}$.

4.4.4 Locally Aperiodic and Commutative

The language over $\Sigma = \{a, b, c, d\}$ shown in Figure 4.14 is $\mathbb{L}$**Acom** but neither $\mathbb{L}\mathbf{J_1}$ nor even $\mathbb{D}\mathbf{A}$. This language consists of all and only those words that contain two distinct instances of the ab substring. This is not in $\mathbb{L}\mathbf{J_1}$ because the equation $(\alpha^\omega x \alpha^\omega)^2 = \alpha^\omega x \alpha^\omega$ is not satisfied for $\alpha = a$ and $x = b$; $a^\omega b a^\omega = aba$ but $(aba)^2 = abab$. Similarly the language of Figure 4.5 on page 68 requires only one instance of ab and is neither $\mathbb{L}\mathbf{1}$ nor $\mathbb{D}\mathbf{A}$. It is not in $\mathbb{L}\mathbf{1}$ because the equation $\alpha^\omega x \alpha^\omega = \alpha^\omega y \alpha^\omega$ is not satisfied for $\alpha = x = a$ and $y = b$; $a^\omega a a^\omega = a$ but $a^\omega b a^\omega = aba = ab$.

It is a theorem of Almeida (1989) that $\mathbb{L}$**Acom** is a characterization of the generalized locally testable class of Thomas (1982) in which words that have the same multiset of length-k substrings saturating at a count of t are treated identically. This nomenclature clashes with another meaning of the phrase "generalized locally testable" which will be discussed later, and so we use the more common "locally threshold testable" name for this class, introduced by Beauquier and Pin (1989). The special case where $t = 1$ is the locally testable class, characterized independently by Brzozowski and Simon (1973) and by McNaughton (1974) as $\mathbb{L}\mathbf{J_1}$.

4.4.5 Locally Trivial

Theorem 4.9. $\mathbb{L}\mathbf{1} \subsetneq \mathbb{D}\mathbf{A}$.

Proof. Recall the equational description of $\mathbb{D}\mathbf{A}$ — $(xyz)^\omega y(xyz)^\omega = (xyz)^\omega$. If a semigroup S is locally trivial, then $\alpha^\omega x \alpha^\omega = \alpha^\omega y \alpha^\omega$. Fix an idempotent $(stu)^\omega$ by setting $\alpha = stu$. Then consider the case where $x = t$ and $y = (stu)^\omega$. The equation for local triviality then says that $(stu)^\omega t(stu)^\omega = (stu)^\omega(stu)^\omega(stu)^\omega = (stu)^\omega$. Notice that this is identical to the equation for $\mathbb{D}\mathbf{A}$ under renaming of variables ($s \mapsto x$, $t \mapsto y$, $u \mapsto z$). ∎

In fact we can go one step further.

Theorem 4.10. *A semigroup is in* $\mathbb{L}\mathbf{1}$ *iff* $S^1 \in \mathbb{D}\mathbf{A}$ *and every idempotent lies in the same* $\mathscr{D}$*-class.*

Proof. ($\Rightarrow$) Let $S \in \mathbb{L}\mathbf{1}$. Then for all idempotents e and for all elements x, it holds that $exe = e$. In particular this holds for an x in the minimal ideal of S, an x such that for all y, $x \leqslant_{\mathscr{J}} y$. Then exe must also be in the minimal ideal, but $exe = e$, so e is itself in the minimal ideal. This means that all idempotents are in the same $\mathscr{J}$-class, and by finiteness they are in the same $\mathscr{D}$-class.

($\Leftarrow$) Now suppose instead that S is a finite member of $\mathbb{D}\mathbf{A}$ whose idempotents all lie in the same $\mathscr{D}$-class. Let e be idempotent and x be an arbitrary element. Note that for any a in the minimal ideal, a^ω remains in the minimal ideal, so by hypothesis e is in this minimal ideal. This implies that $e \leqslant_{\mathscr{J}} x$ and thus $exe = e$ by the $\mathbb{D}\mathbf{A}$ equation. As this holds for all x, $S \in \mathbb{L}\mathbf{1}$. ∎

Thus we will certainly not find an element of $\mathbb{L}\mathbf{1}$ that is not in $\mathbb{D}\mathbf{A}$. But we can find a member of $\mathbb{L}\mathbf{1}$ that is in neither $\mathbf{L}$ nor $\mathbf{R}$. Indeed, we have done so already: the language in which words begin with a and end with b shown in Figure 4.6 on page 69 is locally trivial but neither $\mathscr{L}$- nor $\mathscr{R}$-trivial. However the language over $\Sigma = \{a, b, c\}$ in which all words contain both b and c is verifiably a semilattice but not in $\mathbb{L}\mathbf{1}$.

The variety $\mathbb{L}\mathbf{1}$ characterizes what are known as the generalized definite languages, defined by Boolean combinations of permitted prefixes and suffixes (Zalcstein, 1972; Brzozowski and Simon, 1973; McNaughton, 1974). Recall that the equation for $\mathbb{L}\mathbf{1}$ is $\alpha^\omega x \alpha^\omega = \alpha^\omega y \alpha^\omega$ for all x and y. Specifically this holds for $y = \alpha^\omega$, so $\mathbb{L}\mathbf{1}$ is more simply $\alpha^\omega x \alpha^\omega = \alpha^\omega$. What if this were just a one-sided multiplication? Define $\mathbf{D}$ by $x\alpha^\omega = \alpha^\omega$ (i.e. $Se = e$) and $\mathbf{K}$ by $\alpha^\omega x = \alpha^\omega$ (i.e. $eS = e$), and define $\mathbf{F}$ as $\mathbf{D} \cap \mathbf{K}$. Zalcstein (1972) shows that $\mathbf{D}$ corresponds precisely to the definite languages, those characterized by Boolean combinations of permitted suffixes and that $\mathbf{K}$ corresponds to the reverse definite languages, those characterized by Boolean combinations of permitted prefixes. This makes sense, as the idempotents α^ω are the sufficiently-long suffixes or prefixes of allowed words, alongside those disallowed words long enough to not be extendable to a valid word. The definite languages have been explored in depth (Perles et al., 1963; Ginzburg, 1966), and this class will be particularly relevant in the next chapter.

Theorem 4.11. *A language is in* $\mathbf{F}$ *iff it or its complement is finite.*

Proof. We shall show that for any semigroup in $\mathbf{F}$ there is exactly one idempotent, 0. The only cycles in the corresponding Cayley graph will then be self-loops on 0. Translating to automata, if

this is a rejecting state, then the canonical trim acceptor is acyclic, thus recognizing a finite language. If it is accepting, then it is rejecting in the complement, and the complement is finite.

Let e and f be idempotents. Then the equations for **D** give us that $e = ef$. The equations for **K** give us that $f = ef$, and thus $f = ef = e$. Thus there is only one idempotent. Moreover, $ex = e = xe$ for all x, again by the **D** and **K** equations, so e is a zero. ■

The following theorems follow the same structure as Theorem 4.10.

Theorem 4.12. *A semigroup S is in* **D** *iff* $S^1 \in$ **L** *and every idempotent lies in the same $\mathscr{D}$-class.*

Theorem 4.13. *A semigroup S is in* **K** *iff* $S^1 \in$ **R** *and every idempotent lies in the same $\mathscr{D}$-class.*

Theorem 4.14. *A semigroup S is in* **F** *iff* $S^1 \in$ **J** *and every idempotent lies in the same $\mathscr{D}$-class.*

This suggests that the diamond from **J** through $\mathbb{D}$**A** is a generalization of some sort of the diamond from **F** through $\mathbb{L}$**1**. And indeed as discussed by (Brzozowski and Fich, 1984), this is exactly the case. Define M_e to be the set $\{g : e \leqslant_{\mathscr{J}} g\}$, that is, the set of elements g such that $e = sgt$ for some s and t. We will consider the move from **D** to **L**, and the others follow the same pattern. The equation for **D** is $x\alpha^\omega = \alpha^\omega$. Here, α^ω is an arbitrary idempotent; to restrict to those idempotents e for which $x \in M_e$, use sxt instead of α, as all such idempotents are of the form $(sxt)^\omega$. If we would like to say $M_e e = \{e\}$, our equation becomes $(sxt)^\omega x = (sxt)^\omega$.

Theorem 4.15. *A monoid M satisfies $M_e e = \{e\}$ for all idempotents e iff $M \in$* **L**.

Proof. ($\Rightarrow$) Let M satisfy the equation $x(sxt)^\omega = (sxt)^\omega$. Instantiate $s \mapsto a$, $t \mapsto 1$, and $x \mapsto b$. This yields $b(ab)^\omega = (ab)^\omega$, exactly the equation for **L** under renaming of variables.

($\Leftarrow$) Now instead let M satisfy the equation $b(ab)^\omega = (ab)^\omega$. Consider the instantiation $a \mapsto sg$ and $b \mapsto t$. Then $t(sgt)^\omega = (sgt)^\omega$. Now instead consider the instantiation $a \mapsto s$ and $b \mapsto gt$. Then $(sgt)^\omega = gt(sgt)^\omega = g(sgt)^\omega$, precisely the equation for $M_e e = \{e\}$ under renaming of variables. ■

Theorem 4.16. *A monoid M satisfies $eM_e = \{e\}$ for all idempotents e iff $M \in$* **R**.

A restatement of Corollary 4.2 is the following.

Theorem 4.17. *A monoid M satisfies $eM_e e = \{e\}$ for all idempotents e iff $M \in \mathbb{D}$***A**.

Corollary 4.3. $\mathbb{D}$**A** *is precisely the class of $\mathscr{G}$-trivial monoids, defined by Brzozowski and Fich (1984) as those monoids satisfying $eM_e e = \{e\}$.*

Figure 4.15 depicts the cloned hierarchy with the classes below $\mathbb{L}$**1** included as well as these connections across the branches formed by using M_e.

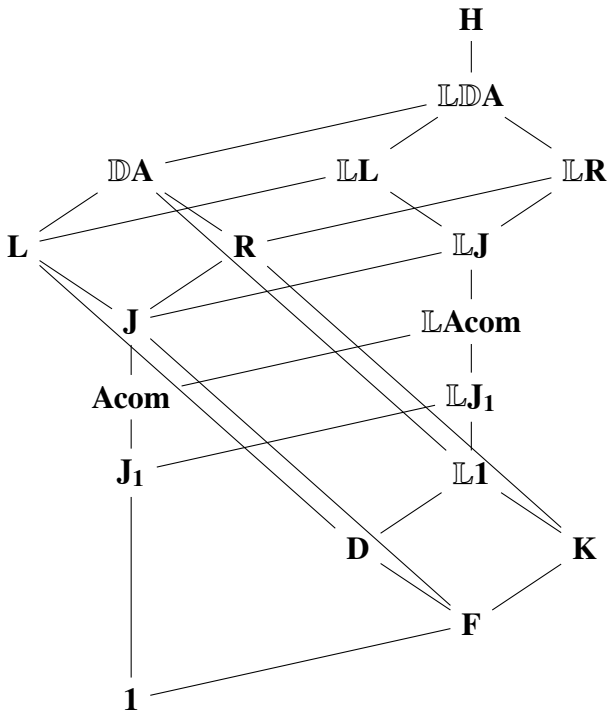


Figure 4.15: The hierarchy including sub-$\mathbb{L}\mathbf{1}$ classes and M_e-based connections.

4.5 Tier-Based Classes

Consider the monoid of Figure 4.13 on page 80. This would be in $\mathbb{L}\mathbf{1}$ if not for the fact that c is the identity, a nonsalient symbol. Recall from Chapter 3 the notion of the projected subsemigroup of a language, and define for each semigroup variety $\mathbf{V}$ a new class $[\![\mathbf{V}]\!]_{\mathrm{T}}$ containing all and only those languages whose projected subsemigroups are in $\mathbf{V}$. In particular, this precisely characterizes the tier-based locally testable and tier-based locally threshold testable classes of Chapter 3 as $[\![\mathbb{L}\mathbf{J_1}]\!]_{\mathrm{T}}$ and $[\![\mathbb{L}\mathbf{Acom}]\!]_{\mathrm{T}}$, respectively.

If $\mathbf{V}$ is a variety, then $[\![\mathbf{V}]\!]_{\mathrm{T}}$ is always a superclass of $\mathbf{V}$, though the containment is not necessarily proper. If $\mathbf{V}$ is a monoid variety, then $[\![\mathbf{V}]\!]_{\mathrm{T}}$ is the same as $\mathbf{V}$, as the identity must be readjoined when checking for membership. Thus for each of the eight classes making up our basic hierarchy, there is no distinct tier-based class. However if $\mathbf{V}$ is a semigroup variety, then in general the tier-based

variant may be distinct, as we have seen with $\mathbb{L}\mathbf{1}$ and $[\![\mathbb{L}\mathbf{1}]\!]_{\mathrm{T}}$. Recall that $\mathbb{L}\mathbf{V}$ is defined to contain all and only those semigroups S such that $eSe \in V$. Denote by $\mathbf{M_e V}$ the variety of monoids where $eM_e e \in V$. Then $\mathbb{L}\mathbf{V}$ is a subclass of $\mathbf{M_e V}$, and the following holds.

Theorem 4.18. *If* $S \in \mathbb{L}\mathbf{V}$ *then* $S^1 \in \mathbf{M_e V}$.

Proof. Let S be a semigroup in $\mathbb{L}\mathbf{V}$, that is, where $eSe \in V$ for all idempotents e of S. If $1 \in S$ then S is a monoid and $S^1 = S$, and thus by containment $S^1 \in M_e V$. On the other hand if $1 \notin S$, then $exe \neq 1$ for any e or x in S, so $eM_e e \in V$ for all nonidentity e. But if $exe \neq 1$ for all e and x, it follows that $M_1 = \{1\}$. Then $1M_1 1 = \{1\}$ and is necessarily in $\mathbf{V}$. In sum, $eM_e e \in \mathbf{V}$ for all idempotents e, including 1, and thus $S^1 \in \mathbf{M_e V}$. ∎

Corollary 4.4. $\mathbf{M_e V}$ *contains the Boolean closure of* $[\![\mathbb{L}\mathbf{V}]\!]_{\mathrm{T}}$.

This containment is, in general, proper. Consider the language L of Figure 4.3 on page 66, wherein all words contain an a nowhere followed by a $b \ldots c$ subsequence, and suppose for simplicity that the alphabet is $\Sigma = \{a, b, c\}$. This is $\mathscr{L}$-trivial and therefore $M_e e = e$. One can construct two words that have the same length-k suffixes on every tier where one is in the language and the other is not. Consider the strings $a^k c^k b^k \in L$ and $a^k b c^k b^k \notin L$. On any tier containing b, the k-suffix is b^k. On any tier containing c but not b, it is c^k. On the tier containing only a, it is a^k. And on the empty tier, it is of course ε. Of the eight possible tiers, four contain b ($\{b\}$, $\{a, b\}$, $\{b, c\}$, and $\{a, b, c\}$), two contain c but not b ($\{c\}$ and $\{a, c\}$), one is a alone, and the other is empty. We have covered all cases, and the two words have the same k-suffixes on both tiers. If the language were multiple-tier-based definite, the words would have to be treated the same, but they are not.

For any monoid variety $\mathbf{V}$, there is a local variety $\mathbb{L}\mathbf{V}$. This local variety extends to a tier-based class $[\![\mathbb{L}\mathbf{V}]\!]_{\mathrm{T}}$. Then the multiple-tier-based $\mathbb{L}\mathbf{V}$ class is the Boolean closure of $[\![\mathbb{L}\mathbf{V}]\!]_{\mathrm{T}}$ and is contained in $\mathbf{M_e V}$. Two of the higher classes have been discussed in prior literature. There is the generalized locally testable class of Brzozowski and Fich (1984), $\mathbf{M_e J_1}$, generalizing both the locally testable class and $\mathbb{D}\mathbf{A}$, and there is also a fragment of first order logic characterized by Krebs et al. (2020), $\mathbf{M_e}\mathbb{D}\mathbf{A}$ corresponding to $\textsc{fo}^2[<, \mathrm{bet}]$. This is the fragment of first-order logic with general precedence restricted to two variables with an additional binary predicate per alphabetic symbol where $a(x, y)$ means "an a appears in the range of positions between x and y". The language $BB_2 = (a(ab)^*b)^+$ discussed by Krebs et al. (2020) is in $\mathbf{H}$ but not $\mathbf{M_e}\mathbb{D}\mathbf{A}$, demonstrating that the latter does not possess the full power of star-free. And U_2 from the same work, defined as $(\Sigma^* - (\Sigma^* ac^* a\Sigma^*)) \cup (\Sigma^* - (\Sigma^* bc^* b\Sigma^*))ac^* a\Sigma^*$, is in $\mathbf{M_e}\mathbb{D}\mathbf{A}$ but not $\mathbf{M_e J_1}$, separating them. A simplified variant of this language, U_2' continues to separate the classes: $U_2' = (\Sigma^* - (\Sigma^* bc^* b\Sigma^* ac^* a\Sigma^*))$ U_2' can be defined more simply as a $[\![\mathbb{L}\mathbf{J}]\!]_{\mathrm{T}}$ language, forbidding on the $\{a, b\}$ tier the occurrence of a bb substring eventually followed by an aa substring. Then $U_2' \in \mathbf{M_e J}$ but not $\mathbf{M_e J_1}$. The equivalence of these two definitions of U_2' has been verified with `plebby`, as has the fact that each language under discussion in this chapter is a separator of classes as claimed. This software will be discussed further in Chapter 9.

4.6 Conclusions

We have constructed from first principles a hierarchy of classes of formal languages. This hierarchy is split into three branches, with the base branch being a collection of monoid varieties. Recall their equational descriptions from Table 4.1. This branch is cloned into a branch of semigroup varieties wherein $S \in \mathbb{L}\mathbf{V}$ iff $eSe \in \mathbf{V}$ for each idempotent e of S. These two branches subsume the

propositional and higher levels of the traditional piecewise-local subregular hierarchy explored in previous chapters. The relativized extensions of classes introduced in Chapter 3 are not closed under the Boolean operations and thus are not varieties. But these classes can be defined algebraically by constructing a class $[\![\mathbb{L}\mathbf{V}]\!]_{\mathrm{T}}$ containing all and only those languages whose projected subsemigroups are $\mathbb{L}\mathbf{V}$.

This full hierarchy also incorporates classes that characterize various fragments of formal logic that were not previously handled in a unified way. It is well known that $\mathbb{D}\mathbf{A}$ corresponds to fragments of various types of formal logic, and $\mathbb{L}\mathbb{D}\mathbf{A}$ and $\mathbf{M_e}\mathbb{D}\mathbf{A}$ are more powerful fragments still below full first-order. Figure 4.16 shows the classes discussed in this chapter alongside their common names, if any. Notably missing from this diagram are the strictly piecewise and (tier-based) strictly local classes. These classes are not closed under complement, which means that they are not characterizable from their syntactic monoids without additional information. However, if acceptance parity is maintained then work of Fu et al. (2011) characterizes strictly piecewise and De Luca and Restivo (1980) characterizes strictly local in a way that can be generalized to account for tier-based strictly local. The ordered semigroups discussed by Pin (1997) are useful in this regard. Exploring these is a task for future work, especially (in light of the next chapter) how such a representation would interact with characterization of transition semigroups of transducers rather than those of acceptors.

This unified hierarchy is still far from complete. Adding constraints to any class will produce a subclass; if the constraints are in the form of equations then a variety can be strengthened into a subvariety. Conversely removing or weaking constraints yields a superclass. Table 4.2 offers a quick

Table 4.2: A summary of classes and their characterizations.

$\mathbf{V}$	Name	$\mathbf{LV}$	$\mathbf{M_eV}$
$\mathbf{1}$	1	GD	$\text{FO}^2[<]$
$\mathbf{J_I}$	1-LT	LT	GLT
Acom	$\langle 1, t\rangle$-LTT	LTT	GLTT
$\mathbf{J}$	PT	⋯	
$\mathbb{D}\mathbf{A}$	$\text{FO}^2[<]$	$\text{FO}^2[<, \lhd]$	$\text{FO}^2[<, \text{bet}]$

reference to the algebraic properties that characterize many of these classes. Omitted are $\mathbf{L}$ and $\mathbf{R}$, as well as the classes below $\mathbb{L}\mathbf{1}$. Each monoid variety $\mathbf{V}$ defines a class of languages, and two derived classes are formed: languages whose semigroups are locally in $\mathbf{V}$ (which admit a tier-based variant), and languages whose generated submonoids eM_ee are in $\mathbf{V}$ (which do not have a distinct tier-based variant).

Note that in Table 4.2 inclusion holds upward and leftward, and the rightmost column begins where the leftmost ends. Figure 4.16 will continue to spiral upward as the inclusions will continue to hold. It would be interesting to explore whether there is a fixed point, whether the spiral ends. If so, where would it be?

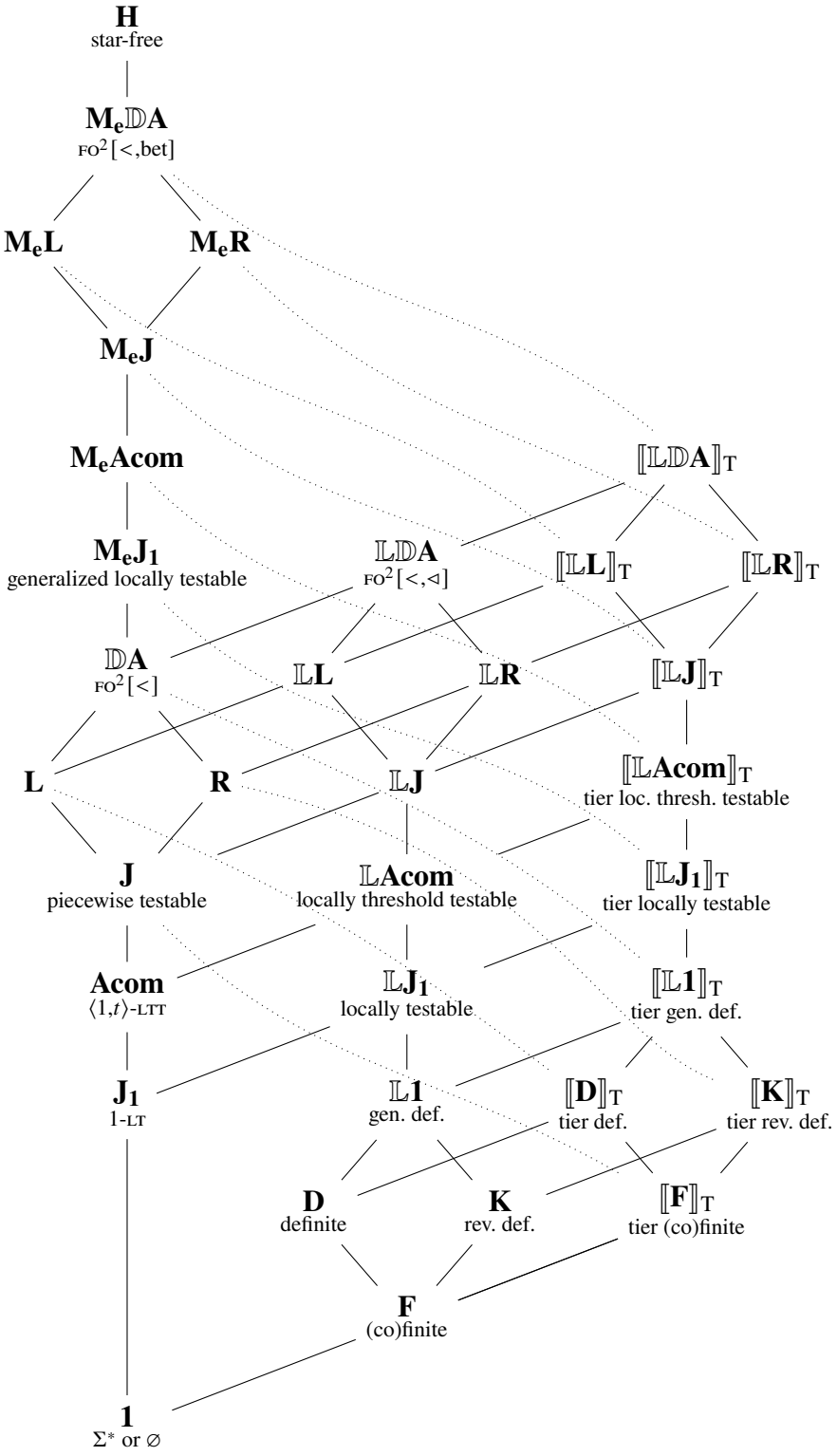


Figure 4.16: The hierarchy including sub-$\mathbb{L}\mathbf{1}$ classes and M_e-based connections.

Chapter 5: Classifying Functions

The previous chapter describes a broad hierarchy of subregular classes of formal languages. Further, it provides specific ways in which any given class may be generalized to maintain desirable closure properties, while noting that in general any addition of constraints results in a subclass while any relaxation yields a superclass. The methods described in that chapter generalize quite easily to string-to-string functions. In this chapter, this generalization is explored, first in the context of one-directional deterministic finite-state transducers, then further for the more powerful class of not-necessarily deterministic two-way machines.

One reason for factoring patterns is to obtain some sort of compositional understand of their complexity. If a formal language is built as a conjunction of two or more constraints, then the complexity of the pattern as a whole is no higher than a class that contains the intersection closures of each individual constraint's class. Moreover if each individual constraint is learnable, each acceptor could be stored separately and a final judgment could be taken by deciding whether all of them accept a given input. As will be shown later in this chapter, care must be taken when trying to deal with functions in the same way. Most of the classes relevant to linguists are closed under intersection, because this operation is a direct product. Function composition is not such an operation, and therefore does not preserve class membership.

This chapter begins with a review of algebraic concepts and a discussion of what are commonly known as subsequential string-to-string functions. These functions cover a large part of the set of phonologically relevant processes, and offer a direct generalization of the methods used for

classifying string acceptors. In order to explore the range of attested patterns, several processes that appear in natural language are demonstrated and classified. Not all relevant processes are not in this class, however, and another method is required to appropriately categorize others. A more general method follows, allowing one to classify any function definable by a two-way finite-state transducer. This generality comes at a price: lacking a canonical form, a machine may witness that a process is of at most a given complexity, but there is no clear way to tell that this is a minimum.

5.1 Structures and Machines

A **semigroup** is a set S closed under some binary operation $\cdot$ (often denoted by adjacency) which is associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$. Given some finite alphabet Σ, the set of all nonempty sequences made up of those letters forms a semigroup with the concatenation operation. This is called the **free semigroup** generated by Σ. A **monoid** is a semigroup in which there exists some element e such that for all x, $e \cdot x = x \cdot e = x$. Typically this identity element is represented by 1. The free semigroup generated by Σ can be adapted to the **free monoid** generated by Σ by including the empty sequence (denoted by ε), the identity for concatenation. The free semigroup and free monoid generated by Σ are often denoted Σ^+ and Σ^*, respectively.

A formal language L over Σ is some subset of this free monoid. Two useful equivalence relations can be defined based on L. The **Nerode equivalence** relation is defined such that $a \overset{N}{\sim} b$ iff for all $v \in \Sigma^*$ it holds that $av \in L$ in all and only those cases where $bv \in L$ (Nerode, 1958). This is often referred to as the Myhill-Nerode equivalence relation, as the well-known Myhill-Nerode theorem states that a language is regular iff its set of equivalence classes is finite. However, Myhill used a finer partition to achieve the same result: the **Myhill equivalence** relation is defined such that

$a \overset{M}{\sim} b$ iff for all $u \in \Sigma^*$, $ua \overset{N}{\sim} ub$. In other words, for all $u, v \in \Sigma^*$, $uav \in L$ iff $ubv \in L$ (Rabin and Scott, 1959). Being a coarser partition, $\overset{N}{\sim}$ can never define more classes than $\overset{M}{\sim}$, and the number of classes defined by $\overset{M}{\sim}$ is in the worst case exponential in that defined by $\overset{N}{\sim}$ (Holzer and König, 2004), so finiteness in one translates to the other.

5.1.1 Illustrating Nerode and Myhill Relations

Consider the example language over $\{a, b, c\}$ consisting of all and only those words that do not contain an ab substring. Consider which classes must exist. $[\![ab]\!]$ is the set of words containing an ab substring. These are rejected, and no suffix can save them. So if $x, y \in [\![ab]\!]$, for all v it holds that $xv \notin L$ and $yv \notin L$. All of these words are related, and distinct from any accepted words. But the accepted words partition into two classes: words that end in a ($[\![a]\!]$) and others ($[\![\varepsilon]\!]$). The former are rejected after adding a b suffix, while the latter remain accepted after adding a b suffix. No suffix distinguishes words within these classes, so the three are all that exist. These three classes can define a minimal deterministic finite-state acceptor for L (Nerode, 1958), as will be discussed shortly.

The equivalence classes under the Myhill relation then necessarily number at least three. But some of the classes split. The two strings a and ba are distinguished by an a prefix, and this generalizes. The class of accepted words ending in a must be split in two: words ending in a that begin with b ($[\![ba]\!]$) and other words ending in a ($[\![a]\!]$). The class of accepted words not ending in a splits in three: words beginning in b ($[\![b]\!]$), nonempty words not beginning in b ($[\![c]\!]$), and the empty word ($[\![\varepsilon]\!]$). The first of these is distinguished from the other two by an a prefix, while the last is distinguished from $[\![c]\!]$ by the $a\_\_b$ circumfix. The six equivalence classes are $[\![\varepsilon]\!]$, $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, $[\![ba]\!]$, and $[\![ab]\!]$.

The equivalence classes under $\overset{N}{\sim}$ are not fully compatible with concatenation. If $u \overset{N}{\sim} u'$, then $uv \overset{N}{\sim} u'v$, but it may be that $v \overset{N}{\sim} v'$ while $uv \overset{N}{\not\sim} uv'$ (it is only a right congruence). In the current example, $b \overset{N}{\sim} c$ but $ab \overset{N}{\not\sim} ac$. Unlike $\overset{N}{\sim}$, the equivalence classes under $\overset{M}{\sim}$ are fully compatible with concatenation (it is a congruence): if $u \overset{M}{\sim} u'$ and $v \overset{M}{\sim} v'$, it follows that $uv \overset{M}{\sim} u'v'$. That means these equivalence classes form the elements of a submonoid of Σ^*. The quotient monoid $\Sigma^*/\overset{M}{\sim}$ (these equivalence classes under concatenation) is called the **syntactic monoid** of L and denoted $M(L)$. If L is a regular language, then $M(L)$ is the smallest monoid which can be used as a deterministic finite-state acceptor that accepts L (Rabin and Scott, 1959). The **syntactic semigroup** is $\Sigma^+/\overset{M}{\sim}$.

A string language is rational if and only if there are finitely many equivalence classes under the Nerode relation (equivalently, under the Myhill relation) Rabin and Scott (1959). A **variety** of finite monoids is a class of monoids closed under the taking of submonoids, quotients and finite direct product Pin (1997). Varieties are also closed under Boolean operations. Eilenberg's theorem states that varieties of finite monoids uniquely pick out certain subclasses of rational languages Eilenberg and Schützenberger (1976). As we explain later these varieties can also pick out certain subclasses of rational functions.

5.1.2 String Acceptors

A deterministic finite-state acceptor is described by a five-tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$ where Σ is a finite alphabet, Q a finite set of states, $\delta : \Sigma \times Q \to Q$ a transition function, q_0 an initial state, and F a set of accepting states. A word is read one symbol at a time. If computation is in state q, the next symbol is σ, and $\delta(\sigma, q) = r$, then after reading that σ computation will be in state r. Given the equivalence classes under $\overset{N}{\sim}$ or $\overset{M}{\sim}$, we can construct such an acceptor. Σ is the alphabet over which

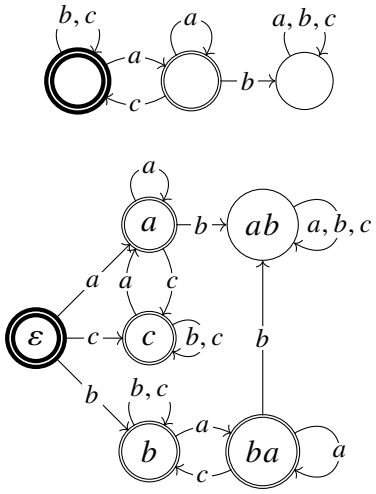


Figure 5.1: The acceptors induced by the Nerode (above) and Myhill (below) equivalence relations for a string language forbidding ab substrings. In the latter, states are labeled by a representative of their equivalence class, and doubly circled states are accepting. The state designated by the extra thick border is the initial state.

words were generated, Q is the set of equivalence classes, $\delta(\sigma, q)$ is the equivalence class of $q\sigma$, q_0 is whichever class contains the empty sequence, and F is the set of equivalence classes which contain accepted words. Hopcroft and Ullman (1979) discuss a dynamic programming algorithm to reduce an arbitrary deterministic acceptor to that associated with its Nerode equivalence relation. Another simple method, which will be described later, transforms that canonical form into the acceptor generated by the Myhill equivalence relation.

Figure 5.1 shows the acceptors induced by $\overset{N}{\sim}$ and $\overset{M}{\sim}$ for the example language over $\{a, b, c\}$ in which no word contains an ab substring. The acceptor induced by the syntactic monoid of L is a right Cayley graph of that monoid (see Zelinka, 1981) augmented with information about whether the elements represent accepted or rejected words.

5.1.3 String-to-String Transducers

Oncina et al. (1993) discuss one method of generalizing these acceptors into functions. A **sequential**

transducer is a five-tuple $\langle \Sigma, \Delta, Q, \delta, q_0 \rangle$, where Σ is the alphabet of input strings, Δ that of output strings, Q a finite set of states, $\delta : \Sigma \times Q \rightarrow \Delta^* \times Q$ a transition function, and q_0 an initial state. This behaves like an acceptor, where all strings are accepted (in the domain of the function) and every edge traversed appends to an accumulated output. Sequential functions are total. A **subsequential transducer** generalizes this notion by associating outputs with states (Oncina et al., 1993); if an input word ends in state q, the output word receives the suffix associated with state q. The function σ mapping states to suffixes is added as a sixth element: $\langle \Sigma, \Delta, Q, \delta, q_0, \sigma \rangle$. The names and order of these components here are not the same as those used in the original work, but seem to have become commonly used in later work. Adding another element, a prefix applied to all output strings, changes nothing because it could simply be added to each edge out of q_0 and to the output associated with that state. So in this work, this universal prefix π will be assumed: $\langle \Sigma, \Delta, Q, \delta, q_0, \pi, \sigma \rangle$. This change leaves most definitions unaffected.

Bruyère and Reutenauer (1999) argue that the subsequential notion is more deserving of the status as the basic object, and refer to such functions as simply sequential, a practice followed by Lombardy and Sakarovitch (2006), among others. A subsequential machine is equivalent to a sequential machine over a larger alphabet that includes explicit boundary symbols, and a well-formed version of the latter can be rewritten as the former. Given this bijection, the remainder of this work will follow this recent notational trend.

The property of being sequential depends on the direction in which the input is read. An iterative regressive harmony pattern cannot be described by a left-to-right sequential function because there is an unbounded delay between seeing a harmonizing symbol and finding the trigger that determines

its surface form (Heinz and Lai, 2013; Mohri, 1997). However, this process can be expressed as a right-to-left sequential function. One might think of this as reversing the output obtained by applying some left-to-right transducer to the reversal of the input. Alternatively, one could say the machine reads the string from right to left and concatenates in the same direction on the output. A left-to-right class will be denoted here in the form →SQ and a right-to-left class in the form ←SQ, where the arrow indicates the direction and SQ refers to the sequential property.

A transducer is **onward** if its output is produced as early as possible: For all states p, $\text{lcp}(\{y \in \Delta^* : \delta(a, p) = \langle y, q\rangle\} \cup \{\sigma(p)\}) = \varepsilon$. 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:

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

Two strings are related iff they share the same set of tails. We will denote this relation by $\overset{N}{\sim}$ to emphasize its connection to the Nerode equivalence for string sets. A transducer in canonical form is onward and has one state per equivalence class under $\overset{N}{\sim}$. Naturally there is a two-sided extension of this that generalizes the Myhill equivalence relation. The **contexts** of x in f are as follows:

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

When restricted to $w = \varepsilon$, the resulting set is essentially equivalent to the tails, so the $\overset{M}{\sim}$ relation derived from C forms, as with string sets, a refinement of the partition induced by $\overset{N}{\sim}$.

5.1.4 Constructing Monoids from Canonical Machines

The canonical form of a machine has states corresponding to the $\overset{N}{\sim}$ relation. The $\overset{M}{\sim}$ relation gives a notion of the influence of prefixes. So, to construct a machine over $\overset{M}{\sim}$ from a canonical machine, i.e. to construct a right Cayley graph of the monoid associated with the structure of the machine, we look to see where each input symbol takes each of the states. In other words, what function over the states does each symbol act as? This is the transition congruence of the machine (Filiot et al., 2016). McNaughton and Papert (1971) use this same construction.

Consider the automata of Figure 5.1. Associate a number which each state of the automaton induced by $\overset{N}{\sim}$: $[\![\varepsilon]\!]$ is 1, $[\![a]\!]$ is 2, and $[\![ab]\!]$ is 3. The numbering is arbitrary. Denote by $\langle x, y, z\rangle$ the function which maps 1 to x, 2 to y, and 3 to z. The identity function always exists and corresponds to ε: $\langle 1, 2, 3\rangle$. From there, a, b, and c act as $\langle 2, 2, 3\rangle$, $\langle 1, 3, 3\rangle$ and $\langle 1, 1, 3\rangle$, respectively. These mappings are enough to complete the structure. Consider ab: this first applies the a mapping, then to that result applies the b mapping, so $\langle 1, 2, 3\rangle$ maps first to $\langle 2, 2, 3\rangle$ by a then to $\langle 3, 3, 3\rangle$ by b (because both 2 and 3 map to 3). By the same process, we find that $aa = ca = a$, $ac = cb = cc = c$, $bb = bc = b$ and ba is a new state $\langle 2, 3, 3\rangle$. Continuing this process with the $ab = \langle 3, 3, 3\rangle$ and $ba = \langle 2, 3, 3\rangle$ states, we find $ab \cdot a = ab \cdot b = ab \cdot c = ba \cdot b = ab$, $ba \cdot a = ba$ and finally $ba \cdot c = b$. This iteration generated no new states, so the process is complete. And this does indeed conform to the structure shown in Figure 5.1.

The Cayley graph corresponding to the syntactic semigroup in Figure 5.1 is shown at the top in Figure 5.1.

Table 5.1: The Cayley table for the syntactic semigroups in Figure 5.1 (left) and Figure 5.2 (right).

	a	b	c	ab	ba
a	a	ab	c	ab	ab
b	ba	b	b	ab	ba
c	a	c	c	ab	a
ab	ab	ab	ab	ab	ab
ba	ba	ab	b	ab	ab

	T	V	D	VT
T	D	V	D	VT
V	VT	V	D	VT
D	D	V	D	VT
VT	D	V	D	VT

Note that the complement of the language which forbids ab substrings – that is, the language which only accepts words with ab substrings – shares the same syntactic semigroup. This is because the whether states or accepting or not in the automata is not relevant to the action initiated by a transition. As such, an automaton and its complement share the same algebraic structure. It follows that classes defined in terms of properties of the syntactic semigroups will be closed under complement.

Now consider the transducer of Figure 5.2. This transducer is a representation of intervocalic voicing, a phonological process where voiceless obstruents become voiced between vowels. As a phonological rule this is T→D/V__V. For example, this transducer maps the string TVTVD to TVDVD.

The transducer above is in canonical form, where each state represents one $\overset{N}{\sim}$ class. State 2 is all those strings that end in V, state 3 is the strings ending in VT, and state 1 represents the others. The five mapping functions are the identity function $\langle 1, 2, 3\rangle$ corresponding to ε, $\langle 1, 1, 1\rangle$, $\langle 1, 3, 1\rangle$, and $\langle 2, 2, 2\rangle$ corresponding to D, T, and V, respectively, and finally $\langle 3, 3, 3\rangle$ for VT. One can verify that for any pair of classes there is a context which distinguishes words in one from words in the other, and that no context distinguishes words within a class. For example, ε and T are distinguished by a V__V context, as for ε that following V contributes just V while for T that following V contributes a DV. Technically, $\langle \mathrm{V}, \mathrm{V}, \mathrm{V}\rangle \in C(\varepsilon)$ while $\langle \mathrm{V}, \mathrm{V}, \mathrm{DV}\rangle \in C(\mathrm{T})$, but by determinism the triples are

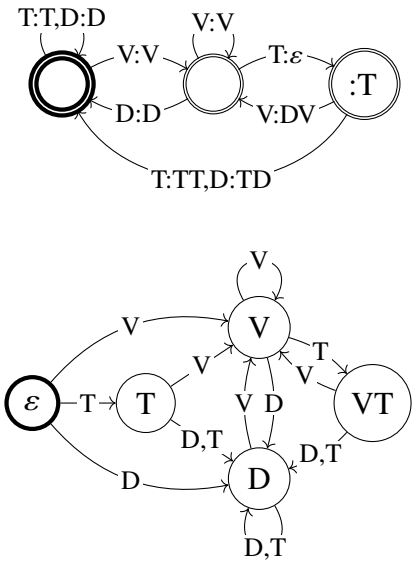


Figure 5.2: Transducer and monoid for "T becomes D directly between two V".

unique in their first two components. Then ε and D are distinguished by a VT__ε context, as the ε contributes T to the former and ε to the latter. That no context distinguishes strings within a class is guaranteed by the construction. The Cayley graph corresponding to the syntactic monoid in Figure 5.2 is shown at bottom in Table 5.1.

While output information has been discarded in this construction of the monoid, it is not unrecoverable, despite appearances. Outputs may be compatibly assigned to the states and edges and the result used as a nonminimal transducer. However, the structure is identical to that of the string language in which all words must end in "VT". This notion of structural equivalence gives rise to a deep theory of function complexity.

5.1.5 Definite Algebraic Structure

A string language L is **definite** if can be defined by a finite set X of permitted suffixes: $L = \{wv : w \in \Sigma^*, v \in X\}$ (Perles et al., 1963). The class of definite languages is denoted **D**. Because X is a finite set there is some longest string in X whose length is n, and thus whether a string belongs to L

can be decided by examining the last n symbols in the string. Such languages are called n-definite. This same class of languages has also been called local (Sakarovitch, 2009), derived from earlier use of the French *fonctions p-locales* (Berstel, 1982; Vaysse, 1986). More generally, as the canonical acceptor for a definite language processes strings, the states correspond to strings in Σ^n that represent the most recent history. In this sense, the state space of definite languages is Markovian in the sense explained by Jurafsky and Martin (2009).

The definite languages were one of the early classes of formal languages to be given an algebraic characterization (Brzozowski and Simon, 1973; Brzozowski and Fich, 1984). Many algebraic structures are defined in terms of idempotents. An element e of a monoid is **idempotent** if $e \cdot e = e$. As an example, the idempotents of the syntactic semigroups shown in 5.1 are $\{a, b, c, ab\}$ and $\{\mathrm{V}, \mathrm{D}, \mathrm{VT}\}$, respectively. Denote the set of idempotents with E.

An algebraic property characterizes exactly the definite languages (Brzozowski and Simon, 1973; Brzozowski and Fich, 1984). The syntactic semigroup of any definite language L has the property that for all $e \in E$ and for all $x \in S$ it holds that $xe = e$. This is commonly expressed as $Se = e$ with the universal quantification over e left implicit.

The string language which forbids ab substrings is not definite. This follows from the algebraic characterization and from the Cayley table for this language in Table 5.1. While b is an idempotent (since $b \cdot b = b$), $a \cdot b = ab \neq b$. Thus $Se \neq e$.

On the other hand, when we consider the idempotents e of the intervocalic voicing function (V, D, and VT), it is the case that $Se = e$. This can easily be verified by inspection of their respective

columns in the Cayley table in Table 5.1. In other words, the most recently read symbols determine the state of a minimal sequential transducer of a definite function as it processes some input string.

The syntactic semigroups such that $Se = e$ form the variety **D** (Brzozowski and Simon, 1973; Brzozowski and Fich, 1984). It follows they are closed under the taking of submonoids, quotients and finite direct product, and Boolean operations. And it follows then that the definite languages are also closed under the Boolean operations. The definite variety has played a key role in the development of an algebraic theory of recognizable languages (Straubing, 1985).

5.2 Input Strictly Local Functions

Chandlee et al. (2014) define input strictly local transducers by a restriction on the tails, which induces a canonical transducer structure. A function is input strictly local iff there is some natural number k such that the function is definable by a sequential transducer whose states are labeled by $\Sigma^{<k}$, whose initial state is that labeled by ε, and whose edges are of the form $\delta(a, q) = \langle w, \text{Suff}^{k-1}(qa)\rangle$. The suffix function is defined as expected:

$$\text{Suff}^n(w) = \begin{cases} \varepsilon & \text{if } n \leqslant 0, \\ w & \text{if } |w| \leqslant n, \\ v & \text{if } w = uv \text{ for } u \in \Sigma^*, v \in \Sigma^n. \end{cases}$$

Conveniently, this canonical form is already a monoid. The operation $u \cdot v = \text{Suff}^{k-1}(u \cdot v)$ is associative, and ε is the identity. Let f be a function, $\overrightarrow{S}$ and $\overleftarrow{S}$ be the syntactic semigroups of the left-to-right and right-to-left transducers associated with f, respectively, and e range over all idempotents of the appropriate semigroup.

Theorem 5.1. *The following are equivalent:*

- f *is a total input strictly local function*
- f *is* $\rightarrow$ **D***:* $\overrightarrow{S}e = e$
- f *is* $\leftarrow$ **D***:* $\overleftarrow{S}e = e$

Proof. The nonidentity idempotent elements of this monoid are Σ^{k-1}, as when $x \in \Sigma^{k-1}$ we have $x = \text{Suff}^{k-1}(x) = \text{Suff}^{k-1}(xx)$, and when $x \in \Sigma^{<k-1}$ we instead have $x \neq \text{Suff}^{k-1}(xx)$. But if $x \in \Sigma^{k-1}$ we have that $\text{Suff}^{k-1}(ux) = x$ for all $u \in \Sigma^*$, so for all elements s of the monoid it holds that $s \cdot x = x$. In other words, $Se = e$ for all idempotent elements e in the syntactic semigroup (which excludes the identity). This is exactly the semigroup property that characterizes the class of definite string languages, which are defined by a set of permitted suffixes (Brzozowski and Simon, 1973; Brzozowski and Fich, 1984).

The directionality statement follows from the fact that input strictly local functions are not directional (Chandlee and Heinz, 2018). ■

This makes sense, as the canonical form of an input strictly local transducer is exactly the same as the canonical form of a definite string language (Perles et al., 1963). Both are defined by the next state being entirely predicted by the most recent symbols encountered, with no long-distance effects at all. Indeed, exactly this class of functions has been discussed as the class of definite

functions decades before Chandlee et al. (2014) introduced them as input strictly local functions to the discourse of linguists (Krohn et al., 1967; Stiffler, 1973).

Strictly local string languages in general follow the same structure but additionally allow some of the states to become rejecting sinks instead of transitioning to the otherwise expected states. The result of these changes does not necessarily retain the algebraic structure, but then a semigroup can be regenerated by the usual method. Some long-distance behavior is enabled by this ability to account for whether a factor in some fixed set has ever occurred.

We invoke this characterization of input strictly local functions as definite structures to provide an effective decision procedure for determining whether an arbitrary finite-state transducer represents an input strictly local map. First, it is converted (if possible) to a canonical sequential form by the algorithm of Mohri (1997). If this conversion is impossible, the map is certainly not in this class, as it is not even sequential. If on the other hand it is, then the syntactic semigroup is constructible by the algorithms shown in section 5.1 (McNaughton and Papert, 1971). After extracting the idempotents, one needs only to check that the column in the Cayley table of the semigroup labeled by each idempotent e consists of only e. Recall that the identity is in the semigroup iff it is reachable by a nonempty string.

It should be noted that being able to define a process as a composition of input strictly local processes does not guarantee that the process as a whole has the same property, as composition does not preserve structure or variety membership. Consider two ISL functions, the first describing vowel-span truncation, V→∅/V__, and the second describing simultaneous application of T→D/TV__ and D→T/DV__, essentially harmonizing voicing over a single vowel. The composition of these,

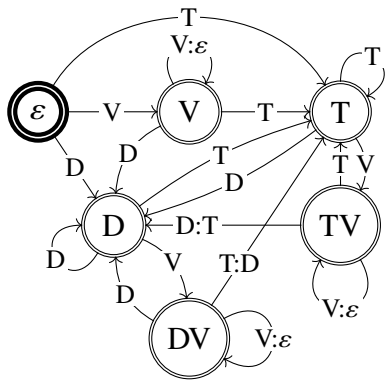


Figure 5.3: A non-ISL function composed from two ISL functions.

applying the second to the result of the first, yields a system, shown in Figure 5.3 that does not satisfy the algebraic property of definiteness. V is idempotent, so any element followed by V should yield V, but the element DV yields DVV = DV instead. As the class of definite functions is not closed under composition, defined by membership in the variety **D**, this operation cannot preserve algebraic structure or variety membership.

5.3 Output Strictly Local Functions

For the input k-strictly local functions, state is determined by the most recent $k-1$ symbols read from the input. A different generalization to functions of the strictly local languages is the output k-strictly local functions, where state is determined by the most recent $k-1$ symbols of the output (Chandlee, 2014; Chandlee et al., 2015). I show here by construction that this class is not characterized by any property of these syntactic semigroups.

Let us first consider some basic exemplars of the class. One canonical example is an iterative spreading process, such as that shown in Figure 5.4: upon reading a nasal (N or $\tilde{\text{V}}$), an immediately subsequent span of vowels (V) becomes nasalized ($\tilde{\text{V}}$). This is a progressive spreading pattern if the

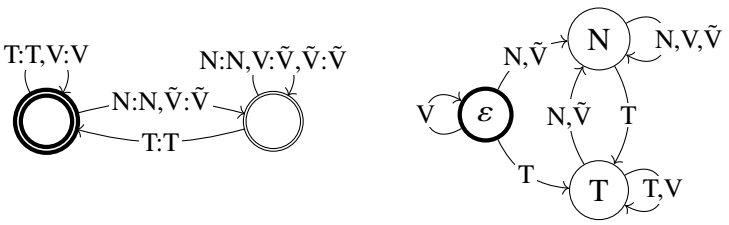


Figure 5.4: Iterative spreading of nasality: an output strictly local function. Minimal transducer at left, syntactic monoid at right.

Table 5.2: A Cayley table for nasal spreading, after tier restriction.

	N	T
N	N	T
T	N	T

input is read left to right, or a regressive iterative spreading if it is read right to left. What are the algebraic properties of this pattern?

Following the approach of the previous chapter, one can classify this pattern in many ways. Recall the hierarchy shown in Figure 4.16. We can first show that this pattern is tier-based definite (a subclass of tier-based generalized definite, TGD). First, notice that V acts as an identity in the syntactic monoid, which means that, perhaps surprisingly, this symbol is not on the tier of symbols salient for this pattern. Upon removal of this symbol, ε is no longer included in the corresponding semigroup, and the remaining elements are N and T. The Cayley table for this restriction is shown in Table 5.2. All elements are idempotent, and both columns consist entirely of a single element. In other words, this restricted semigroup satisfies the algebraic property of definiteness. The spreading pattern is tier-based definite, either $\rightarrow [\![\mathbf{D}]\!]_{\mathrm{T}}$ for a progressive spreading, or $\leftarrow [\![\mathbf{D}]\!]_{\mathrm{T}}$ for a regressive spreading.

Let us consider the $\mathscr{J}$-classes of the monoid for this spreading pattern, in order to test for membership in PT. Recall from the previous chapter that two elements a and b are $\mathscr{J}$-equivalent iff they have

the same two-sided ideals, $MaM = MbM$. The two-sided ideal of ε is $\{\varepsilon, \mathrm{N}, \mathrm{T}\}$, or in other words M itself. This always holds for the identity. That of N is $\{\mathrm{N}, \mathrm{T}\}$, as is that of T. Because these two elements have the same two-sided ideal, they are $\mathscr{J}$-equivalent. But since they are not themselves equal, it follows that this pattern is not PT. In fact, because ε is an idempotent in S, it follows that this pattern is not even locally $\mathscr{J}$-trivial; in other words, this type of spreading pattern looks quite complex from both local and piecewise branches of the subregular hierarchy, and only relativized adjacency shows its simplicity.

As an aside, the previous section notes that $\rightarrow \mathbf{D}$ and $\leftarrow \mathbf{D}$ are one and the same. This spreading pattern demonstrates that the same does not hold in general: it cannot be recognized by any sequential transducer that reads in the opposite direction. Figure 5.5 shows a small nondeterministic machine for a reversed reading. Aside from the initial state, there is a nasalizing state and a nonnasalizing state. The nasalizing state requires a nasal to immediately follow (in order to license the feature), while the nonnasalizing state forbids an immediately following nasal (as such a segment should spread). It can be verified that this transducer cannot be determinized: no matter how long the span of V, it must be the case that $\mathrm{V}^n\mathrm{N}$ and $\mathrm{V}^n\mathrm{T}$ map the entire span differently. For a deterministic machine, each V must output ε, and a span of the correct length must be emitted before the following consonant. In order to guarantee this, there must be one state per possible length, and because this length is unbounded, the number of states is not finite. Iterative spreading in the same direction as the string is read is tier-based definite, but spreading in the opposite direction is not even sequential.

Let us now consider another output strictly local function, specifically the one depicted in Figure 5.6. This one lacks phonological motivation, but provides evidence that OSL in general can span a wide

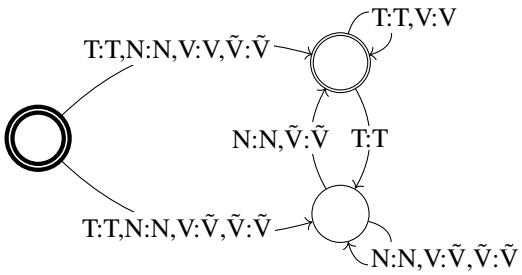


Figure 5.5: Nondeterministic transducer for nasal spreading opposite the read direction.

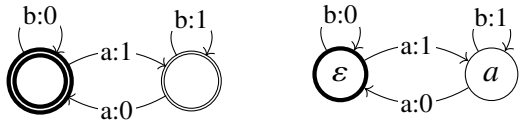


Figure 5.6: Periodic 2-OSL function and its monoid.

range of complexity classes from the algebraic perspective. This function outputs streams of zero and one, beginning with zero and swapping upon reading an "a". This type of function is useful in computer science, essentially implementing a conversion from a bit stream to NRZI coding. Its syntactic semigroup is however not even aperiodic; a generates a nontrivial group consisting of the identity (ε) and the self-inverse a. Algebraically, this function is properly sequential, nothing lower.

Indeed, for any finite semigroup S, one can construct a k-OSL function whose transition semigroup is precisely S. For each element x of S^1, construct a unique state and a unique string of length k; let $q(x)$ denote the state corresponding to x, and $w(x)$ its corresponding string. These strings will be possible outputs for transitions. Further, select a subset $G \subseteq S^1$ such that $G^+ = S$, and assign for each element g in G a unique symbol $\sigma(g)$. These symbols make up the input alphabet. For all elements x of S^1 and y of G, construct an edge from $q(x)$ to $q(xy)$ labeled $\sigma(y) : w(xy)$.

For instance consider the syntactic semigroup shown by Cayley table as Table 5.3. Let $G = \{a, b\}$, let σ map a to "a" and b to "b", and let w map a to "x", b to "y", and c to "z". The transducer

Table 5.3: An arbitrary syntactic semigroup.

	a	b	c
a	a	c	c
b	c	b	c
c	c	c	c

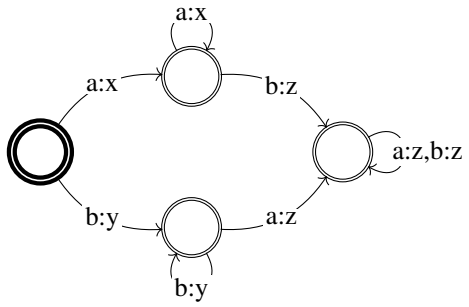


Figure 5.7: A transducer derived from Table 5.3.

constructed by this method is shown in Figure 5.7. The result is a 2-OSL function, and in fact this holds in general: the result is a (n+1)-OSL function where n is the length of the longest string in the range of w.

5.4 Harmony: Not Strictly Local

Simultaneous application of a phonological rule A→B/C__D, where CAD represents a finite set, is represented by a definite function, an input strictly local function. Iterative spreading is output strictly local, and specifically as shown in the previous section, it is tier-based definite. This is more complex than a simultaneous application, but not by much. In the case of string acceptors, a tier-based strictly local language can be learned using nothing beyond a standard strictly local learner (Lambert, 2021a), and a similar generalization may be applicable to functions. Some phonologically relevant functions cannot be represented as either. For example, consider the sibilant harmony pattern of Samala (Applegate, 1972), in which "s" and "ʃ" may not appear in the same word. A $\overset{M}{\sim}$-minimal transducer, isomorphic to the transition monoid, for this function is shown in Figure 5.8,

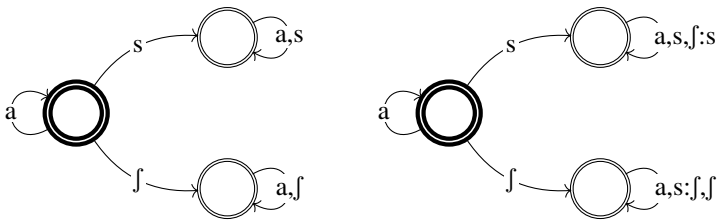


Figure 5.8: Samala sibilant harmony: $\overset{N}{\sim}$-minimal trimmed acceptor (left) and $\overset{M}{\sim}$-minimal transducer (right). Outputs omitted when identical to inputs.

alongside the strictly piecewise minimal acceptor for its output language.

One might wish to say that this function, like the corresponding acceptor, is strictly piecewise. Strictly piecewise is a subclass of piecewise testable, so, rather than attempt to define a strictly piecewise function, let us consider the $\mathscr{J}$-classes of the transition monoid. Let 1, s, and ∫ represent the equivalence classes of ε, s, and ∫, respectively. Then the two-sided ideal of 1 is $\{1, \text{s}, \int\}$, that of s is $\{\text{s}, \int\}$, and that of ∫ is also $\{\text{s}, \int\}$. The function is not even piecewise testable, as two unequal elements have the same two-sided ideal. What is it then?

Notice that there are nonsalient symbols, namely "a". Restricting to the tier of salient symbols, the semigroup S contains two elements, s and ∫. Each is idempotent, and each is a left-zero: $\text{s}S = \text{s}$ and $\int S = \int$. This is the algebraic property of **reverse definiteness**. In other words, this kind of harmony across transparent segments is tier-based reverse definite, where the output form is determined by the first k salient symbols.

Neither the most recent input nor the most recent output can determine state, as an unbounded span of input "a" results in the same span of output "a". But this function is still quite simple, residing in a subclass of tier-based strictly local. In fact, the acceptor is in TSL as well. What if there are

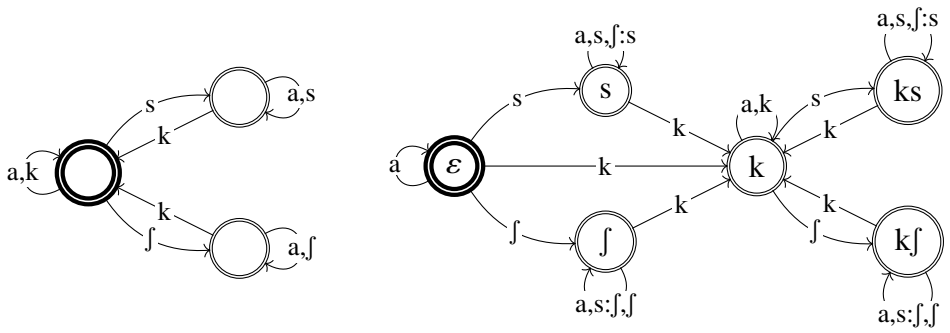


Figure 5.9: A variant of Samala sibilant harmony, adding blockers: $\overset{N}{\sim}$-minimal trimmed acceptor (left) and $\overset{M}{\sim}$-minimal transducer (right).

blockers? This does not cause the acceptor to escape TSL, but observe in Figure 5.9 the increase in complexity one creates by adding a blocking symbol "k" to the alphabet.

Every element is idempotent, so the complexity can be no higher than $\text{FO}^2[<]$ as described in the previous chapter. However, even the highest class that does not contain this on the tier-based branch, tier-based locally $\mathscr{J}$-trivial, is insufficient to describe this function. To show this, first we restrict to the salient symbols, removing "a" and leaving a five-element semigroup, where again all elements are idempotent. The local subsemigroups are listed in Table 5.4. Most of them are trivial; they cannot serve as counterexamples. But the local subsemigroup generated by "s" will do nicely. It has three elements: s, ks, and kʃ. The identity is "s", so its two-sided ideal is naturally the entire set. The two-sided ideal of both "ks" and "kʃ" is the set $\{\text{ks}, \text{k}\int\}$. This suffices to establish that the local subsemigroup is not $\mathscr{J}$-trivial, that the system as a whole is not even tier-based locally $\mathscr{J}$-trivial. In other words, the lowest complexity class in Figure 4.16 that contains this function is that denoted by $\text{FO}^2[<]$. One can verify also that the tier-based locally $\mathscr{L}$-trivial class, above TLP but incomparable with $\text{FO}^2[<]$, does contain this function: in no local subsemigroup do two distinct elements share the same left ideal. This is a relativized variant of the local extension of a generalization of the definite structure, not simple at all.

Table 5.4: Local subsemigroups from harmony with blockers.

e	eSe
k	{k}
s	{s, ks, kʃ}
ʃ	{ʃ, ks, kʃ}
ks	{ks}
kʃ	{kʃ}

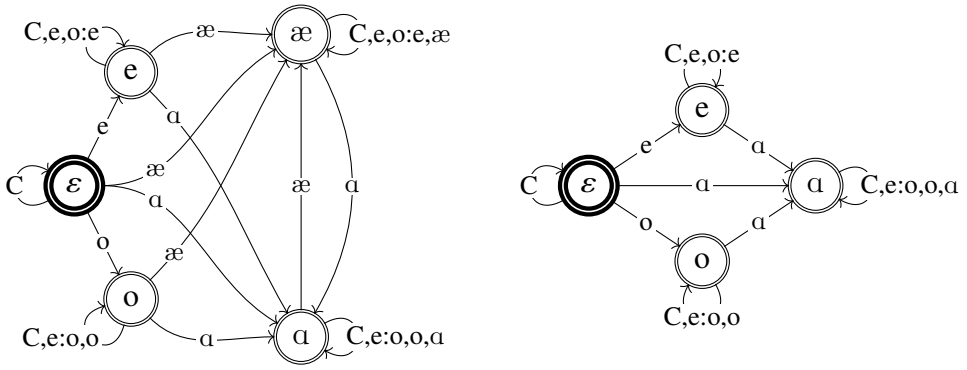


Figure 5.10: A symmetric (left) and an asymmetric (right) override of harmony.

This sort of blocking pattern is essentially a reset. Seeing the opaque segment places the computation back into its initial state, where the next harmonizing segment dictates the pattern. However, this is not the only possible kind of blocking process. Rather than a reset, we may encounter an override, such as the two functions shown in $\overset{M}{\sim}$-minimal form in Figure 5.10. In both cases, the mid vowels "e" and "o" harmonize. An "ɑ" overrides the harmony to the back ("o") pattern, and an "æ" overrides to the front ("e") pattern. Consonants ("C") are of course transparent to this process.

One can verify that the symmetric pattern has exactly the same complexity as the process with a full reset: it is $\text{FO}^2[<]$ or tier-based locally $\mathscr{L}$-trivial. The asymmetric process, however, is simpler. It is still $\text{FO}^2[<]$ on one branch, but more tractable with relativized adjacency. Consonants are not salient, and after stripping them away the identity element is no longer in its semigroup; the remaining elements are "e", "o", and "ɑ". Their local subsemigroups are {e, ɑ}, {o, ɑ}, and {ɑ}, respectively,

and since each is a semilattice, the process as a whole is merely tier-based locally testable.

5.5 Ambiguous and Two-Way Transducers

The classification algorithms of the preceding sections are built upon the same mechanism as for acceptors. But they work only on sequential machines, which maintain a property of order-preservation (Bojańczyk, 2014; Filiot, 2015). Carton and Dartois (2015) provide a more general mechanism for constructing semigroups that describe functions, which applies equally well to nondeterministic and two-way machines, generalized from the analysis of two-way acceptors by Pécuchet (1985). Their main result is that first-order definable graph transductions in the sense of Courcelle (1994) correspond to two-way transducers with aperiodic semigroups, generalizing the correspondence between first-order definable languages and aperiodic semigroups. This method of constructing transition monoids is slightly more difficult than that used to describe sequential functions. Worse, two-way machines in general are plagued by a lack of a canonical form, so one can only show that a class is sufficient to describe a function; necessity is unprovable unless such a form can be found. This section serves only to introduce the methodology with respect to two phonologically relevant functions that cannot be represented by a sequential transducer: high-tone plateauing (Jardine, 2016) and Tutrugbu ATR-harmony (McCollum et al., 2020). We begin with high-tone plateauing. This process transforms a low tone into a high tone iff there is a high tone somewhere on either side of it. Figure 5.11 depicts a nondeterministic one-way transducer for precisely this function.

Essentially the contexts of a string as defined in section 5.1.3 could be thought of as state-pairs in the canonical transducer of the function, where the first component is the state from which reading

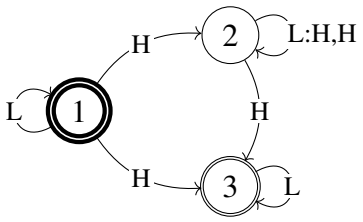


Figure 5.11: High-tone plateauing as a one-way nondeterministic machine.

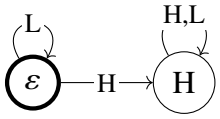


Figure 5.12: Transition monoid for the one-way high-tone plateauing.

begins and the other component is the state in which the computation concludes. This is generalized for the case of two-way machines by collecting four variants of these contexts: one can begin at either the leftmost or rightmost character of the string, and one can end by exiting the left side or the right side. These are referred to by Carton and Dartois (2015) as **behaviors** and are denoted $\mathrm{bh}_{\ell\ell}$, $\mathrm{bh}_{\ell r}$, $\mathrm{bh}_{r\ell}$, and bh_{rr}, for left-to-left, left-to-right, right-to-left, and right-to-right behaviors, respectively. For a left-to-right one-way machine, the behaviors that describe exiting the left edge will always be the empty set, and similarly for a right-to-left one-way machine, those describing exiting the right edge will be the empty set. The four tuple $\mathrm{bh}(w) = \langle \mathrm{bh}_{\ell\ell}(w), \mathrm{bh}_{\ell r}(w), \mathrm{bh}_{r\ell}(w), \mathrm{bh}_{rr}(w) \rangle$ describes the overall behaviors of a string w, and a semigroup can be formed by taking a quotient of Σ^+ over the relation wherein two strings u and v are equivalent iff $\mathrm{bh}(u) = \mathrm{bh}(v)$. When describing a one-way machine, only the set of behaviors that matches the read action of the machine matters; for a left-to-right machine, only the left-to-right behaviors are relevant. Table 5.5 shows the left-to-right behaviors of this machine up to the point where all equivalences are determined. Notice that L acts as an identity, in that LL = L and LH = HL = H. The generated monoid is shown in graph form in Figure 5.12.

Table 5.5: Behaviors of high-tone plateauing.

w	$\mathrm{bh}_{\ell r}$	$\equiv$
L	$\{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$	
H	$\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,2\rangle, \langle 2,3\rangle\}$	
LL	$\{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$	L
LH	$\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,2\rangle, \langle 2,3\rangle\}$	H
HL	$\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,2\rangle, \langle 2,3\rangle\}$	H
HH	$\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,2\rangle, \langle 2,3\rangle\}$	H

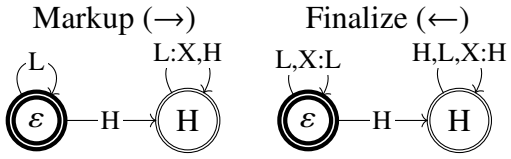


Figure 5.13: High-tone plateauing decomposed into two sequential functions.

This is a two-element idempotent monoid, so it is a semilattice, both 1-PT and 1-LT. When restricted to the tier of nonidentity elements, the resulting semigroup has only a single element: H. Thus the pattern additionally satisfies the algebraic properties of the tier-based (co-)finite class.

One could also decompose this function into two: a left-to-right sequential function that marks up the input followed by a right-to-left sequential function that finalizes the output. The two components are shown in Figure 5.13. Both L and X are everywhere self-loops, meaning they are nonsalient. Restricting to the tier of salient symbols, H alone, the corresponding semigroups have only a single element. Each function is therefore tier-based (co-)finite, just like the nondeterministic one-way instantiation. That these decomposed complexities are low can only be relevant, however, if this kind of separated two-pass approach is used, as composition fails to preserve essential algebraic properties.

One may consider one final way of describing this process: a single two-way function. Figure 5.14 shows such a machine, where $\rtimes$ and $\ltimes$ represent left and right boundary markers, respectively.

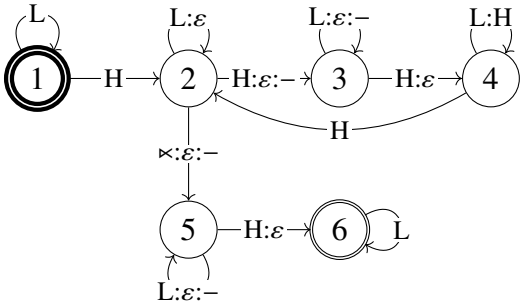


Figure 5.14: High-tone plateauing as a two-way transducer.

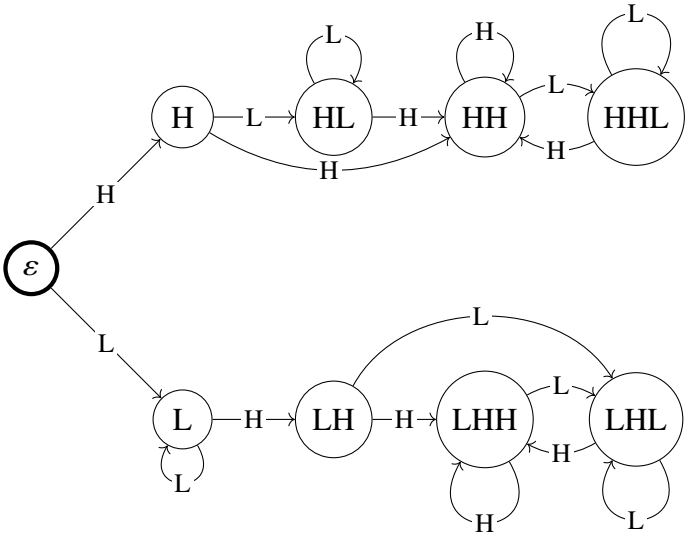


Figure 5.15: Monoid generated from Figure 5.14.

Carton and Dartois (2015) suggest generating the behaviors of all and only those words that do not contain boundary symbols in order to construct a transition monoid, and the result of doing so is Figure 5.15.

Table 5.6 is the Cayley table of this semigroup. There are five idempotents, L, HH, LHL, LHH, and HHL, whose local subsemigroups are {L, LHL}, {HH}, {LHL}, {LHH}, and {HHL}, respectively. Four of these are singleton, and therefore trivially semilattices. The remaining one is a two-element idempotent monoid, and therefore also a semilattice. Each local subsemigroup is a semilattice, and thus this two-way function satisfies the algebraic property of local testability. Because the local

Table 5.6: The Cayley table of Figure 5.15, idempotents boxed.

	L	H	LH	HL	HH	LHL	LHH	HHL
L	[L]	LH	LH	LHL	LHH	LHL	LHH	LHL
H	HL	HH	HH	HHL	HH	HHL	HH	HHL
LH	LHL	LHH	LHH	LHL	LHH	LHL	LHH	LHL
HL	HL	HH	HH	HHL	HH	HHL	HH	HHL
HH	HHL	HH	HH	HHL	[HH]	HHL	HH	HHL
LHL	LHL	LHH	LHH	LHL	LHH	[LHL]	LHH	LHL
LHH	LHL	LHH	LHH	LHL	LHH	LHL	[LHH]	LHL
HHL	HHL	HH	HH	HHL	HH	HHL	HH	[HHL]

subsemigroup generated by L is not trivial, this function is not (tier-based) generalized definite, unless a simpler structure can be found. Thus, the simplest analysis is the single one-way nondeterministic transduction, tier-based (co-)finite.

These types of patterns in which the output associated with a given input segment depends on information both to its left and to its right which may be unboundedly far away is known as an unbounded circumambient pattern (Jardine, 2016). Jardine (2016) shows that this type of pattern is not **weakly deterministic** in the sense of Heinz and Lai (2013); essentially, any deterministic decomposition requires markup through some additional symbol. This was used as a basis to claim that tonal phonology differs from segmental phonology in terms of computational complexity. McCollum et al. (2020) note that unbounded circumambient processes occur in segmental phonology as well, citing several examples from vowel harmony systems and concluding that these patterns refute a subregular analysis of phonology. One such pattern is the vowel harmony of Tutrugbu, where a [+ATR] spreads leftward from the root to prefixes, targeting all vowels, but if the vowel of the first syllable is [+high] then [−high] vowels act as blockers (McCollum et al., 2020). A right-to-left one-way nondeterministic finite-state transducer is provided by McCollum et al. (2020); the same machine is provided here in Figure 5.16 after removing the transitions on boundary symbols,

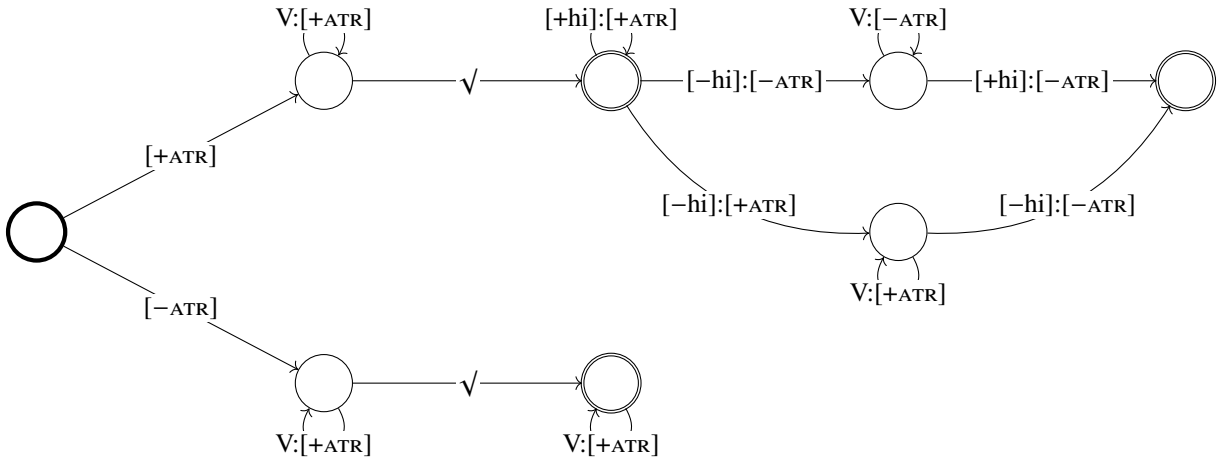


Figure 5.16: Nondeterministic transducer for Tutrugbu ATR harmony.

reassociating the final states accordingly, and merging equivalent states. The $\surd$ symbol represents a distinguished boundary between the root (read first) and the prefix (read later).

This is a fairly standard way of representing the pattern; the ATR value of the root dictates the pattern, then at some point there is a boundary symbol, and the prefix undergoes some potential change. As in the source material, the consonants have been omitted, as they are nonsalient. One can verify by constructing the seventeen-element syntactic monoid that this machine falls properly into the $\text{FO}^2[<]$ class, and is also (tier-based) locally $\mathscr{L}$-trivial. However, this is not the version I would like to discuss. Rather than using a distinguished boundary symbol between morphemes, one may wish to encode root and affix segments differently. Consider the transducer in Figure 5.17, where a subscript R represents a root segment, A an affix segment, and a lack of subscript denotes either. Consider a four-vowel system: i is [+high, +ATR], ɪ [+high, −ATR], o [−high, +ATR], and ɔ [−high, −ATR]. Under this alphabet the generated monoid still has seventeen elements, just like the other description, but the complexity is lower: it remains in $\text{FO}^2[<]$ but is now only (tier-based) locally testable. Note that this transducer assigns an arbitrary behavior to root segments that are read after having read an

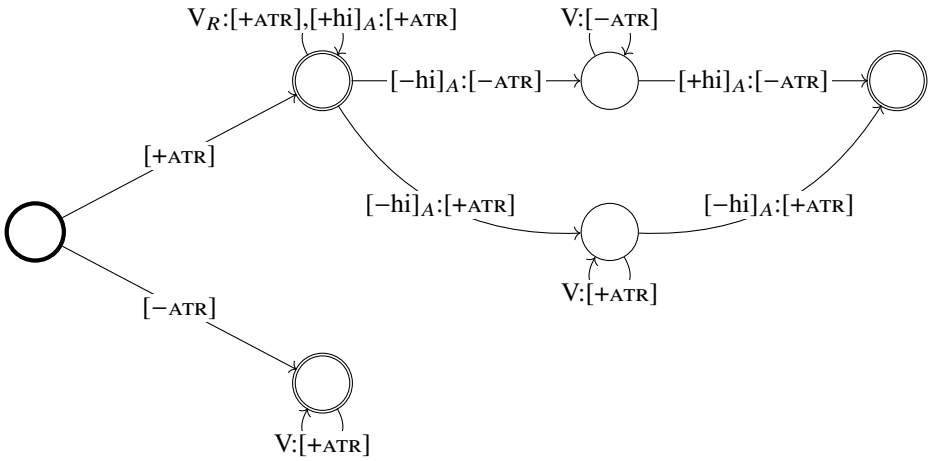


Figure 5.17: Tutrugbu ATR harmony with separate symbols for roots and affixes.

affix segment. A different assignment may increase or decrease complexity without changing how the process behaves on well-formed words.

5.6 Conclusions

For deterministic one-way transducers, the exact algorithms for classifying string acceptors by algebraic means generalize, and the resulting algebraic structure induces a graph that can be assigned transition labels in such a way that the original function is preserved. Total input strictly local maps suffice to describe a large number of phonological processes, and correspond exactly to the algebraic class of definite functions. Output strictly local maps are not so easy to describe algebraically; the only class that properly contains k-OSL, for $k \geqslant 2$, is sequential itself. However, the canonical phonologically relevant example of an OSL function, iterative spreading, is simply tier-based definite. Long-distance harmony patterns such as Samala sibilant harmony can be tier-based reverse definite if there are no opaque segments, but if such segments are present then complexity jumps quite a bit. Finally, nonsequential functions may be analyzed by a further generalization of this process. High-tone plateauing is in the simplest of classes: CB (semilattices) and tier-based (co-)finite, and Tutrugbu vowel harmony can be analyzed in such a way that tier-based locally testable suffices.

Nondeterministic and two-way transducers lack the canonical form afforded by sequential functions, so while this is a sufficient complexity, it may be more than is truly necessary.

Aside from some types of long-distance harmony with opaque segments, all of these patterns are at most tier-based locally testable, one of the simpler classes of the piecewise-local subregular hierarchy. The output structure is defined based on the set of salient k-substrings that have appeared so far in the input as well as the most recent $(k-1)$ input symbols.

Part II

Chapter 6: Learning Tier-Based Strictly Local Languages

The strictly local (SL) class known from McNaughton and Papert (1971) cannot account for long-distance dependencies. Meanwhile the strictly piecewise (SP) class of Rogers et al. (2010, see also Haines, 1969) can account for long-distance dependencies but cannot handle local constraints. In 2011, Heinz et al. presented a new tier-based strictly local (TSL) class of stringsets, generalizing SL to account for both local phenomena and certain types of long-distance phenomena by relativizing adjacency over a subset of the alphabet. Much like the traditional SL and SP classes, TSL is parameterized by the width k of the factors to which a learner or acceptor attends. But there is another parameter. If the subset T of the alphabet over which adjacency is relativized is known *a priori*, then the input data can be preprocessed by deleting unnecessary symbols and the grammar inferred by any of a number of strictly local learning algorithms, such as that of Garcia et al. (1990) or that of Heinz (2010b). But learning T from data alone has proven difficult, especially in a bounded-memory setting.

Since the introduction of the TSL class, it has been characterized model-, language-, and automata-theoretically by Lambert and Rogers (2020), extended to an even richer class by De Santo and Graf (2019), and shown to be batch-learnable in the style of Gold (1967) by Jardine and Heinz (2016) for k of 2 and by Jardine and McMullin (2017) for arbitrary k. But despite all this progress, no general online learning algorithm has yet been produced, leaving the TSL class behind in an area where several other subregular classes flourish.

That changes now. Here we discuss an online learning algorithm in the style of Heinz (2010b) that essentially combines the approaches taken in the earlier batch-learning algorithms with a novel

representation of the TSL grammar to accommodate online learning. This development removes one argument against TSL descriptions of phonological patterns, namely that human learners likely do not operate in batch. (Batch learning requires perfect awareness of all prior input, which is implausible at best for human learners.)

Section 6.1 discusses background material. Then section 6.2 summarizes prior literature on discovering the set of salient symbols and provides space- and time-complexity analyses of the algorithms. Section 6.3 introduces a structure that can be gathered while determining salience, which provides sufficient information to recover the grammar of forbidden factors while avoiding unbounded data storage. Then section 6.4 demonstrates a pointwise application of the string extension learning algorithm of Heinz (2010b) to this problem, and introduces a simplification that allows for reinterpretation of an SL grammar as a TSL one.

6.1 Preliminaries

This section contains background material fundamental to this work. Section 6.1.1 provides a brief overview of the SL and TSL classes, with examples. Section 6.1.2 discusses the learning framework in use, and section 6.1.3 details the family of learning algorithms that includes our result.

6.1.1 (Tier-Based) Strict Locality

In general, a **factor** is some substructure of a word that is connected in some sense. For the SL languages as defined by McNaughton and Papert (1971), these substructures are simply adjacent sequences of symbols. For example, there are seven factors in the string "abc": "", "a", "b", "c", "ab", "bc", and "abc" itself. Notably, "ac" is not a factor under this interpretation. The size of a factor is the length of the sequence. An SL language is characterized by a set of forbidden factors,

containing all and only those strings in which no forbidden factor occurs. If the largest factor in the set of forbidden factors is of size k, then the language is k-SL.

One example of an SL language over the alphabet $\Sigma = \{a, b, c, d\}$ is that in which the forbidden factors are "aa" and "bb". This requires that "a" and "b" alternate within blocks of these two symbols, so "abaca" and "abada" are valid words but "**aa**ca" is not.

One can prove that a language is not k-SL by using what is known as Suffix Substitution Closure: finding two valid words $w = w_p x w_s$ and $v = v_p x v_s$ that share a common factor x of length $k - 1$ where $w_p x v_s$ is not a valid word (Rogers and Pullum, 2011, see also De Luca and Restivo, 1980). If such w and v can be found for any k, then the language is not SL.

Consider a slight modification to this example language: not only are "aa" and "bb" forbidden, but also "ada", "bdb", "adda", "bddb", etc. Essentially, "d" is invisible to the constraint. Now $w = ad^{k-1}b$ and $v = bd^{k-1}a$ are both valid words, but after suffix-substitution we have "$ad^{k-1}a$" which is not valid. Thus this pattern is not SL. The TSL class seeks to capture exactly the constraints that would be SL if only some category of symbols could be ignored. It is defined by applying an SL grammar not to the words themselves but to their projection to a **tier alphabet** T (Heinz et al., 2011). In this case, $T = \{a, b, c\}$ and the grammar is the same as before. See Chapter 3 for further information on the TSL class, including how to decide membership. Note that while the previous discussion involved forbidden factors, the finiteness of both the factor width and alphabet size allows an equivalent description in terms of permitted factors.

6.1.2 Our Learning Problem

Gold (1967) introduces a number of learning paradigms, but here we will focus on the case when a language is learned in the limit from distribution-free positive data. We further restrict ourselves to consider only online learning, where the induction function takes as arguments not an entire input set, but only a single input item along with a previously proposed grammar. This section formalizes these notions.

Let $L \subseteq \Sigma^*$ be a stringset and let $L_\circledcirc$ be L with an adjoined element $\circledcirc$ that represents the lack of any string. A text for L is a total, surjective function $t\colon \mathbb{N} \to L_\circledcirc$, an infinite sequence of strings drawn from L that contains each string at least once, and may at some points present no data. Note that for any non-empty stringset, there are infinitely many possible distinct texts. The addition of $\circledcirc$ is a deviation from the original work by Gold but without it no text exists for the empty stringset (Osherson et al., 1986). For a given text t, let $\vec{t}_n$ represent the sequence $\langle t_0, t_1, \ldots, t_n \rangle$, i.e. the initial segment of t of length $n+1$. We denote the class of texts for $L_\circledcirc$ by $\mathbb{T}$ and the class of initial segments thereof by $\vec{\mathbb{T}}$.

A grammar is some representation of a mechanism by which the membership of a string in a stringset may be decided. Let $\mathbb{G}$ be a set of possible grammars, and let $\mathscr{L}\colon \mathbb{G} \to \mathscr{P}(\Sigma^*)$ be a function that transforms a grammar into its extension, i.e. the set of all strings that it accepts. Two grammars G_1 and G_2 are equivalent (written $G_1 \equiv G_2$) iff they are extensionally equal, i.e. $\mathscr{L}(G_1) = \mathscr{L}(G_2)$. A batch learner is then a total function $\varphi\colon \vec{\mathbb{T}} \to \mathbb{G}$. In words, a batch learner is an algorithm that takes as input an initial segment of a text and outputs a guess at the correct grammar.

Given a text t and a learner φ, we say that φ converges on t iff there is some point after which its guess never changes. Formally, that means there exists some $i \in \mathbb{N}$ and some grammar G such that for all $j \geqslant i$, it holds that $\varphi(\vec{t}_j) \equiv G$. If given any text t for a stringset L, φ converges on t to a grammar G such that $\mathscr{L}(G) = L$, then we say that φ identifies L in the limit. For any class of stringsets $\mathbb{L} \subseteq \mathscr{P}(\Sigma^*)$, we say that φ identifies $\mathbb{L}$ in the limit iff it identifies L in the limit for all $L \in \mathbb{L}$.

An **online learner** differs from a batch learner only in how it assumes the data is presented. For a batch learner, the function is of type $\varphi\colon \vec{\mathbb{T}} \to \mathbb{G}$ as discussed. On the other hand, an online learner, sometimes called an **incremental learner**, is of type $\varphi\colon \mathbb{G} \times L_{\circledcirc} \to \mathbb{G}$, taking as input a previous guess and a single data point (Jain et al., 2007). Ideally the online learner can take more input over time in an efficient way while maintaining a bounded information store.

6.1.3 String Extension Learning

Heinz (2010b) described a general algorithm for learning (among others) stringsets that can be described by a set of permitted factors of length bounded above by k. For this case $\mathbb{G} = \mathscr{P}(\Sigma^k)$. In general, given a function $f\colon \Sigma^* \to \mathscr{P}(\Sigma^k)$ that maps a string to the set of factors it contains, one can define a batch learner φ_f as follows

$$\varphi_f(\vec{t}_i) \triangleq \begin{cases} \varnothing & \text{if } i = 0,\ t_i = \circledcirc \\ f(t_i) & \text{if } i = 0,\ t_i \neq \circledcirc \\ \varphi_f(\vec{t}_{i-1}) & \text{if } i \neq 0,\ t_i = \circledcirc \\ \varphi_f(\vec{t}_{i-1}) \cup f(t_i) & \text{otherwise.} \end{cases}$$

Effectively one begins with a guess of the empty grammar, and for each string provided, this guess is updated to include all factors encountered in that string. A factor is **attested** iff it appears in some string in the input. The online variant of this same function is identical except that strings are provided one at a time.

$$\varphi_f(G, w) \triangleq \begin{cases} G & \text{if } w = \circledcirc \\ G \cup f(w) & \text{otherwise.} \end{cases}$$

If the parameter k is fixed and known, this approach identifies the k-SL class in the limit. If this holds and further the parameter T is fixed and known, this approach also identifies k-TSL$^{\text{T}}$ in the limit. But in general when learning TSL generalizations, we want to be able to account for the case where T is unknown.

6.2 Deciding Salience

When learning a k-TSL$^{\text{T}}$ stringset, if T is not provided then a learner must discover not only the underlying SL constraints, but also the class of symbols that are salient for these constraints. The model-theoretic view of k-TSL$^{\text{T}}$ given by Lambert and Rogers (2020) makes this explicit in that the ordering relation never connects to a position containing a symbol outside of T. There is no way to even talk about the other symbols. It follows then that there is no way to restrict their occurrence, meaning they are freely insertable and deletable in all strings. This property is what allows for the salience-finding algorithm of Jardine and McMullin (2017). We discuss here a simplification of that original work.

Given the set of all factors under adjacency of width up to $k + 1$ in the stringset, a symbol x is freely insertable iff for each attested factor of width less than or equal to k, inserting x at each possible

point in turn results in an attested factor one symbol wider. Similarly, x is freely deletable iff for each attested factor of width $k+1$ that contains x, the removal of each instance of x in turn results in an attested factor one symbol narrower. This is a deviation from the original work in that Jardine and McMullin use only factors of widths in the range $k \pm 1$, but the formulation here avoids making a special case of shorter words. The symbols that are not both freely insertable and deletable are the salient ones.

In summary, let $\mathcal{F}_{k+1}$ represent all attested factors of width $k+1$ or less and $\mathcal{F}_k$ represent all those of width k or less, and define for each symbol $\sigma \in \Sigma$ two sets: $\sigma_\oplus$ containing all possible factors obtained by adding a single instance of σ to $\mathcal{F}_k$, and $\sigma_\ominus$ containing all possible factors obtained by deleting a single instance of σ from $\mathcal{F}_{k+1}$. Then the set of salient symbols is

$$\mathrm{T} = \{\sigma : \sigma_\oplus \not\subseteq \mathcal{F}_{k+1} \text{ or } \sigma_\ominus \not\subseteq \mathcal{F}_k\}.$$

Thus to decide salience we can use exactly the SL string extension learner described in section 6.1.3 and then post-process the resulting grammar by this algorithm.

In terms of time and space complexity, this portion of the algorithm is relatively efficient. There are $|\Sigma|^k$ possible factors of width k, and thus by summation there are $\frac{|\Sigma|^{k+2}-|\Sigma|}{|\Sigma|-1}$ possible factors of width up to $k+1$. Supposing we store one bit per possible factor that represents whether or not it is attested, we require $O(|\Sigma|^{k+1})$ bits. For each word, its factors can be found in linear time, and each factor can be marked as attested in this set in time logarithmic in the set's size. That is, for an input of size n, the time complexity of gathering factors to determine salience is $O(nk \log |\Sigma|)$.

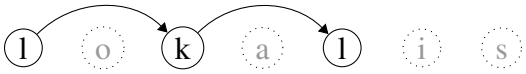


Figure 6.1: The tier-successor relation preserves linear order, but ignores certain symbols. Importantly, if a symbol is included then it is not also ignored.

6.3 The Substructures

Because Jardine and McMullin (2017) assume a batch-learning setting, they can simply learn the set of salient symbols, then erase all other symbols from the input and (in a second pass) analyze the result with any SL learner. Performing a second pass over the input requires this input to be retained. This, of course, results in unbounded space requirements and is therefore unsuitable for an online setting.

Fortunately, we can avoid this memorization of the input. A factor over relativized adjacency is made up of a sequence of symbols that appear in order, but not necessarily adjacently. These structures are, in general, referred to as **subsequences** (Heinz, 2010a; Rogers et al., 2010). Figure 6.1 shows one possible factor of width 3 ("lkl") in the string "lokalis". Crucially, when gathering subsequences, if a symbol is included in the subsequence, it can never later be excluded in that same subsequence. The factors "lki" and "lks" then would be invalid in this example and excluded from consideration because they both contain an "l" but go on to skip the second "l".

There are $\binom{n}{k}$ subsequences of width k in a word of length n, where this notation represents a binomial coefficient. It follows that the time complexity to find them is $O(n^k/k!)$, and the same holds when including smaller factors as well. If for each factor we store the set of symbols that intervened, much like the paths of Jardine and Heinz (2016), then we need to account for the time it takes to find the set associated with the particular factor and to mark the intervener-set as attested.

These are additional multipliers of $k \log |\Sigma|$ and $|\Sigma|$, respectively. So in total the time complexity is $O\big(n^k/(k-1)! \cdot |\Sigma| \log |\Sigma|\big)$.

Formally, if $w = \sigma_1 \ldots \sigma_n$ ($\sigma_i \in \Sigma$) is a string and $X = \langle i_1, \ldots, i_k \rangle$ ($k \leqslant n$) is an increasing sequence of indices, then the subsequence indicated by X is $Q = \langle \sigma_{i_1}, \ldots, \sigma_{i_k} \rangle$. The intervener-set is $\mathcal{I} = \{\sigma_j : i_1 < j < i_k \text{ and } j \notin X\}$. The pair $\langle Q, \mathcal{I} \rangle$ is the **augmented subsequence** indicated by X. A **valid subsequence** is one where no symbol appears in both Q and $\mathcal{I}$. Henceforth, any mention of subsequences is restricted to the valid ones.

Unfortunately it appears that to store all possible augmented subsequences, we would need space significantly beyond exponential in the size of the alphabet and factor width, $O(|\Sigma|^k \cdot 2^{|\Sigma|})$. But it turns out that we can exploit some structure in order to store significantly less. If a subsequence is in fact contiguous, that is it skips nothing, then no matter how adjacency is relativized that subsequence will still be an attested factor as long as it is valid. In fact a generalization holds: if a factor is attested with intervener-set $\mathcal{I}$, then it can also be assumed to be attestable for any superset of $\mathcal{I}$ for which it remains valid. So one needs only maintain the smallest observed intervener-sets (partially-ordered by subset). This means that the size of the set stored by any particular factor will never exceed $O\left(\binom{|\Sigma|}{|\Sigma|/2}\right)$, which is still exponential in $|\Sigma|$, but many factors will store just a single set: $\varnothing$. Given these interactions, we conjecture that the space complexity will often be sub-exponential in the size of $|\Sigma|$. Table 6.1 shows the possible augmented subsequences for $k = 2$ in the string "cabacba" and indicates which subset of those actually need to be maintained. However, we show in section 6.4 that we can avoid this source of space complexity entirely.

Table 6.1: In gathering augmented subsequences for "cabacba", many possibilities can be ignored. The intervener-sets are shown simply as sorted strings to avoid nested braces. Here, only the undecorated sets are maintained; those struck through were invalid from the start, while those in light gray are subsumed.

Factor	Intervener-Sets
aa	{b, ~~abc~~, bc}
ab	{∅, ~~abc~~, c}
ac	{~~ab~~, ∅}
ba	{∅, ~~abc~~}
bb	{ac}
bc	{a}
ca	{∅, ~~ab~~, ~~abc~~, b}
cb	{a, ~~abc~~, ∅}
cc	{ab}

Once the set of salient symbols T is known, we can derive a standard k-TSL$^{\text{T}}$ grammar from this set of augmented subsequences. A subsequence is a permitted factor iff all of the symbols that comprise it are salient and it is attested for an intervener-set that is disjoint with T. Otherwise it is a forbidden factor.

6.4 Pointwise String Extension Learning

In sections 6.2 and 6.3, we discussed two different kinds of substructure that can be gathered when learning k-TSL: the substrings of length bounded above by $k+1$, which allow us to determine which symbols are salient, and the augmented subsequences of length bounded above by k, which allow us to select a set of permitted factors once salience has been determined. If we let the hypothesis space $\mathbb{G} = \mathscr{P}(\Sigma^{\leqslant k+1}) \times \mathscr{P}(\Sigma^{\leqslant k} \times \mathscr{P}(\Sigma))$, then we can define a learner

$$\varphi(\langle G_\ell, G_s \rangle, w) \triangleq \begin{cases} \langle G_\ell, G_s \rangle & \text{if } w = \circledcirc \\ \big\langle G_\ell \cup f(w), r\big(G_s \cup x(w)\big) \big\rangle & \text{otherwise,} \end{cases}$$

where $f\colon \Sigma^* \to \mathscr{P}(\Sigma^{\leqslant k+1})$ gathers all and only those substrings of w whose width is bounded above by $k+1$, $x\colon \Sigma^* \to \mathscr{P}(\Sigma^{\leqslant k} \times \mathscr{P}(\Sigma))$ extracts the valid augmented subsequences of w of width bounded above by k, and $r\colon \mathscr{P}(\Sigma^{\leqslant k} \times \mathscr{P}(\Sigma)) \to \mathscr{P}(\Sigma^{\leqslant k} \times \mathscr{P}(\Sigma))$ restricts the set of augmented subsequences to exclude any that are entailed by any other. This is effectively two distinct string extension learners run in parallel, pointwise on the composite grammar.

The composite grammar can immediately be used as an acceptor without further processing. We replace the cost of deciding salience by that of finding augmented subsequences.

$$\mathscr{L}(\langle G_\ell, G_s\rangle) \triangleq \left\{w\colon f(w) \subseteq G_\ell \text{ and } r\big(G_s \cup x(w)\big) \subseteq G_s\right\}.$$

In words, a string is accepted iff it has only permitted substrings and each of its valid augmented subsequences is attested or entailed by something that is attested.

Depending on the parameters and the size of the input words, this strategy might be a good one. In other situations, it might be better to actually determine which symbols are salient. Recall that a text contains every valid word at least once, and that non-salient symbols are freely deletable in all strings. Free deletability of non-salient symbols implies that any subsequence that includes only salient symbols will, in some word of the text, have only its salient interveners as interveners. Those subsequences that do not violate constraints on the tier then must appear with an empty intervener set. In other words, such subsequences will appear as factors in terms of adjacency and will be accounted for by that component of the composite grammar.

Thus upon convergence the left component of this composite grammar, G_ℓ, is sufficient on its own to decide salience and to extract the grammar itself. G_s is unnecessary. Let $s\colon \mathbb{G} \to \mathscr{P}(\Sigma)$ be the function that decides salience in the manner described in section 6.2, and let $\pi_{\mathrm{T}}(w)$ represent the projection to T of w as described by erasing symbols that are not in T. Then we might have the following as an equivalent alternative definition

$$\mathscr{L}(G) \triangleq \left\{w\colon f\left(\pi_{s(G)}(w)\right) \subseteq G\right\}.$$

As an aside, the deletion-closure of the strictly piecewise class of stringsets (Rogers et al., 2010, see also Haines, 1969) enables this same sort of learning of long-distance patterns using only adjacent substrings.

Revisiting the time and space complexities mentioned previously, this optimized version of the grammar can be learned in $O(nk \log|\Sigma|)$ time and $O(|\Sigma|^k)$ space, exactly those values that were assumed for only the salience-decision component of learning. The time complexity is deferred to later, in interpreting the grammar as k-TSL rather than as $(k+1)$-SL.

6.5 A Worked Example of the Final Simplified Approach

Consider a blocked-assimilation constraint such as the liquid dissimilation of Latin (Cser, 2010), where identical liquids may not occur in sequence unless a non-coronal intervenes. This is equivalent to the example language described in section 6.1.1. We will assume for now that no other constraint interacts with this. Assuming an alphabet consisting of two distinct liquids ("l" and "r"), a

Table 6.2: Some words of varying degrees of phonological plausibility. Each set is in the order produced by a sliding 3-window. Factors already accounted for are in light gray.

akkalkak	{akk, kka, kal, alk, lka, kak}
klark	{kla, lar, ark}
kralk	{kra, ral, alk}
karlakalra	{kar, arl, rla, lak, aka, kal, alr, lra}
akrala	{akr, kra, ral, ala}
aklara	{akl, kla, lar, ara}
rakklarkka	{rak, akk, kkl, kla, ark, rkk, kka}
arkralkla	{ark, rkr, kra, ral, alk, lkl, kla}
laarlraalr	{laa, aar, arl, rlr, lra, raa, aal, alr}
kaaakkrka	{kaa, aaa, aak, akk, kkr, krk, rka}
klkkklrk	{klk, lkk, kkk, kkl, klr, lrk}
krlkrkl	{krl, rlk, lkr, krk, rkl}
alrla	{alr, lrl, rla}

non-coronal consonant ("k"), and a vowel ("a"), we discuss a worked example that learns this constraint.

As this can be described by the 2-TSL$^{\{l,k,r\}}$ constraint whose forbidden factors are $\{ll, rr\}$, the text must contain all substrings of width 3 or less whose projection to $\{l, k, r\}$ do not contain these prohibited bigrams. The words shown in Table 6.2 would constitute a representative sample for this constraint, though one may notice that some of the 3-factors that need to appear are phonologically implausible.

If we assume that domain boundaries are explicit, then we would also need to encounter any permitted factors that include these boundary symbols. For the sake of brevity such an account has been omitted, but one could easily construct similarly implausible words to account for this change.

6.6 Non-Strict Locality

These methods can be extended beyond just TSL. Chapter 3 demonstrates that the locally testable (McNaughton and Papert, 1971) and locally threshold testable (Beauquier and Pin, 1989) stringsets admit T-relativized variants with the same properties as TSL^{T}:

- A string appears iff its projection to T appears, and
- All symbols that are not in T are freely insertable and deletable.

Locally threshold testable stringsets are characterized by not just the factors in each word but the saturating multisets of factors in the words. A **saturating multiset** is a variant of the multiset in which the counts associated with elements are capped to some maximum value, t. Multisets that saturate at a count of $t = 1$ are simply sets, and these are the characterization of locally testable.

Due to the two properties of relativization, if we collect the saturating multisets of substrings of width bounded above by k along with the individual factors of width $k + 1$, we can still use the algorithm described in section 6.2 to determine salience and again treat the non-relativized grammar in a relativized way. The issue is not finding a learning algorithm; instead it is the space complexity. Figure 6.2 shows the amount of space required for factors, $O(|\Sigma|^k)$, compared to that of saturating multisets, $O\left((t+1)^{|\Sigma|^k}\right)$. Of course, Σ and k are fixed parameters for any given run of the algorithm, and thus constant, but this complexity should not be ignored.

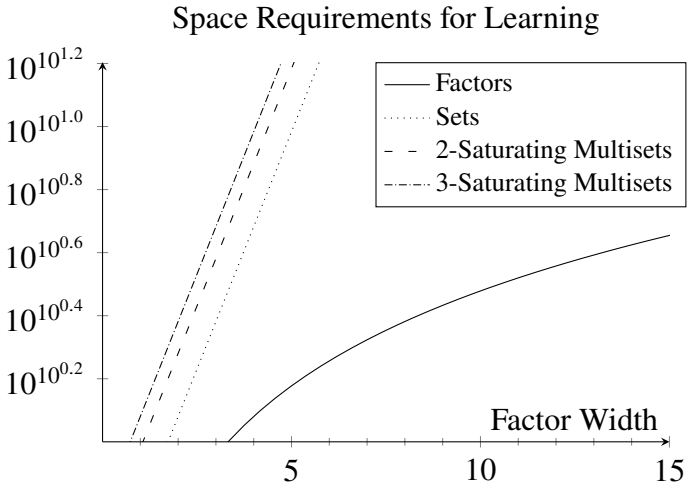


Figure 6.2: While gathering factors requires space exponential in terms of factor width, the requirements are doubly exponential for any of the larger structures we might employ. Here the space requirements are shown for just a binary alphabet.

6.7 Conclusions

We proposed an online learning algorithm for the tier-based strictly k-local class of stringsets that operates in linear time, $O(nk \log |\Sigma|)$, and constant space, $O(|\Sigma|^{k+1})$, in terms of the size of the input. This space complexity is exponential in the factor width. We demonstrated that the grammar representation given by a strictly k-local learner can also be interpreted as a strictly k-piecewise or tier-based strictly $(k-1)$-local grammar. The difference comes later, in the interpretation of the grammar. The algorithms presented here can be incorporated into any sufficiently general implementation of string extension learners (Heinz, 2010b).

Efficient learning of interacting constraints remains an open question. Generally the set of symbols salient to a pattern as a whole will be some superset of those sets for its constraints. If a stringset L consists of a TSL component with an additional constraint imposed that restricts the set of substrings that may occur, then L will not in general be learnable by the known TSL-learning algorithms including those presented here. If multiple TSL constraints over different tier alphabets interact, the

learned stringset will consider the set of salient symbols to be the union of all such alphabets, but other sorts of interactions have yet to be explored. Notably, SL is equivalent to TSL$^{\Sigma}$ by definition, and any SP constraint imposes a tier of salience including the symbols that it mentions, so many cases of interaction will result in a TSL$^{\Sigma}$ (that is, SL) approximation of the target stringset.

This lack of robustness in the face of constraint interaction poses a challenge for the learnability of TSL constraints within a more complex structure in a natural setting. If the solution to this problem is learning a new grammar for each possible tier alphabet, rather than trying to determine which such alphabet to consider, then we must bear in mind the additional space requirements, exponential in the size of the alphabet. Such an approach yields the multiple-tier-based strictly local languages of De Santo and Graf (2019).

Chapter 7: Tree Recognition with Strongly Directed Hypergraphs

Finite-state acceptors over sequences are used for many purposes such as text search (Thompson, 1968) or control flow. They are commonly represented as a directed graph, but such an acceptor can represent only a regular language (Kleene, 1956). Using trees instead of sequences allows for exact classification of a context-free language, such as the Dyck language of balanced brackets, while maintaining a finite amount of state information (Rogers, 1997). Klein and Manning (2004) provide a hypergraph representation of finite-state acceptors over trees, but the resulting structure requires instantiating free indices in the nodes during a parse, which can result in the generation of unboundedly many ground states. This is unnecessarily redundant, generating a state-space which grows along with the input to be checked, mirroring a typical chart-parse.

This chapter introduces a generalization of hypergraphs that map sequences to sequences rather than mapping sets to sets. This can directly encode the transition relation of a tree acceptor, which maps a sequence of states to a single state. The resulting structure is a direct generalization of the graph structure used to represent string acceptors. Graph-based analyses of common decision problems and operations are provided. String acceptors can be reinterpreted unchanged as tree acceptors under this framework. When deciding if a tree is acceptable, the state-space remains constant no matter the size of the input. This can also be useful pedagogically, as in contrast to introductory texts on string acceptors such as that of Hopcroft and Ullman (1979), works regarding tree acceptors such as those of Comon et al. (2007) or Gécseg and Steinby (1984) have no illustrations of the machines.

First we discuss background material in section 7.1. This includes the concepts of finite-state acceptors over sequences and over trees, as well as graph-based representations that have been used

for each. Next, a generalization of the hypergraph is introduced in section 7.2, which will serve as a foundation for a novel representation of tree acceptors. Standard automaton operations including determinization, minimization, completion, trimming, and the Boolean operations are discussed in section 7.3. Finally we conclude with a summary.

7.1 Background

This section provides a brief overview of finite-state acceptors through a structural lens.

7.1.1 Trees

A sequence can be thought of as a linked list, with each position containing a symbol from the alphabet as well as a single connection to the rest of the structure. A tree is similar, except rather than a single connection there is an ordered sequence of connections to smaller structures. Traditionally trees are described in terms of a ranked alphabet, where the alphabet consists not of symbols alone, but of symbol/rank pairs. This bounds the expansion of the width of the tree. If the pair $\langle x, n \rangle$ appears in the alphabet, representing the symbol x with rank n, then a node labeled x is allowed to have exactly n children. A symbol may appear with more than one rank. That is, a tree is a structure built of the symbol x and a sequence of n subtrees.

Unranked tree languages, where there is no limit on the lengths of the sequences of subtrees, are used in some circumstances such as XML parsing (Barrero, 1991; Gelade et al., 2013). As will be discussed later when detailing completion and complementation algorithms, the structures described here can represent both ranked and unranked tree languages.

7.1.2 Finite-State Acceptors

Recall that a string acceptor is definable by a five-tuple $\langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is a finite alphabet, Q is a finite set of states, δ is a finite relation in $\Sigma \times Q \times Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. Recall also that assignment of states may proceed either left-to-right or right-to-left, as regular languages are closed under reversal. When it comes to trees, the situation is similar. States may be assigned either top-down (from the root to the leaves) or bottom-up (from the leaves to the root). If δ may be an arbitrary relation, if the acceptor is nondeterministic, then both orderings are expressively equivalent (Comon et al., 2007). However if δ is required to be a function, if the acceptor is required to be deterministic, the bottom-up ordering is strictly more powerful than the top-down ordering, and indeed as expressive as a nondeterministic acceptor (Comon et al., 2007). Therefore only bottom-up acceptors will be considered here. A deterministic bottom-up finite-state tree acceptor for trees over a finite alphabet Σ can be defined by a finite set Q of states, a transition function $\delta \colon \Sigma \times Q^* \to Q$ whose preimage is finite, and a set $F \subseteq Q$ of accepting states. Note that the rank information can be inferred from the δ function and need not be explicitly encoded as part of Σ, although we will see in section 7.3 that the rank information is needed for completion. No initial state is needed, as the initial states are those reached by leaves (symbols of rank 0). If a node labeled x has children $\langle c_1, \ldots, c_n \rangle$ and each c_i has been assigned a state q_i, that node is assigned the state $\delta(x, \langle q_1, \ldots, q_n \rangle)$. This holds for leaves as well; a leaf labeled x is assigned the state $\delta(x, \varepsilon)$, where ε represents the empty sequence. This can be effectively computed by beginning at the root, descending to the leaves, and bubbling up the state information. Represent a tree by $x[t_1, \ldots, t_n]$, where x is the label of the root node, and each t_i is a child subtree. The state computed for this tree is $\delta^*(x[t_1, \ldots, t_n]) = \delta(x, \langle \delta^*(t_1), \ldots, \delta^*(t_n) \rangle)$. When a leaf is reached, the sequence of children is

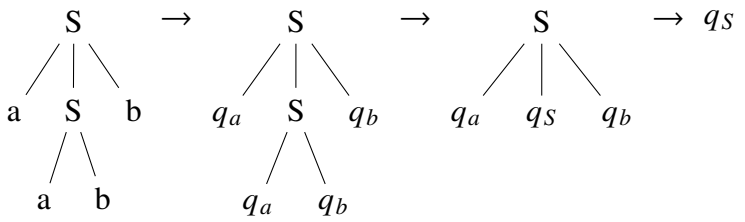


Figure 7.1: Accepting a tree.

empty and the base case is reached: $\delta^*(x[]) = \delta(x[])$.

Given a context-free grammar (CFG) defined by a set N of nonterminal symbols, a set T of terminal symbols, and a relation $R \subseteq N \times (N \cup T)^*$ describing rules, a deterministic bottom-up tree acceptors (DBFTA) representing the corresponding language can be constructed. For each terminal $t \in T$, assign $\delta(t, \varepsilon) = q_t$, and for each rule $n, \langle \sigma_1, \ldots, \sigma_n \rangle$, assign $\delta(n, \langle \sigma_1, \ldots, \sigma_n \rangle) = q_n$. This directly encodes the rules as a DBFTA, and all that remains is to mark the states associated with start symbols as accepting. Consider the CFG whose terminals are "a" and "b", whose sole nonterminal and start symbol is "S", and whose rules are S → ab and S → aSb. Figure 7.1 shows the assignment of states to a particular tree using the DBFTA derived from this CFG.

7.1.3 Directed Graphs and Extensions Thereof

A directed graph (a digraph) consists of a set V of nodes and a set $E \subseteq V \times V$ of edges between them. These edges may be labeled by elements of some finite alphabet Σ, in which case there is a labeling function $\lambda \colon \sigma \to \mathscr{P}(E)$. In other words, a labeled digraph is a collection of overlaid digraphs, sharing nodes but each having its own edges.[1] A finite-state string acceptor is representable as a labeled digraph, where the states are represented as nodes, the transition $\delta(\sigma, q_s) = q_f$ is

[1]Alternatively, a labeled graph may be represented as a heterogeneous 2-colored graph, where one color represents the actual nodes and the other represents the edge labels. The two are equivalent iff edges are allowed to lack sources or sinks.

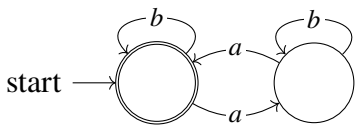


Figure 7.2: A finite-state string acceptor: doubly-outlined states are accepting, and the initial state is marked by "start". All and only those strings which contain an even number of occurrences of a are accepted.

represented by the existence of the edge $\langle q_s, q_f \rangle$ in $\lambda(\sigma)$, and the initial and accepting states are somehow marked as such. Figure 7.2 depicts a string acceptor whose associated set is all and only those strings which contain an even number of occurrences of the symbol 1 that has been discussed above.

If the set of edges were instead $E \subseteq \mathscr{P}(V) \times \mathscr{P}(V)$ then the underlying structure would be a hypergraph. This is not in itself sufficient to represent a tree acceptor, as trees are ordered. Moreover, a single state may inhabit two positions in the source set, which is not representable directly. That said, these hypergraph-structured automata have been used to represent trees by encoding free indices into the state-space (Klein and Manning, 2004). A representation of a tree acceptor as an unlabeled directed hypergraph like those of Klein and Manning (2004) is shown in Figure 7.3.

This representation is unsatisfying for at least two reasons. First, there is no ability to directly interpret a standard finite-state string acceptor as such a tree acceptor, despite the fact that a sequence is just a degenerate tree. Additionally, parsing a particular string instantiates the indices on the states, potentially many times. For instance, the string "aaabbb" will be parsed as follows:

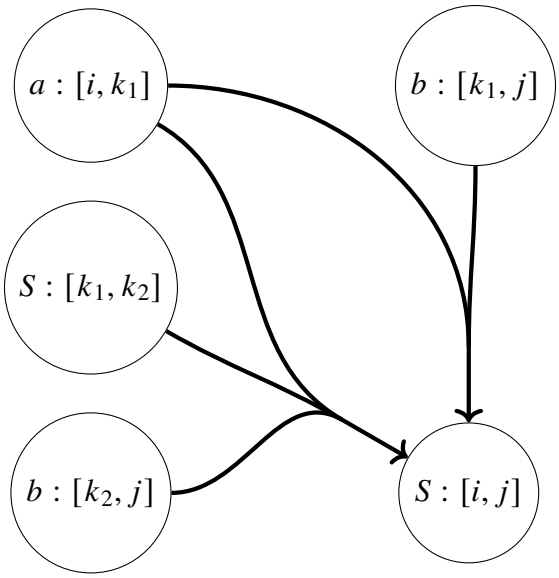
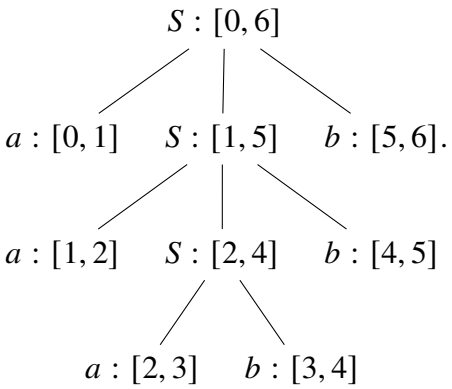


Figure 7.3: S → aSb and S → ab as an unlabeled directed hypergraph. Each rule is represented by a hyperedge.



Each node in this parse tree becomes a state in the derivation, despite the fact that $a : [i, k_1]$ appears only once in the automaton.

7.2 Strongly Directed Hypergraphs

This section introduces a variant of hypergraphs in which edges connect not sets to sets, but sequences to sequences. This modification avoids the need for multiple nodes per symbol, as well as multiple instantiations of nodes during parsing. Rather than having nodes for symbols augmented

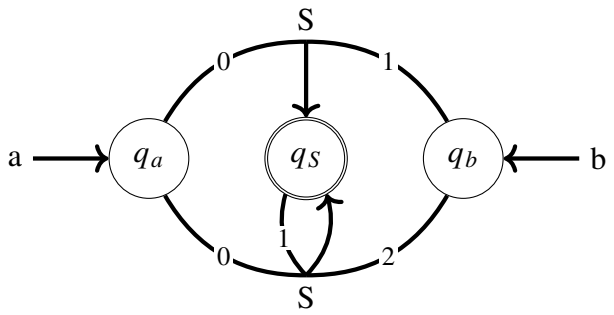


Figure 7.4: S → aSb and S → ab as a labeled strongly directed hypergraph. Each rule is represented by a hyperedge, and accepting states are represented by being doubly outlined.

by free indices, these objects will have one node per state of the acceptor, which for a context-free grammar need not be more than one per symbol.

A strongly directed hypergraph is a structure comprised of a set V of nodes and a set $E \subseteq V^* \times V^*$ of edges. Representing a DBFTA in this form is simple. As in the case of a string acceptor, each state is represented by one and only one node. A transition on symbol x from states $\langle q_1, \ldots, q_n \rangle$ to state q is represented by exactly such a labeled edge. The $\{S \to aSb, S \to ab\}$ tree acceptor is shown again in Figure 7.4, this time as a strongly directed hypergraph. Each edge has a sequence of sources and a sequence of sinks, denoted by numeric indices on the edges. In the case of a tree acceptor, the sequence of sinks is always singleton (they are B-graphs) and the redundant label is omitted.

An alternative graphical representation is inspired by Valdivia et al. (2021). Hypergraphs and their (strongly) directed variants can be represented in tabular format where each state labels a row and each edge labels a column. Accepting states are denoted by their row header being boxed. If a state is a source of an edge, the set of indices at which it appears is placed in the cell (or just some marker, if the sources are unordered). If a state is the sink of an edge, then the corresponding cell is marked. The graphs in question here will never have multiple sinks, so denoting their indices is unnecessary.

	a	b	S	S
q_a			{0}	{0}
q_b			{1}	{2}
q_S				{1}

Figure 7.5: A tabular representation of the DBFTA of Figure 7.4.

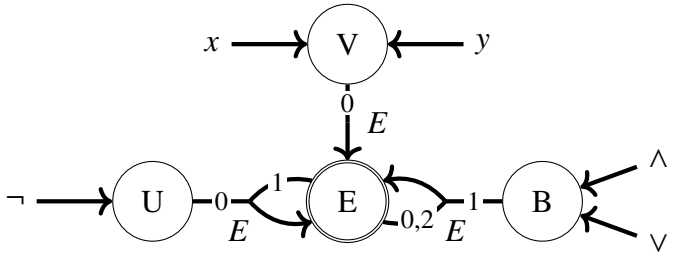


Figure 7.6: A tree acceptor for Boolean expressions over two variables.

Figure 7.5 shows the same automaton as Figure 7.4. This tabular representation may be better for visualization purposes, since there are no layout constraints regarding the crossing of edges.

As another example, the following CFG, representing Boolean expressions over the two variables x and y, is represented in Figure 7.6 and Figure 7.7.

$$N = \{E\}$$

$$T = \{x, y, \neg, \wedge, \vee\}$$

$$R = \{E \to x, E \to y, E \to \neg E, E \to E \wedge E, E \to E \vee E\}$$

$$F = \{E\}$$

If both the source and sink sequences of edges are limited to singletons, this sort of tree acceptor is identical to a string acceptor. The string is then represented such that the rightmost symbol is the

	¬	∧	∨	x	y	E	E	E
U	□					{0}		
B		□	□				{1}	
V				□	□			{0}
E						{1}	{0, 2}	□

Figure 7.7: The tabular representation of Figure 7.6.

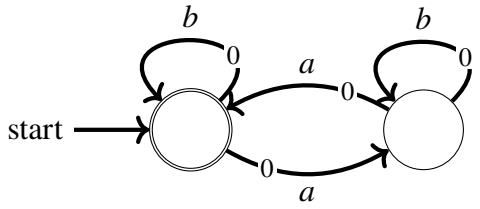


Figure 7.8: The string acceptor of Figure 7.2 as a strongly directed hypergraph. The two are identical, except that this version has additional labels for the sequence indices, which are always zero.

root and preceding symbols are children of their successors. A unique "start" symbol is added to occupy the leaf slot. This accounts for the fact that states are assigned from the bottom to the top.

7.3 Decisions and Operations

This section discusses decision procedures and automaton operations from the graph-based perspective. Concrete implementations are provided for some of the less trivial algorithms. Traditionally, tree automata operate over a ranked alphabet. The rank of a symbol can be inferred from the edges in the graph structure, although one should be careful to retain this information for symbol/rank pairs with no corresponding edges if the transition function is partial. All of these are straightforward generalizations of known algorithms for string acceptors, possibly after first reducing the strongly directed hypergraph to a simpler structure.

7.3.1 Reachability and Satisfiability

One of the questions that one may ask of a grammar is whether it is consistent, if there are any structures that satisfy it. The view of a tree acceptor as a strongly directed hypergraph yields an efficient test for finite-satisfiability of a given tree language L: there exists a tree in L iff there is some accepting state that is reachable. Reachability needs little information: all label information may be discarded, including both the sequence indices and the symbol labels. Then, a dynamic programming algorithm for deciding this question is as follows. If a node is the sink of an edge which has no sources, then that node is reachable. Once these are accounted for, one can iterate considering the following: if a node is the sink of an edge which has only reachable sources, then that node is reachable. Eventually, no updates will occur to the set of known-reachable nodes. At that point, everything not yet known to be reachable is unreachable. If any accepting state is reachable, then the DBFTA is satisfiable by a finite tree.

A DBFTA is reduced iff all of its states are reachable. To reduce a nonreduced DBFTA, simply remove all unreachable states as well as the edges to which they connect. Satisfiability is checking that the returned set of final states is nonempty.

7.3.2 Determinization

Sometimes one may be presented with a graph that represents not a deterministic function, but a nondeterministic relation. Because each edge has exactly one sink, the Rabin-Scott powerset construction (Rabin and Scott, 1959) applies to these tree automata in much the same way as to more typical string acceptors. Given a tree automaton with stateset Q and transition relation R,

proceed as follows. For each symbol x which labels an edge with n sources[2] and for each possible sequence a of length n over $\mathscr{P}(Q)$, construct an edge as follows:

$$\delta(x, \langle a_1, \ldots, a_n \rangle) = \{q : (x, \langle q_1, \ldots, q_n \rangle, q) \in R \text{ and } q_i \in a_i\}.$$

Rather than constructing the entire powerset of states however, it is best to begin with the leaves and construct their corresponding states, then iteratively add edges considering only the states that exist so far. Eventually, no new states will be added, and a deterministic equivalent will have been constructed.

7.3.3 Minimization

Like Comon et al. (2007), this section describes DBFTA minimization in terms of the Myhill-Nerode theorem for trees. The following algorithm operates only on reduced deterministic automata. Similar to the string case, a table must be constructed with one fewer row and column than there are states. The table will be filled with partial derivations which distinguish the states. A first approximation of the partition, which will later be refined, says only that accepting states are distinct from rejecting states. Consider the DBFTA of Figure 7.9. It has four states, so the minimization table should have three rows and columns:

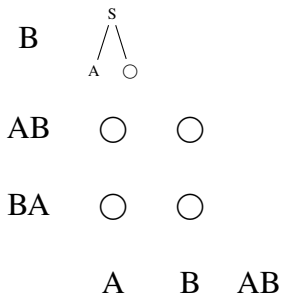
B			
AB	○	○	
BA	○	○	
	A	B	AB

[2] In other words, for each symbol x of rank n.

	a	b	S	S	S	S	S	S
A	□		0	1	0	0	2	2
B		□	1	0	2	2	0	0
□AB□			□		□1□	□	1	
□BA□				□		1	□	□1□

Figure 7.9: A nonminimal DBFTA. Set braces are omitted.

In this example, only two pairs of states remain undistinguished at this point. Next, to determine if states p and q are distinct, consider each symbol x that labels an edge with n sources. For each sequence a of length $n-1$ and for each $0 \leqslant i < n$, consider the outcome of the transition $\delta(x, \langle a_0, \ldots, a_{i-1}, p, a_i, \ldots, a_{n-1}\rangle)$ and that of the transition $\delta(x, \langle a_0, \ldots, a_{i-1}, q, a_i, \ldots, a_{n-1}\rangle)$. If the resulting states are known to be distinguishable, then p and q are also distinguishable. Here we see that states A and B are distinguished by $x = \mathrm{S}$ and $a = \langle \mathrm{A} \rangle$. Specifically $\delta(\mathrm{S}, \langle \mathrm{A}, \mathrm{A}\rangle)$ is undefined (and thus rejecting) while $\delta(\mathrm{S}, \langle \mathrm{A}, \mathrm{B}\rangle) = \mathrm{AB}$ (and accepting). No such sequence yet suffices to distinguish AB from BA, however:



After another iteration, nothing changes. The states AB and BA are still not distinguishable, and no further updates can be made. All distinguished pairs are now known, and the partial trees stored in the table can help in constructing the distinguishing examples. If there is no need to find such an example, it suffices to simply store a mark in the cells representing distinguishable pairs.

	a	b	S	S	S	S
[A]	□		0	1	0	2
[B]		□	1	0	2	0
[AB]			□	□	1	1

Figure 7.10: A minimal form of Figure 7.9.

Minimization involves merging these indistinguishable states. For each state q_i, construct a state $[q_i]$ labeled by the set of states equivalent to q_i itself. Then for each edge, replace each state q in its sequence of sources and of sinks by its equivalence class. In this example, $[\mathrm{AB}] = [\mathrm{BA}] = \{\mathrm{AB}, \mathrm{BA}\}$, and all other classes are singleton. The minimal acceptor then is shown in Figure 7.10.

7.3.4 Completion and Trimming

A DBFTA is complete iff every n-source edge is attached to every possible sequence of n edges. A DBFTA is trim iff every state has some usable path to an accepting state. A path is usable iff every source of every edge along the path is reachable. To complete an automaton whose stateset is Q, find all symbols x that label edges of n symbols, and find all sequences a of length n over $Q \cup \{\bot\}$, where $\bot$ is some new state. Construct a new transition function

$$\delta'(x, a) = \begin{cases} \delta(x, a) & \text{if it is defined,} \\ \bot & \text{otherwise.} \end{cases}$$

If the automaton was already complete, then $\bot$ will be unreachable. Reduce the resulting automaton so that $\bot$ is not inserted unnecessarily. One may note that this only completes the automaton with respect to the supplied ranked alphabet. Several trees still have no path in the resulting automaton. In the case of string acceptors, one may work around this by adding edges labeled by some special

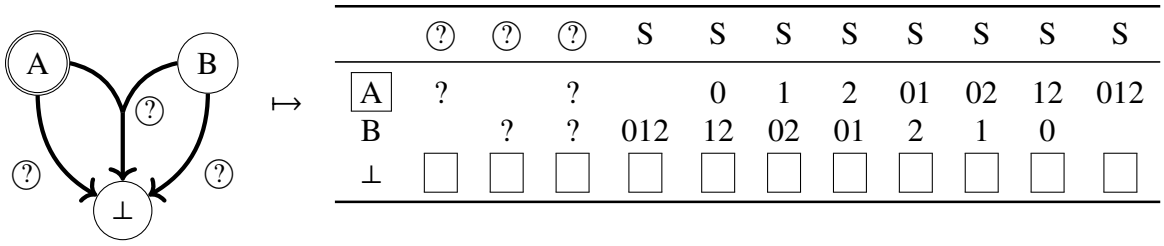


Figure 7.11: Instantiating ? for a rank-3 S.

symbol, ?, which represents all symbols not in the alphabet (Hulden, 2009; Lambert and Rogers, 2020). The same applies in the case of tree acceptors, but there are significantly more of them. In order to account for universal completion, an edge on ? should be attached from every set of states to $\perp$, including the empty set. Semantically adding a symbol/rank pair $\langle x, r \rangle$ to the alphabet requires duplicating and instantiating all such edges whose source sets have size bounded below by 1 and above by r to every possible surjective map from the first r natural numbers onto this source set. An example is shown in Figure 7.11.

The simplest way to trim a DBFTA is to first minimize and reduce it. All states lacking a path to an accepting state will be merged into a single state, as nothing can distinguish them. If a state lacks a path to an accepting state, then all of its usable out-edges leads to another such state. Because the DBFTA is minimal, that means these are self-loops. Thus trimming a minimal, reduced DBFTA involves inspecting the rejecting states in search of one where all it is the sink of all its out-edges. Then remove that state and all edges attached to it.

7.3.5 Finiteness

The language of a reduced, trimmed automaton is finite iff there are no cycles. Similar to the test for emptiness, the finiteness decision problem can be checked using a structure with reduced information capacity. Only a standard digraph is needed. Given the strongly directed hypergraph

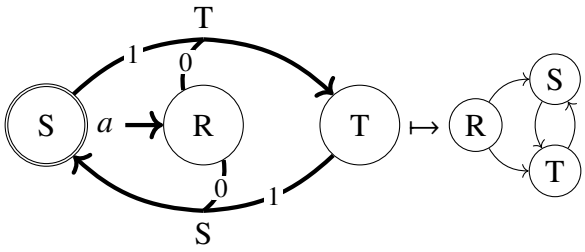


Figure 7.12: A DBFTA and its associated connection graph.

representation of a reduced, trimmed DBFTA, construct a connection graph as follows. Remove all state parity information. Replace each hyperedge of n sources with n edges, connecting each source to the sink. The resulting graph has cycles iff the original structure did as well. Reducedness is required to guarantee that all edges are traversable, and trimness removes the rejecting sink and its associated loops. Figure 7.12 shows a DBFTA and its connection graph, each of which has a clearly visible cycle of length two. The advantage of using the connection graph is that cycle search in graphs is already commonly implemented.

7.3.6 Boolean Operations

The complement of a complete DBFTA is found by swapping the parity of each state. All rejecting states become accepting, and all accepting states become rejecting. Note that this is really only a complement relative to the set of all trees over the specified ranked alphabet, and that there exist trees that are in neither the represented tree language nor its complement due to containing symbol/rank pairs outside of this alphabet. The same caveat applies to string acceptors and the same workaround is useful: completion with ⑦ edges. Barrero (1991) proves that tree acceptors cannot represent the complement of an unranked tree language, but the ⑦ edges essentially act as infinitely many edges, bypassing this restriction.

The union of two tree automata is even simpler than the complement. The simplest construction is

the same as for string acceptors. If each is represented by a graph, then both graphs together, with no connections at all between them, suffice to give a nondeterministic representation of their union. If either has edges on ⍰, they should be instantiated as necessary such that both graphs operate over the same ranked alphabet. The resulting disconnected graph can be determinized, minimized, and trimmed if desired, but as noted by Heinz and Rogers (2013) it may often be preferable to leave Boolean combinations in this factored state. The simplest implementation of intersection involves application of De Morgan's laws: $A \cap B = \complement(\complement A \cup \complement B)$. Unfortunately, complementation requires determinism, but the factored union representation is nondeterministic. So there is no intrinsic factored representation of intersections.

Alternatively, the union and intersection can be represented by the product construction. Provided are two tree acceptors $\mathcal{A}_i = \langle \Sigma_i, Q_i, \delta_i, F_i \rangle$ for $i \in \{1, 2\}$. For each symbol x of rank zero, construct a state $\langle q_a, q_b \rangle$, where $q_a = \delta_1(x, \varepsilon)$ and $q_b = \delta_2(x, \varepsilon)$. Construct an edge from ε to $\langle q_a, q_b \rangle$. Then iterate as follows. Let Q represent the set of states generated so far. Then for each symbol/rank pair $\langle x, r \rangle$ and for each sequence s of length r over Q, let $\langle u_i, v_i \rangle = s_i$ and construct a state $\langle q_a, q_b \rangle$ where $q_a = \delta_1(x, u)$ and $q_b = \delta_b(x, v)$. Construct an edge from s to $\langle q_a, q_b \rangle$, and repeat. Eventually no new states will have been created, and the necessary portion of the product is generated. For the union, the accepting states are those where either component of the labeling pair is accepting in its associated acceptor. For the intersection, both components must be accepting. Continuing a recurring theme, this is the same algorithm as for string acceptors, except that it has to account for all possible sequences of sources rather than considering only a single state at a time.

7.4 Conclusions

This chapter introduced a novel generalization of hypergraphs which allows a direct encoding of a finite-state tree acceptor in a graph-like structure. Decision problems and Boolean operations are interpreted with respect to these structures, in some cases by reducing them to simpler structures. The meaning of complementation was discussed, and a solution implemented to guarantee that all possible trees exist are accepted by either a DBFTA or its complement. The framework allows for reinterpretation of finite-state string acceptors as tree acceptors without change. As many classes of string languages are characterized by graph-theoretic properties (Caron, 2000), this new representation as well as the simpler forms discussed throughout the chapter may pave the way for effectively characterizing tree languages by the same kinds of mechanisms.

Chapter 8: Accumulators and the Problems They Bring

Some have expressed interest in describing certain phenomena using semirings (Lothaire, 2005). These are essentially finite-state acceptors augmented with a monoidal accumulator. This chapter discusses the expressive power of such a structure. First I show that a CKY-style chart parse can be expressed as a monoid. Then I generalize, demonstrating that a run of a Turing machine can also be expressed as a monoid. In order to maintain a truly finite amount of state, one may demand the use of a finite monoid, but this prohibits representing even the identity function. A fundamental question remains: what kinds of restrictions should a monoidal accumulator satisfy? Essentially, how much power are we adding by using this technique, and how much less learnable are the resulting constructions?

8.1 Parsing as a Monoid

Originally published by Ichirō Sakai (Sakai, 1962) and later rediscovered by John Cocke (Hays, 1962), Tadao Kasami (Kasami, 1966), and Daniel Younger (Younger, 1967), the CKY algorithm (named for the latter trio) is a cubic-time bottom-up dynamic programming parser for context-free grammars. This section describes the algorithm in terms of a monoid.

A **context-free grammar** (a type 2 phrase-structure grammar) is a system of rewrite rules, consisting of a set of terminal symbols (words), nonterminal symbols, and rules that transform nonterminal symbols into a sequence of zero or more symbols of either variety (Chomsky, 1956, 1959). They are called "context-free" because a type 1 ("context-sensitive") grammar allows rules to apply only in given contexts.

For notation here, nonterminal symbols are represented by (sequences of) capital letters and terminals by anything else. A rule is written $A \to w$, where A is the nonterminal being rewritten and w is the output sequence.

Consider the grammar with terminals '(' and ')', nonterminal S, and rules S $\to$ (), S $\to$ SS, and S $\to$ (S). It generates the Dyck language of balanced parentheses such as "()", "()()", "(())", etc. This is equivalent (in terms of string yield) to a grammar with the same terminals but a larger set of nonterminals, $\{S, T, L, R\}$, and the following set of rules: S $\to$ LR, S $\to$ SS, S $\to$ LT, T $\to$ SR, L $\to$ (, and R $\to$). Figure 8.1 shows some sample parses with this latter grammar. It has a few features that make processing easier.

This second grammar is in a special form known as **Chomsky Normal Form**, where a nonterminal rewrites to either a single terminal or a sequence of exactly two nonterminals. Any context-free grammar can be written in this form. This arrangement lends itself nicely to a divide-and-conquer style of algorithm. The grammar used in subsequent examples is:

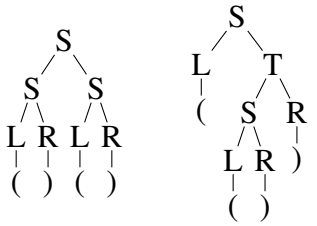


Figure 8.1: Parses for "()()" and "(())".

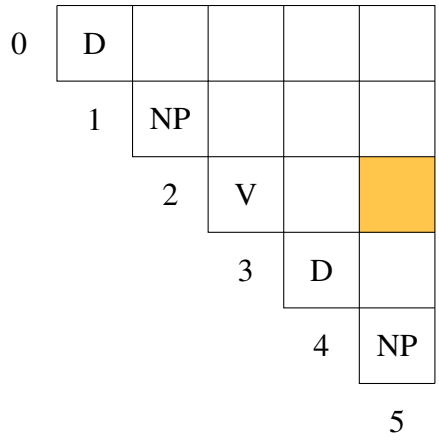


Figure 8.2: Basic shape of the parsing matrix for "$_0$the$_1$girl[$_2$saw$_3$the$_4$crow$_5$]". The cell in row 2, column 5 is highlighted and corresponds to the bracketed region of the input from index 2 to 5.

S $\rightarrow$ DP VP	D $\rightarrow$ the
DP $\rightarrow$ DP PP	NP $\rightarrow$ binoculars
DP $\rightarrow$ D NP	NP $\rightarrow$ crow
PP $\rightarrow$ P DP	NP $\rightarrow$ girl
VP $\rightarrow$ VP PP	P $\rightarrow$ with
VP $\rightarrow$ V DP	V $\rightarrow$ saw

For a sentence of n words, parsing occurs in an upper-triangular n-dimensional matrix whose rows are numbered from zero and whose columns are numbered from one. Concretely, consider the sentence "the girl saw the crow". Indices are placed between words, with 0 at the start of the sentence and n after word n. Figure 8.2 shows the structure of the matrix.

The cells along the main diagonal correspond to single words. If a nonterminal N can rewrite to that word w, then a tree $N \rightarrow w$ is placed in that cell. These have also been filled in (denoted only by N) in Fig. 8.2.

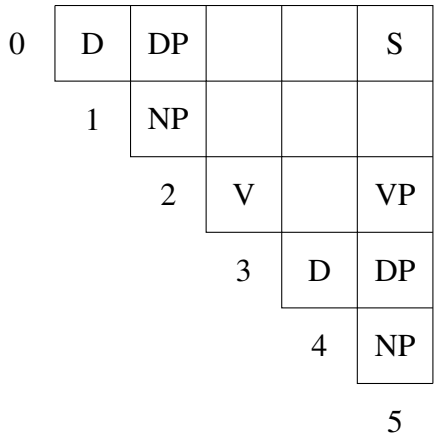


Figure 8.3: Parsed "$_0$the$_1$girl$_2$saw$_3$the$_4$crow$_5$".

For any other cell, the span from i to k that it represents can be split in a number of ways. For all j where $i < j < k$, it can be split into a region spanning i to j and one spanning j to k. The highlighted cell spanning 2 through 5 in Fig. 8.2 can split into 2–3 and 3–5, or into 2–4 and 4–5. For each tree T_1 in the cell corresponding to the left region and for each T_2 in the right region, the roots of T_1 and T_2 will be nonterminals N_1 and N_2. If there is a rule rewriting N to N_1N_2 for some nonterminal N, then a tree $N \rightarrow N_1, N_2$ is placed in that cell. If no such rules exist, then the cell must remain empty. Figure 8.3 shows the entire table filled.

Cells are not limited to a single entry. Some sentences offer more than one valid parse, and the result will be a forest of possible interpretations. Consider the sentence "the girl saw the crow with the binoculars", shown parsed in Figure 8.4. After having parsed the three additional words into their own structure, their matrix can be adjoined diagonally to the earlier structure. Only for the newly formed cells where the rows from the left portion and columns from the right meet, highlighted in the figure, is there work to be done. There are two instances of VP in the cell spanning 2–8, because both the 2–3/3–8 (V DP) split, representing the case where the crow has binoculars, and the 2–5/5–8 (VP PP) split, where the girl has binoculars, are valid interpretations of the sentence. For the same

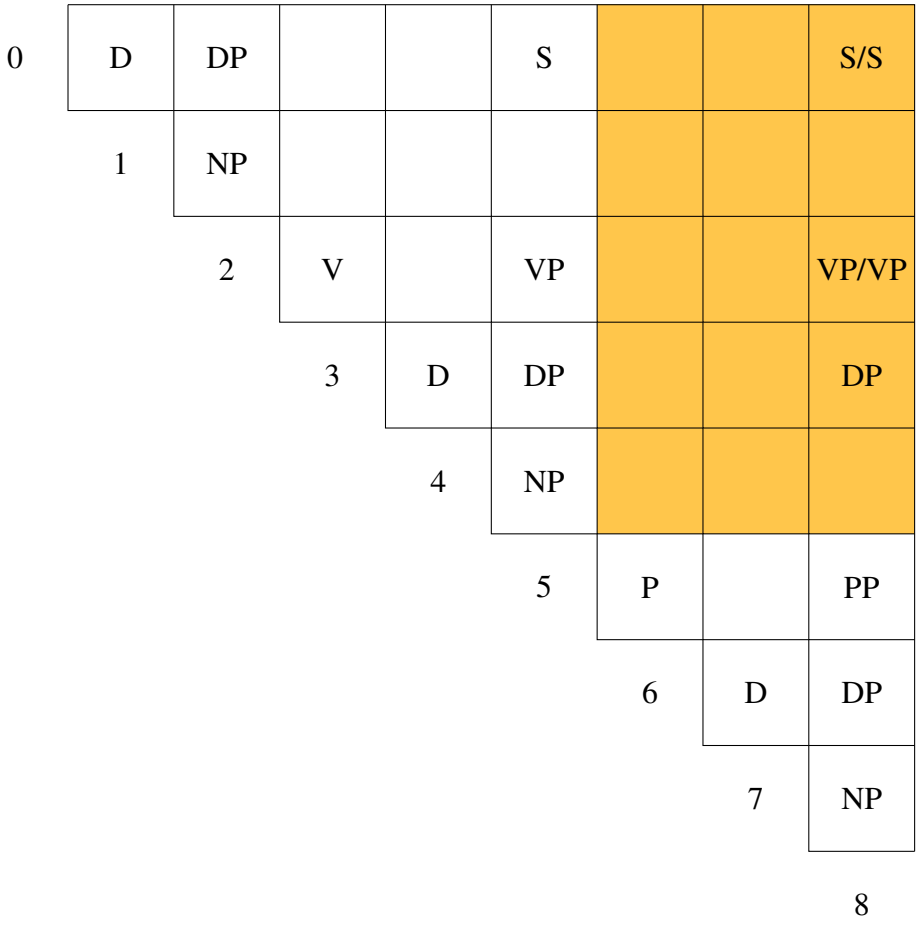


Figure 8.4: Parsed "$_0$the$_1$girl$_2$saw$_3$the$_4$crow$_5$with$_6$the$_7$binoculars$_8$".

reason, there are two instances of S in the upper right cell. Note that the S from 0–5 no longer represents a valid parse of the sentence, as this span does not contain the entirety of the sentence. We can see then that any given CFG has a parsing monoid associated with it: these upper triangular matrices are the elements, and the operation is the act of combining them as was done here.

The preceding portion describes the use of a standard grammar. Weighted or stochastic grammars may be used with only a slight modification. Along the main diagonal, the weight assigned to the tree is exactly the weight of the rule that generated it. In the other cells, the weight is an appropriate combination of the weights of the subtrees with that of the generating rule. This combination might be a product (Smith and Johnson, 2007) or a sum (Katsirelo et al., 2008). The parsing monoid is formed exactly as before, with the elements being the matrices and the operation being the act of diagonally adjoining them then filling in the newly created cells.

A finite-state machine with a single state and a monoidal accumulator is thus sufficient to parse context-free languages. Each input symbol loops back to this same state while outputting a single-cell parse chart, and the accumulator dutifully builds the table as per the monoid operation. One may object that the size of the structure within the accumulator grows quadratically with the size of the input, and this complaint would be well-founded. The infinitely many states needed to decide well-formedness of the context-free string language have been pushed into the monoid.

8.2 Going Further: Turing Completeness

Recall that a monoid is a set S equipped with an associative total function $\lozenge\colon S \times S \to S$, where some element $e \in S$ is an identity for $\lozenge$, i.e. for all $x \in S$, it holds that $e \lozenge x = x = x \lozenge e$. Given a

function $f\colon B \times A \to B$, an initial value b, and a sequence σ of elements of A, the function that computes the following:

$$f(\ldots f(f(b, \sigma_1), \sigma_2) \ldots, \sigma_n)$$

is a left-fold. Here I show that for any such f, a new function $\square$ can be constructed such that $\square$ is a monoid operation.

Suppose we are given a function $f\colon B \times A \to B$. The two types need not be the same, so it is clear that the function need not be associative. Consider new unit types L, N, and R and construct a new aggregate type:

$$C = \big(L \times B\big) \cup N \cup \big(R \times A\big).$$

We can then define a function $g\colon C \to C$:

$$g(x, y) = \begin{cases} \langle L, f(b, a)\rangle & \text{if } x = \langle L, b\rangle,\ y = \langle R, a\rangle \\ y & \text{if } x = N \\ x & \text{otherwise.} \end{cases}$$

Then we have that $g(N, c) = c = g(c, N)$, or in other words N is an identity for g. Then we have a set C and a binary operation over that set with an identity. The only thing missing is a guarantee of associativity. But no matter. Let us finally define yet another function $\square\colon ([C] \times C) \times ([C] \times C) \to$

$([C] \times C)$ as follows:

$$a \,\square\, b = \begin{cases} b & \text{if } a = \langle \varnothing, N \rangle \\ a & \text{if } b = \langle \varnothing, N \rangle \\ \langle x + y, & \\ \quad \text{foldl}(g, N, x + y) \rangle & a = \langle x, c \rangle, \text{b=} \langle y, d \rangle. \end{cases}$$

The pair $\langle \varnothing, N \rangle$ is an identity for $\square$. Then for both of $a \,\square (b \,\square\, c)$ and $(a \,\square\, b) \,\square\, c$, the first element of the resulting tuple (which represents an ordered list of seen elements) is identical, and the other element is computed from that alone. So we have a set $([C] \times C)$ with an associative total binary operation ($\square$) and an identity ($\langle \varnothing, N \rangle$). Thus we have a monoid.

Then an automaton with a single monoid-type accumulator is enough to compute any fold: there is a single state whose start-edge is labeled with a pair of the form $\langle [\langle L, i \rangle], \langle L, i \rangle \rangle$ and all other edges (and the termination output) are labeled with either $\langle \varnothing, N \rangle$ or pairs of the form $\langle [\langle R, x \rangle], \langle R, x \rangle \rangle$. The monoid operation is of course the $\square$ derived from the intended fold function. Computing right-folds instead of left-folds requires only minor modification.

Thus a finite-state machine with a monoidal accumulator and just a single state can express any fold. Just like with parsing, all of the state information is pushed into the monoid. And it should be noted that an infinite monoid is generated in the process which enforces associativity, remembering the entire input as it progresses. A run of a Turing machine can be expressed in this way: collect the input, and run the intended Turing machine over the collected input. Restricting ourselves to finite

monoids certainly reduces the computational power, as the input can no longer be remembered. Indeed, Σ^* is an infinite monoid, so even the identity transformation is no longer representable under such a harsh restriction.

8.3 Conclusions

The semiring-based analysis of transduction by Lothaire (2005) is elegant in its simplicity, but when unrestrained it produces unlimited computational power (Hutton, 1999). A problem arises when trying to appropriately restrict the power. This analysis was designed to unify acceptors and transducers, but a restriction to finite monoids prevents defining even the most trivial of transductions. Perhaps the types of monoids allowed should be all and only those whose operations are computable with a finite-state machine, but this would include these machines, so we have to be careful not to accidentally allow for, say, all primitive recursive functions.

Chapter 9: The Language Toolkit

This dissertation has extended the piecewise-local subregular hierarchy, adding a branch for relativized locality (tier-based classes) as well as incorporating classes characterized by various fragments of formal logics. This chapter in particular describes a software package implementing the tools discussed throughout this work for constructing, classifying, factoring, and learning formal languages. All of the software discussed in this chapter is written in Haskell, and the language toolkit (LTK) core that forms the foundation of it all is a general-purpose library, available at `https://hackage.haskell.org/package/language-toolkit`.

We open with discussion of the PLEB language, used for constructing formal languages based on containment of factors. Next, we look at features of the interactive theorem-prover built upon this library, the PLEB interpreter (`plebby`). This includes visualization, classification, and grammatical inference. The classification algorithms are also available in a standalone program, `classify`. Another program, `factorize`, allows one to automatically extract some types of constraints from provided patterns. Finally we discuss some other companion software.

9.1 Construction: The PLEB Language

The Piecewise-Local Expression Builder, PLEB, is a description format for formal languages. A PLEB file defines a language (a set of strings) by a logical expression, and may also define named sets of symbols or named expressions. The language is powerful enough to define any language that a finite-state automaton can describe, but its design centers around the subset of these languages in which the set of factors that occur in a word is sufficient information to determine whether that word is in the language.

9.1.1 Basic Syntax

Whitespace, ⟨ws⟩ is ignored everywhere except within tokens or where it is explicitly mentioned in the grammar. Comments are treated as whitespace.

⟨comment⟩ := '#' [⟨non-newline⟩ ...] ⟨newline⟩

A file (program) is a non-empty sequence of statements. When using the LTK core function `LTK.Porters.Pleb.readPleb` to evaluate a program in the LTK core, the result is the value of the special variable `it`. This is generally the final bare expression (if any). In the case that `it` has no value, this method of evaluation returns an error. The resulting automaton (if any) is constructed with respect to the alphabet described by the special variable `universe`, which collects all symbols used throughout the file.

⟨program⟩ := ⟨statement⟩ [⟨statement⟩ ...]

A statement is either an expression or an assignment of either a symbol set or an expression.

⟨statement⟩ := ⟨sym-assign⟩ | ⟨exp-assign⟩ | ⟨exp⟩

⟨sym-assign⟩ := '=' ⟨name⟩ ⟨symbols⟩

⟨exp-assign⟩ := '=' ⟨name⟩ ⟨exp⟩

An expression comes in one of three types. It can be an *n*-ary expression, a unary expression, or a

factor. Additionally it can be the name of a defined subexpression.

⟨exp⟩ := ⟨name⟩ | ⟨nary⟩ | ⟨unary⟩ | ⟨factor⟩

⟨n-ary⟩ := ⟨n-op⟩ '{' ⟨exp⟩ [',' ⟨exp⟩ ...] '}'

| ⟨n-op⟩ '(' ⟨exp⟩ [',' ⟨exp⟩ ...] ')'

⟨unary⟩ := ⟨u-op⟩ ⟨exp⟩

Factors are a bit more complicated. Most forms are enclosed in angle brackets (U+27E8 and U+27E9). The basic form of a factor is a sequence of symbol sets separated by ⟨r-op⟩, which can be either ⟨ws⟩ for adjacency, or ',' for (long-distance) precedence. Additionally, a factor can be either free (without anchors) or anchored to one or both of the head or tail of a string. Finally, a factor can be the name of a factor-type variable.

⟨factor⟩ := ⟨name⟩

| [⟨anchors⟩] '⟨' [⟨symbols⟩ [⟨r-op⟩ ⟨symbols⟩ ...]] '⟩'

| '.⟨' ⟨factor⟩ [',' ⟨factor⟩ ...] '⟩'

| '..⟨' ⟨factor⟩ [',' ⟨factor⟩ ...] '⟩'

The first compound kind of factor combines its sub-factors with the adjacency relation, and the other

with the (long-distance) precedence relation. The anchors are denoted as follows:

⟨anchors⟩ := ⟨head-tail⟩ | ⟨head⟩ | ⟨tail⟩

⟨head-tail⟩ := '⋊⋉'

⟨head⟩ := '⋊'

⟨tail⟩ := '⋉'

Note that ⟨head-tail⟩ is a single token; no whitespace may occur within the '⋊⋉' sequence (in particular, not between ⋊ and ⋉).

As described previously, the relation operators, ⟨r-op⟩, in a ⟨factor⟩ can be either whitespace to represent the adjacency relation, or a comma to represent the (long-distance) precedence relation.

⟨r-op⟩ := ⟨ws⟩ | ','

The PLEB language uses the alphabet-agnostic automata of Chapter 3 internally to specify factors, using the ⍰ symbols to preserve the semantics of the construction. For this reason, they are also called **semantic automata**. When combined, the alphabets are semantically extended to be compatible, with missing symbols being placed in parallel to the ⍰ transitions. For this reason, a constraint needs only mention the symbols about which it cares.

Symbol sets are defined by the following syntax:

⟨symbols⟩ := ‘{’ ⟨symbols⟩ [‘,’ ⟨symbols⟩ . . .] ‘}’

| ‘(’ ⟨symbols⟩ [‘,’ ⟨symbols⟩ . . .] ‘)’

| ‘[’ ⟨symbols⟩ [‘,’ ⟨symbols⟩ . . .] ‘]’

| ‘/’ ⟨name⟩

| ⟨name⟩

The first and second of these, using curly braces or parentheses, denote the union of the contained symbol-expressions. The third, using square brackets, indicates an intersection. The fourth, denoted by ‘/’ is a singleton set where the ⟨name⟩ is the symbol itself. And finally, the fifth option is a variable name that must refer to an already-bound symbol set.

A name is a letter as defined by Unicode followed by any sequence of characters other than whitespace or characters from the following set:

, [ ] ( ) { } < > ⟨ ⟩

Note that the ‘#’ character is valid in a name, so a comment must be separated from a name by a space.

9.1.2 N-ary Operators

An n-ary operator can be any of the following. The operators are described in Unicode here, but ASCII equivalents are given in a table at the end of this section.

$\bigwedge$ (U+22C0) The set intersection (logical conjunction) of the operands.

$\bigvee$ (U+22C1) The set union (logical disjunction) of the operands.

•• (U+2219 U+2219) The gapped-concatenation of the operands, equivalent to normal concatenation except that arbitrary strings may be inserted between the operands.

• (U+2219) The concatenation of the operands. Note that nonanchored ends of factor-expressions automatically allow arbitrary strings to occur, so this concatenation may not be what one expects. Notably, •(⟨/a⟩, ⟨/b⟩) is not "words that contain an **ab** substring", but rather "words that contain an **ab** subsequence", as arbitrary text may intervene between the nonanchored "a" and "b" factors. It may be better to use the .⟨. . .⟩ form when concatenating factors in this way.

\\ Left-quotients of the operands, associating to the left. The left-quotient $A\backslash B$ is the set of strings that can be appended to strings in A to get a string in B, a generalization of the Brzozowski derivative.

// Right-quotients of the operands, associating to the right. The right-quotient B/A is the set of strings that can be prepended to strings in A to get a string in B.

9.1.3 Unary Operators

This section describes the unary relations just as the previous described the n-ary ones.

¬ (U+00AC) The logical negation of the operand. Equivalent to its set complement.

∗ (U+2217) The iteration closure of the operand, sometimes called the Kleene closure. Allow it to occur zero or more times. This has the same caveat as the concatenation operator: factor-expressions that are not fully anchored may have arbitrary strings at the non-anchored ends.

↓ (U+2193) The downward closure of the operand. Accept all and only those strings that can be formed by deleting zero or more symbols from a string accepted by the operand.

[] "Tierify":

‘[’ ⟨symbols⟩ [‘,’ ⟨symbols⟩ . . .] ‘]’

The symbols defined by the ⟨symbols⟩ components specify a tier on which the operand should occur. This returns the preprojection of the operand: the largest language that when projected to the given tier is equal to the operand.

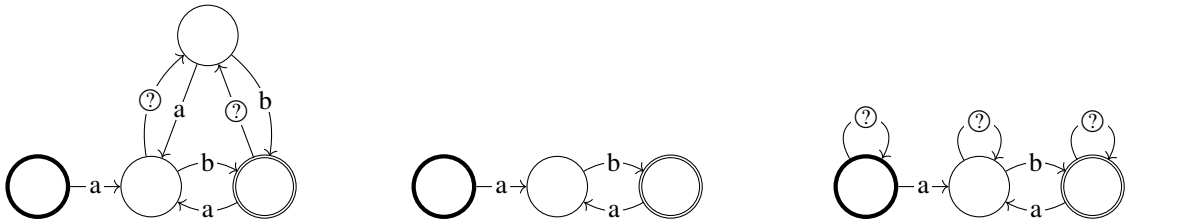
Table 9.1: Equivalent ASCII syntax for PLEB.

Unicode	ASCII
⋊	%|
⋉	|%
⟨	<
⟩	>
⋀	/\
⋁	\/
••	@@
•	@
¬	! or ~
∗	*
↓	$

9.1.4 Remarks

The PLEB language contains a mechanism for describing words. For example, ⋊⋉⟨/a /b⟩ is the single word "ab". It also allows finite unions with ⋁, concatenations with •, and the Kleene star with ∗. These are the defining operations on regular languages, and therefore any regular language is expressible as a piecewise-local expression. Further because expressions are internally represented as finite automata, every piecewise-local expression represents a regular language. The two are expressively equivalent. Provided in the software distribution is a major mode for the GNU Emacs text editor that covers syntax highlighting and entry of the Unicode operators. However, ASCII equivalents are available and described in Table 9.1.

The fact that automata preserve their semantics is meaningful. Consider the language defined over $\Sigma = \{a, b\}$ where words begin on "a", end on "b", and "a" alternates with "b". Figure 9.1 shows three different ways of defining this. If this were a constraint embedded into a larger alphabet, any symbols other than "a" or "b" would follow the ⍰ transitions.



$\bigwedge\{\rtimes\langle a\rangle, \ltimes\langle b\rangle, \neg\langle a, a\rangle, \neg\langle b, b\rangle\}$ $\quad *\rtimes\ltimes\langle a\ b\rangle \quad$ $[a, b]\ \bigwedge\{\rtimes\langle a\rangle, \ltimes\langle b\rangle, \neg\langle a, a\rangle, \neg\langle b, b\rangle\}$

Figure 9.1: Three definitions for the same language over $\Sigma = \{a, b\}$ and their semantics.

9.2 Interacting with the Interpreter

The Piecewise-Local Expression Builder Interpreter, `plebby`, is an interactive theorem-prover for subregular logics. It reads and evaluates single-line PLEB expressions and provides various commands that one can use to explore and interact with languages. A bare expression is automatically assigned to the special expression-variable `it`, and used symbols are automatically accumulated in the special symbol-variable `universe`. All interpreter-commands begin with a colon, and must be the very first thing on the line.

9.2.1 Interpreter Basics

- :quit

  Exit the interpreter. On a UNIX system, Ctrl+D does the same thing.

- :help

  Print an alphabetical list of commands and descriptions.

9.2.2 Saving and Loading

- :savestate ⟨file⟩

  Write the current state to ⟨file⟩.

- :writeATT ⟨file⟩ ⟨file⟩ ⟨file⟩ ⟨exp⟩

  Write an AT&T tabular format automaton representing ⟨exp⟩ to the three ⟨file⟩ arguments, which represent the transitions, input symbols, and output symbols, respectively. If _ is given as the name of a symbol table, then that table is not written. The input and output tables are identical, so writing both is redundant.

- :write ⟨file⟩ ⟨exp⟩

  Write a binary form of ⟨exp⟩ to ⟨file⟩.

- :loadstate ⟨file⟩

  Restore a state previously written by :savestate from ⟨file⟩.

- :readATT ⟨file⟩ ⟨file⟩ ⟨file⟩

  Read an AT&T tabular format transducer from the three ⟨file⟩ arguments (transitions, input symbols, and output symbols), and store the input-projection of the result in the special variable `it`. If _ is given as the name of a symbol table, then no file is read for that table.

- :readATTO ⟨file⟩ ⟨file⟩ ⟨file⟩

  Equivalent to :readATT except that it is the output-projection that is stored.

- :readBin ⟨file⟩

  Read a binary format expression from ⟨file⟩ and store the result in the special variable `it`.

- :readJeff ⟨file⟩

  Read an automaton file in Jeff format from ⟨file⟩ and store the result in the special variable `it`. This will never result in an alphabet-agnostic constraint automaton. In other words, adding to the alphabet has no effect on whether a word is accepted by the resulting automaton, because it is always in ground form.

- :read ⟨file⟩

  Read ⟨file⟩ as a PLEB program, assimilating all bindings. If there are any bare expressions, the last one is assigned to the special variable `it`.

- :import ⟨file⟩

  Read ⟨file⟩ one line at a time as if its contents had been typed into the interpreter. Specifically this means that each statement must be entirely on one line, and some lines may be interpreter commands (including :import itself).

9.2.3 Determining the Class of an Expression

The commands in this section determine whether a given expression is in the corresponding class with respect to the current `universe`. Symbols used in the expression and symbols in the special variable `universe` are the only ones considered in making this decision. The list is given in alphabetic order, along with a brief description of what the classs is and what other names, if any, it has. Most of these classes are discussed in full detail in Chapter 4. Figure 9.2 provides a summary of these classes, showing the containment relations between them. This is nearly all of the classes of Figure 4.16, augmented with other classes from the traditional piecewise-local subregular hierarchy

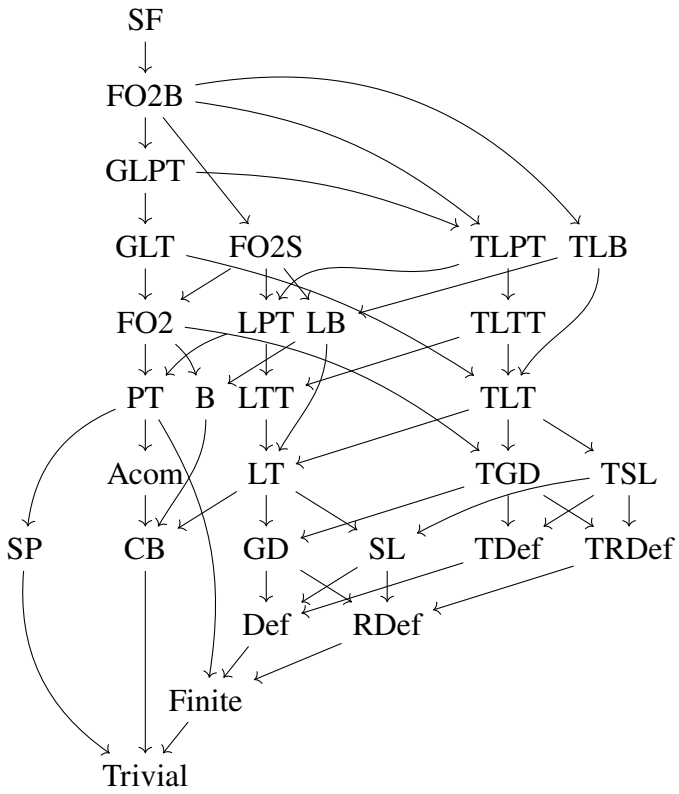


Figure 9.2: Hierarchy of classes for which decision algorithms are provided.

which are not complement-closed and thus not characterized by their syntactic monoids alone, like SL, SP, and TSL.

- :isAcom ⟨exp⟩

  Languages whose syntactic monoids are aperiodic and commutative, equivalent to $\langle 1, t \rangle$-LTT (Almeida, 1989).

- :isB ⟨exp⟩

  Languages whose syntactic monoids are bands (everywhere idempotent).

- :isCB ⟨exp⟩

  Languages whose syntactic monoids are idempotent and commutative, equivalent to 1-LT

(Brzozowski and Simon, 1973; McNaughton, 1974).

- :isDef ⟨exp⟩

  Languages characterized by Boolean combinations of permitted suffixes (Perles et al., 1963; Ginzburg, 1966; Brzozowski and Fich, 1984).

- :isFinite ⟨exp⟩

  Finite languages. To test if a language is cofinite, simply test its complement.

- :isFO2 ⟨exp⟩

  Languages definable in first-order logic with general precedence restricted to two variables, $\text{FO}^2[<]$ (Thérien and Wilke, 1998). See Chapter 4 for a proof that these are equivalent to the $\mathscr{G}$-trivial languages discussed by Brzozowski and Fich (1984).

- :isFO2B ⟨exp⟩

  Languages definable in first-order logic with general precedence and per-symbol betweenness operations, $\text{FO}^2[<, \text{bet}]$ (Krebs et al., 2020).

- :isFO2S ⟨exp⟩

  Languages definable in first-order logic with general precedence and successor, $\text{FO}^2[<, \lhd]$ (Krebs et al., 2020).

- :isGD ⟨exp⟩

  Generalized definite, definable by Boolean combinations over prefixes and suffixes (Ginzburg, 1966; Brzozowski and Fich, 1984).

- :isGLPT ⟨exp⟩

  Generalized locally $\mathscr{J}$-trivial. Generalized by the same means as (Brzozowski and Fich, 1984) applied to LT to obtain GLT: $eM_e e$ is $\mathscr{J}$-trivial in the syntactic monoid for all idempotents e.

- :isGLT ⟨exp⟩

  Generalized locally testable in the sense of (Brzozowski and Fich, 1984); $eM_e e = e$ in the syntactic monoid for all idempotents e. Specifically this is not generalized locally testable in the sense of (Thomas, 1982), which referred to what we now call locally threshold-testable.

- :isLB ⟨exp⟩

  Locally a band, all subsemigroups are idempotent.

- :isLPT ⟨exp⟩

  Locally $\mathscr{J}$-trivial. Related to LT and PT, and also a superclass of dot-depth one (Knast, 1983).

- :isLT ⟨exp⟩

  Locally testable, defined by Boolean expressions over local factors (McNaughton and Papert, 1971).

- :isLTT ⟨exp⟩

  Locally threshold-testable, named as such by Beauquier and Pin (1989) and proven to be equivalent to first-order logic with successor by Thomas (1982).

- :isPT ⟨exp⟩

  Piecewise testable, defined by Boolean expressions over piecewise factors (Simon, 1975). Equivalently, languages whose monoids are $\mathscr{J}$-trivial (Simon, 1975).

- :isRDef ⟨exp⟩

  Reverse definite, defined by Boolean combinations of permitted prefixes (Perles et al., 1963; Ginzburg, 1966; Brzozowski and Fich, 1984).

- :isSF ⟨exp⟩

  Star-free, equivalent to first-order logic with general precedence (Thomas, 1982) or languages whose syntactic monoids are aperiodic (Schützenberger, 1965).

- :isSL ⟨exp⟩

  Locally testable in the strict sense, also known as strictly local. Languages defined by a set of forbidden local factors (McNaughton and Papert, 1971). To test if a language is coSL, simply test its complement.

- :isSP ⟨exp⟩

  Piecewise testable in the strict sense, also known as strictly piecewise. Languages defined by a set of forbidden piecewise factors (Rogers et al., 2010). To test if a language is coSP, simply test its complement.

- :isTDef ⟨exp⟩

  The preprojection of a definite language.

- :isTGD ⟨exp⟩

  The preprojection of a generalized definite language.

- :isTLB ⟨exp⟩

  The preprojection of a language that is locally a band.

- :isTLPT ⟨exp⟩

  The preprojection of a language that is locally $\mathscr{J}$-trivial.

- :isTLT ⟨exp⟩

  The preprojection of a language that is locally testable.

- :isTLTT ⟨exp⟩

  The preprojection of a language that is locally threshold-testable.

- :isTRDef ⟨exp⟩

  The preprojection of a language that is reverse definite.

- :isTrivial ⟨exp⟩

  Either Σ^* or $\varnothing$.

- :isTSL ⟨exp⟩

  The preprojection of a language that is strictly local. To test if a language is coTSL, simply test its complement.

9.2.4 Grammatical Inference

String-extension learning (Heinz, 2010b; Heinz et al., 2012) provides an efficient mechanism for learning subregular classes. The LTK provides such learners for only three classes: SL, SP, and TSL. In each case, the desired factor width is given as a parameter. The TSL learning is accomplished through the SL-reinterpretation described in Chapter 6. In `plebby` these are exposed as :learnSL ⟨int⟩ ⟨file⟩, and similarly for SP and TSL. In each case, words are read from ⟨file⟩, formatted as one word per line, with spaces between each symbol. An automaton over the desired factor width is given as output. Symbols not in the data will always be rejected.

9.2.5 Comparing Expressions

- :strict-subset ⟨expr⟩ ⟨expr⟩

  Determine whether the first ⟨expr⟩ is a proper subset of the second with respect to the current `universe`.

- :subset ⟨expr⟩ ⟨expr⟩

  Determine whether the first ⟨expr⟩ is a (not necessarily proper) subset of the second with respect to the current `universe`.

- :equal ⟨expr⟩ ⟨expr⟩

  Determine whether the first ⟨expr⟩ is equal to the second with respect to the current `universe`, i.e. each is a subset of the other.

- :implies ⟨expr⟩ ⟨expr⟩

  Equivalent to :subset in every way.

9.2.6 Graphical Output

All commands that display graphical output require both a `dot` and `display` program to be accessible, in a path that the system is configured to search. The `dot` program must be GraphViz-compatible, and `display` should accept a PNG image over the standard input and display it appropriately. ImageMagick, for example, provides a reasonable `display` program.

- :display ⟨expr⟩

  Show a normal-form automaton representation of ⟨expr⟩ graphically.

- :eggbox ⟨expr⟩

  Show the syntactic monoid associated with the language of ⟨expr⟩ as an egg-box diagram.

- :psg ⟨expr⟩

  Show the powerset graph of a normal-form automaton representation of ⟨expr⟩ graphically.

- :synmon ⟨expr⟩

  Show the syntactic monoid associated with the language of ⟨expr⟩ as a Cayley graph.

9.2.7 Generating Dot Files Without Displaying Them

- :dot ⟨expr⟩

  Print the normal-form automaton representation of ⟨expr⟩ as a Dot file.

- :dot-psg ⟨expr⟩

  Print the powerset graph of a normal-form automaton representation of ⟨expr⟩ as a Dot file.

- :synmon ⟨expr⟩

  Print the syntactic monoid associated with the language of ⟨expr⟩ as a Dot file representing its Cayley graph.

9.2.8 Operations on the Environment

- :bindings

  Print a list of currently-bound variables and their bindings. Because expression variables have large representations, these representations are omitted from this listing but can be displayed individually with :show.

- :show ⟨var⟩

  Print the current binding of ⟨var⟩, if any, or a message indicating that it is not bound.

- :unset ⟨var⟩

  Remove any binding for ⟨var⟩ from the current environment.

- :reset

  Eradicate the environment.

- :restore-universe

  Set the special variable `universe` to the symbool set that contains all and only those symbols used in other bindings in the current environment.

- :compile

  Convert all saved expressions into automata, retaining the metadata that allows the expression to be alphabet-agnostic.

- :ground

  Convert all saved expressions into automata, discarding the metadata that allows the expression to be alphabet-agnostic.

- :restrict

  Remove all symbols that are not in the current `universe` from all current bindings. This may result in an empty symbol set. Non-satisfiable factors are uniformly replaced by $\neg\langle\rangle$ for simplicity.

9.2.9 Remarks

Uninterpretable assignments are ignored without warning. This includes uninterpretable bare expressions, which are essentially just assignments to `it`. However, warnings are provided when attempting to :display or otherwise apply interpreter-commands to an ill-formed expression.

The `plebby` interpreter is designed for interactive use, not for batch processing. For classification tasks, the standalone program `classify` is provided. It takes as command-line arguments class names (the CLASS suffix of any of the :isCLASS commands of `plebby`), and reads an automaton over the standard input, outputting to the standard output a summary of containments. The `classify` program accepts a few arguments. The `-e` and `-u` switches control under which situations the exit status is "success" — with `-u`, all classes in question must contain the given pattern, while a

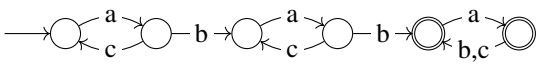


Figure 9.3: A DFA that can be factored.

single class is sufficient in other cases. The `-a`, `-A`, `-j`, and `-p` switches control the format of the input, being the input-projection of an AT&T-format file (the default), the output-projection of an AT&T-format file, a Jeff file, or a PLEB file, respectively. Finally `-n` and `-N` determine whether a non-PLEB automaton is normalized before processing; using `-N` to disable this step is important when classifying transducers using the output of the companion software discussed later.

9.3 Factoring Patterns

A DFA offers an efficient means of deciding whether a word is in a language, using linear time and constant space. For the same reasons that one might wish to define a language via factors using the PLEB language, factoring a finite-state automaton can be quite useful. For example, a factor-based representation might offer insight regarding how similar two patterns are, or which kinds of constraints apply to an entire collection of patterns (Lambert and Rogers, 2019)

The factorization algorithms of Rogers and Lambert (2019b) are implemented in a program called `factorize`. This program takes one or more AT&T-format files as arguments, and for each file outputs a description of the pattern in a `Results` directory, which is created if necessary. Consider the DFA of Figure 9.3. If this is saved as `dfa.att`, then `factorize dfa.att` will create a new file `Results/dfa` whose contents are shown in Figure 9.4.

These output files begin with some metadata. The name of the pattern and the alphabet over which it operates are provided, where the latter is a space-separated sequence of symbols. Next, a line

```
[metadata]
name="dfa"
alphabet=a b c
is_sl="no"
k_sl=3
k_sp=0
k_cosl=0
k_cosp=2

[forbidden substrings]
%|b
%|c
%||%
a a
b b
b c
c b
c c
%|a |%

[forbidden subsequences]

[required substrings (at least one)]

[required subsequences (at least one)]
b b

[nonstrict constraints]
complete="yes"
file=
```

Figure 9.4: The result of factoring Figure 9.3.

describes whether the pattern is strictly local. The remaining `k_`-lines of the metadata section describe the factor widths needed for various components of the pattern: strictly local (forbidden substrings), strictly piecewise (forbidden subsequences), or their complements (disjunctively required factors). In this example, the strictly local component is obtained with a factor width of 3 (it turns out that 2 is sufficient), while there is an additional component which is the complement of a strictly piecewise constraint, using a factor width of 2. The 0-widths indicate that a component of that type is not needed.

Five sections remain, and the first four of them describe components that were automatically extracted. The forbidden substrings form a strictly local superset of the language. Like the alphabet, factors are listed as space-separated sequences of symbols. The factors listed are unanchored by default, but may be prefixed by `%|` or suffixed by `|%` to indicate a head-anchored or tail-anchored factor, respectively. Using both the prefix and the suffix indicates a forbidden word. The forbidden subsequences are the strictly piecewise component, and anchors are never present there. In this case, there are no constraints of this type. Required factors are disjunctive within each section; one of them is required to occur, but not more than that. If both required substrings and required subsequences are present, then one of each category is required. Here, the only positive requirement is that a $\langle \mathrm{b}, \mathrm{b} \rangle$ factor must occur.

The factors in these sections are filtered such that a strictly local is favored over a strictly piecewise component whenever possible. However, the fully-anchored forbidden substrings (the forbidden words) are not filtered by the required factors. Forbidding of $\rtimes\ltimes\langle\rangle$ and of $\rtimes\ltimes\langle \mathrm{a} \rangle$ here is not strictly necessary.

The final component incorporates other classes of constraints which are not automatically extractable. If a file exists at the relative path `Compiled/constraints`, and each line of that file is a path to an AT&T-format file, then the files it references can be used as hypotheses for higher-complexity constraints. If the approximation formed by the first sections is equal to the target language, then no further checks are necessary and the factorization is complete. This is denoted here by the `complete="yes"` line. If it is not, then the other hypotheses are checked, one at a time, until either a match is found or the hypotheses run out. In the first case, the file that created the match is noted and the factorization marked complete. In the other, no file is listed and the factorization is marked incomplete. There is a program, `make-nonstrict-constraints` which populates the current directory with a set of hypotheses that have been useful in factoring the stress patterns in the StressTyp2 database of stress patterns (Goedemans et al., 2015; Rogers and Lambert, 2019b).

9.4 Companion Software

Neither `plebby` nor `classify` can natively handle processes represented as transducers. However, additional software available at `https://github.com/vvulpes0/transducers` provides filters that can convert a transducer to its corresponding transition monoid, which can then be classified using `classify -N`. One must be careful to use the `-N` switch to avoid normalization, in order to avoid accidentally classifying the complexity of the function's preimage instead of the function itself.

The `tmon` filter can currently handle only determinstic finite-state transducers where the inputs are symbols and the outputs are sequences of symbols. Another module, `T2`, is in progress to

accommodate two-way and nondeterministic machines, the former of which will require a different file format.

9.5 Conclusions

This chapter introduced the language toolkit (LTK) and its key components `plebby`, `classify`, and `factorize`, as well as the companion software for dealing with transducers. All of this software is released under the MIT license, with the most recent version available on Github. Most references to the PLEB language have involved its Unicode syntax, but ASCII alternatives are available in Table 9.1.

This software has been used throughout this dissertation to verify class membership of patterns, to verify that classes are distinct, and to arrange visualizations.

Chapter 10: Conclusions

I presented here a structure unifying between string acceptors and tree acceptors in order to improve our understanding of linguistic structure. For the same reasons, I have discussed characterizations and learning algorithms for the tier-based extensions of classes of formal languages in the piecewise-local subregular hierarchy, and additionally situated into this hierarchy some classes based on definability in restrictions of first-order logic. Further, I have discussed the downfalls of using algebraic structures for transducers without regard to their expressive power.

The characterizations of the classes of the piecewise-local subregular hierarchy provide information regarding the properties of the languages they contain. Language-theoretic characterizations allow one to prove that a given formal language lies outside of a class, that the properties may not hold. Of course, such a characterization can also prove inclusion if the entire language is accessible, but a negative answer requires only finitely many words witnessing an exceptions. Model-theoretic results describe the kind of logic required in order to be able to represent a class. These can easily provide a positive guarantee that a language is in the class, but it may be more difficult to show that a language cannot be represented in the given form. Algebraic characterizations based on syntactic semigroups or syntactic monoids can provide both positive and negative inclusion claims. With these in mind, I characterized the tier-based extensions of the classes of the piecewise-local subregular hierarchy language-theoretically, model-theoretically, and algebraically. Because the complements of the strictly local languages had not formerly been characterized language-theoretically, I include those results as well. Having thoroughly explored the tier-based extensions of formal language classes, I go on to provide online learning algorithms for them as well. However, learning of the tier of salient

symbols is a constant plague for these classes. Learning the multiple-tier-based classes can bypass this issue, but is unfeasible.

Beyond the tier-based extensions of subregular classes, several classes of formal languages are known from the computer science literature which have not, to my knowledge, been used with respect to linguistic description. These are the classes based on definability in first-order logic restricted to two variables. I show how these classes fit in with the others in the subregular hierarchy by using the appropriate formalism to define factors over the commonly used relations or by finding languages that fail to satisfy the algebraic characterizations of the classes.

String languages and tree languages have traditionally been treated in different ways. While the algebraic theory of trees has not been completely settled, I do provide a unifying structure for string acceptors and tree acceptors, such that operations on tree languages are simplified and perhaps an algebraic theory might progress via the same route as for strings.

The leading algebraic theory of finite-state functions, that based on semirings from Lothaire (2005), offers a beautiful unification of string acceptors and string transducers. It can define standard acceptors, weighted automata, string-to-string relations both with and without probabilities, and more. So much more. With particular choices of semiring, parsing can be represented with a machine of a single state. In fact, any computable function can be represented in this way. All of the state is pushed into the semiring. But when restricted to finite structures, transformations become impossible and the unifying beauty of the formalism is lost. It is an open question then which semirings make for reasonable accumulators. In avoidance of this question, I explored a more recent mechanism for generalizing algebraic classifications from string acceptors to string-to-string

functions, showing that most if not all phonologically-relevant string-to-string functions have complexity bounded above by the tier-based locally testable class, even if they are not definable with an order-preserving graph transduction in the style of Courcelle (1994). However, functions outside of the subsequential class lack a canonical form, so the true upper bound may be lower for some of these patterns.

In short, I hope that discussion of a structure-based approach to linguistic analysis may encourage others to more freely discuss the variety of ways in which natural language patterns can be described and to consider structural and learnability concerns when proposing new classes of formal languages and transformations for linguistic purposes.

Paths for future exploration are numerous. Mixed relation functions, like the piecewise-locally testable and strictly piecewise-local classes defined by factors involving both adjacency and general precedence, are still largely unexplored. Perhaps there are varieties of monoids or semigroups to which these correspond, or perhaps the algebraic view might expose a similar class worth exploring. Function composition does not preserve algebraic properties, but direct products do, so a mechanism for decomposing functions such that they may be recombined with a direct product might prove useful. Further, syntax relies on trees, and while there are some ideas regarding a subregular hierarchy of tree languages, it is nowhere neare as well-developed as that for strings. And finally, exploring subregular (or, subrational) classes of relations may prove useful in describing phonological or morphological transformations.

Bibliography

Alfred Vaino Aho, Michael Randolph Garey, and Jeffrey David Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. doi: 10.1137/0201008.

Alëna Aksënova and Sanket Deshmukh. Formal restrictions on multiple tiers. In *Proceedings of the Society for Computation in Linguistics*, volume 1, pages 64–73, Salt Lake City, Utah, 2018. doi: 10.7275/R5K64G8S.

Jorge Almeida. Semidirect products of pseudovarieties from the universal algebraist's point of view. *Journal of Pure and Applied Algebra*, 60(2):113–128, October 1989. doi: 10.1016/0022-4049(89)90124-2.

Jorge Almeida. A syntactical proof of locality of DA. *International Journal Algebra and Computation*, 6(2):165–177, 1996. doi: 10.1142/S021819679600009X.

Richard Brian Applegate. *Ineseño Chumash Grammar*. PhD thesis, University of California, Berkeley, 1972.

Alejandro Barrero. Unranked tree languages. *Pattern Recognition*, 24(1):9–18, 1991. doi: 10.1016/0031-3203(91)90112-I.

Danièle Beauquier and Jean-Éric Pin. Factors of words. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming: 16th International Colloquium*, volume 372 of *Lecture Notes in Computer Science*, pages 63–79. Springer Berlin / Heidelberg, 1989. doi: 10.1007/BFb0035752.

Danièle Beauquier and Jean-Éric Pin. Languages and scanners. *Theoretical Computer Science*, 84 (1):3–21, July 1991. doi: 10.1016/0304-3975(91)90258-4.

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

Jean Berstel. Fonctions rationnelles et addition. In M. Blab, editor, *Théorie des Langages, École de printemps d'informatique théorique*, pages 177–183. LITP, 1982.

Mikołaj Bojańczyk. Transducers with origin information. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8–11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 26–37. Springer Berlin / Heidelberg, 2014. doi: 10.1007/978-3-662-43951-7_3.

Mikołaj Bojańczyk, Laure Daviaud, Bruno Guillon, and Vincent Penelle. Which classes of origin graphs are generated by transducers? In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 114:1–114:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ICALP.2017.114.

Véronique Bruyère and Christophe Reutenauer. A proof of Choffrut's theorem on subsequential functions. *Theoretical Computer Science*, 215(1–2):329–335, February 1999. doi: 10.1016/S0304-3975(98)00163-7.

Janusz Antoni Brzozowski and Faith Ellen Fich. On generalized locally testable languages. *Discrete Mathematics*, 50:153–169, 1984. doi: 10.1016/0012-365X(84)90045-1.

Janusz Antoni Brzozowski and Robert Knast. The dot-depth hierarchy of star-free languages is infinite. *Journal of Computer and System Sciences*, 16(1):37–55, February 1978. doi: 10.1016/0022-0000(78)90049-1.

Janusz Antoni Brzozowski and Imre Simon. Characterizations of locally testable events. *Discrete Mathematics*, 4(3):243–271, March 1973. doi: 10.1016/S0012-365X(73)80005-6.

Julius Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6(1–6):66–92, 1960. doi: 10.1002/malq.19600060105.

Phillip Burness and Kevin McMullin. Efficient learning of output tier-based strictly 2-local functions. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 78–90, Toronto, Canada, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-5707.

Pascal Caron. LANGAGE: A Maple package for automaton characterization of regular languages. In Derick Wood and Sheng Yu, editors, *Automata Implementation*, volume 1436 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin / Heidelberg, 1998. doi: 10.1007/BFb0031380.

Pascal Caron. Families of locally testable languages. *Theoretical Computer Science*, 242(1–2):361–376, 2000. doi: 10.1016/S0304-3975(98)00332-6.

Olivier Carton and Luc Dartois. Aperiodic two-way transducers and FO-transductions. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 160–174, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.CSL.2015.160.

Jane Chandlee. *Strictly Local Phonological Processes*. PhD thesis, University of Delaware, 2014. URL `https://chandlee.sites.haverford.edu/wp-content/uploads/2015/05/Chandlee_dissertation_2014.pdf`.

Jane Chandlee and Jeffrey Heinz. Strict locality and phonological maps. *Linguistic Inquiry*, 49(1):23–60, January 2018. doi: 10.1162/ling_a_00265.

Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics*, 2:491–503, November 2014. doi: 10.1162/tacl_a_00198.

Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. Output strictly local functions. In Marco Kuhlmann, Makoto Kanazawa, and Gregory M. Kobele, editors, *Proceedings of the 14th Meeting on the Mathematics of Language*, pages 112–125, Chicago, USA, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/w15-2310.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956. doi: 10.1109/TIT.1956.1056813.

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

Alexander Clark. The syntactic concept lattice: Another algebraic theory of the context-free languages? *Journal of Logic and Computation*, 25(5):1203–1229, October 2015. doi: 10.1093/logcom/ext037.

Robin Clark and Ian Roberts. A computational model of language learnability and language change. *Linguistic Inquiry*, 24(2):299–345, 1993.

Rina S. Cohen and Janusz Antoni Brzozowski. Dot-depth of star-free events. *Journal of Computer and System Sciences*, 5(1):1–16, February 1971. doi: 10.1016/S0022-0000(71)80003-X.

Thomas Colcombet. Green's relations and their use in automata theory. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications: Proceedings of the 5th International Conference, LATA 2011*, volume 6638 of *Theoretical Computer Science and General Issues*, pages 1–21, Heidelberg, 2011. Springer-Verlag. doi: 10.1007/978-3-642-21254-3_1.

Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, October 2007. URL `http://tata.gforge.inria.fr`.

Bruno Courcelle. Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science*, 126(1):53–75, April 1994. doi: 10.1016/0304-3975(94)90268-2.

András Cser. The -alis/-aris allomorphy revisited. In Franz Rainer, Wolfgang Dressler, Dieter Kastovsky, and Hans Christian Luschützky, editors, *Variation and Change in Morphology: Selected Papers from the 13th International Morphology Meeting*, pages 33–52. John Benjamins Publishing Company, Vienna, Austria, 2010. doi: 10.1075/cilt.310.02cse.

Aldo De Luca and Antonio Restivo. A characterization of strictly locally testable languages and its application to subsemigroups of a free semigroup. *Information and Control*, 44(3):300–319, March 1980. doi: 10.1016/S0019-9958(80)90180-1.

Aniello De Santo and Thomas Graf. Structure sensitive tier projection: Applications and formal properties. In Raffaella Bernardi, Greg Kobele, and Sylvain Pogodalla, editors, *Formal Grammar 2019*, volume 11668 of *Lecture Notes in Computer Science*, pages 35–50. Springer Verlag, 2019. doi: 10.1007/978-3-662-59648-7_3.

Hossep Dolatian and Jonathan Rawski. Multi-input strictly local functions for templatic morphology. In *Proceedings of the Society for Computation in Linguistics*, volume 3, pages 282–296, New Orleans, Louisiana, 2020. URL `https://scholarworks.umass.edu/scil/vol3/iss1/28`.

Matt Edlefsen, Dylan Leeman, Nathan Myers, Nathaniel Smith, Molly Visscher, and David Wellcome. Deciding strictly local (SL) languages. In Jon Breitenbucher, editor, *Proceedings of the 2008 Midstates Conference for Undergraduate Research in Computer Science and Mathematics*, pages 66–73, 2008.

Samuel Eilenberg and Marcel-Paul Schützenberger. On pseudovarieties. *Advances in Mathematics*, 19(3):413–418, March 1976. doi: 10.1016/0001-8708(76)90029-3.

Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, January 1961. doi: 10.2307/1993511.

Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. In *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 228–235, July 1997. doi: 10.1109/LICS.1997.614950.

Emmanuel Filiot. Logic-automata connections for transformations. In *Logic and Its Applications*, volume 8923 of *Lecture Notes in Computer Science*, pages 30–57. Springer Berlin / Heidelberg, 2015. doi: 10.1007/978-3-662-45824-2_3.

Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. First-order definability of rational transductions: An algebraic approach. In *LICS '16: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 387–396. Association for Computing Machinery, July 2016. doi: 10.1145/2933575.2934520.

Christiane Frougny and Jacques Sakarovitch. Synchronized rational relations of finite and infinite words. *Theoretical Computer Science*, 108:45–82, 1993. doi: 10.1016/0304-3975(93)90230-Q.

Jie Fu, Jeffrey Heinz, and Herbert G. Tanner. An algebraic characterization of strictly piecewise languages. In Mitsunori Ogihara and Jun Tarui, editors, *Theory and Applications of Models of Computation*, volume 6648 of *Lecture Notes in Computer Science*, pages 252–263. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-20877-5_26.

Pedro Garcia, Enrique Vidal, and José Oncina. Learning locally testable languages in the strict sense. In *Proceedings of the 1st International Workshop on Algorithmic Learning Theory*, pages 325–338, Tokyo, Japan, 1990. URL `https://grfia.dlsi.ua.es/repositori/grfia/pubs/111/alt1990.pdf`.

Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, Hungary, 1984.

Wouter Gelade, Tomasz Idziaszek, Wim Martens, Frank Neven, and Jan Paredaens. Simplifying XML schema: Single-type approximations of regular tree languages. *Journal of Computer and System Sciences*, 79(6):910–936, September 2013. doi: 10.1016/j.jcss.2013.01.009.

Christian Germain and Jean Pallo. Langages rationnels définis avec une concaténation non-associative. *Theoretical Computer Science*, 233(1–2):217–231, February 2000. doi: 10.1016/S0304-3975(98)00043-7.

Abraham Ginzburg. About some properties of definite, reverse-definite and related automata. *IEEE Transactions on Electronic Computers*, EC-15(5):806–810, October 1966. doi: 10.1109/pgec.1966.264264.

R. W. N. Goedemans, Jeffrey Heinz, and Harry van der Hulst. StressTyp2, April 2015. URL http://st2.ullet.net/.

Edward Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, May 1967. doi: 10.1016/S0019-9958(67)91165-5.

Thomas Graf. The power of locality domains in phonology. *Phonology*, 34(2):385–405, 2017. doi: 10.1017/S0952675717000197.

Thomas Graf. Why movement comes for free once you have ajdunction. In Daniel Edmiston, Marina Ermolaeva, Emre Hakgüder, Jackie Lai, Kathryn Montemurro, Brandon Rhodes, Amara Sankhagowit, and Michael Tabatowski, editors, *Proceedings of the Fifty-third Annual Meeting of the Chicago Linguistic Society*, pages 117–136, Chicago, Illinois, 2018. Chicago Linguistic Society.

James Alexander Green. On the structure of semigroups. *Annals of Mathematics*, 54(1):163–172, July 1951. doi: 10.2307/1969317.

Leonard H. Haines. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98, 1969. doi: 10.1016/s0021-9800(69)80111-0.

David Glenn Hays. Automatic language-data processing. In Harold Borko, editor, *Computer Applications in the Behavioral Sciences*, pages 394–423. 1962.

Jeffrey Heinz. Learning long-distance phonotactics. *Linguistic Inquiry*, 41(4):623–661, October 2010a. doi: 10.1162/ling_a_00015.

Jeffrey Heinz. String extension learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 897–906, Uppsala, Sweden, July 2010b. Association for Computational Linguistics. URL https://www.aclanthology.org/P10-1092.

Jeffrey Heinz and Regine Lai. Vowel harmony and subsequentiality. In *Proceedings of the 13th Meeting on the Mathematics of Language*, pages 52–63, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL https://aclanthology.org/W13-3006.

Jeffrey Heinz and James Rogers. Learning subregular classes of languages with factored deterministic automata. In *Proceedings of the 13th Meeting on the Mathematics of Language*, pages 64–71, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL https://www.aclanthology.org/W13-3007.

Jeffrey Heinz, Chetan Rawal, and Herbert G. Tanner. Tier-based strictly local constraints for phonology. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Short Papers*, volume 2, pages 58–64, Portland, Oregon, 2011. Association for Computational Linguistics. URL https://aclanthology.org/P11-2011.

Jeffrey Heinz, Anna Kasprzik, and Timo Kötzing. Learning in the limit with lattice-structured hypothesis spaces. *Theoretical Computer Science*, 457:111–127, October 2012. doi: 10.1016/j.tcs.2012.07.017.

Markus Holzer and Barbara König. On deterministic finite automata and syntactic monoid size. *Theoretical Computer Science*, 327(3):319–347, November 2004. doi: 10.1016/j.tcs.2004.04.010.

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

John M. Howie. *Fundamentals of Semigroup Theory*. Oxford University Press, Oxford, NY, 1995.

Liang Huang and David Chiang. Better k-best parsing. In Harry Bunt and Robert Malouf, editors, *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–65, Vancouver, British Columbia, October 2005. URL https://aclanthology.org/W05-1506.

Mans Hulden. *Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology*. PhD thesis, The University of Arizona, 2009. URL https://hdl.handle.net/10150/196112.

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999. doi: 10.1017/s0956796899003500.

Larry M. Hyman and Francis X. Katamba. A new approach to tone in Luganda. *Language*, 69(1): 34–67, March 1993. doi: 10.2307/416415.

Sanjay Jain, Steffen Lange, and Sandra Zilles. Some natural conditions on incremental learning. *Information and Computation*, 205(11):1671–1684, November 2007. doi: 10.1016/j.ic.2007.06.002.

Adam Jardine. Computationally, tone is different. *Phonology*, 33(2):247–283, August 2016. doi: 10.1017/s0952675716000129.

Adam Jardine and Jeffrey Heinz. Learning tier-based strictly 2-local languages. *Transactions of the Association for Computation in Linguistics*, 4:87–98, 2016. doi: 10.1162/tacl_a_00085.

Adam Jardine and Kevin McMullin. Efficient learning of tier-based strictly k-local languages. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications: 11th International Conference*, volume 10168 of *Lecture Notes in Computer Science*, pages 64–76. Springer, Cham, 2017. doi: 10.1007/978-3-319-53733-7_4.

Jing Ji and Jeffrey Heinz. Input strictly local tree transducers. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications: Proceedings of the 14th International Conference, LATA 2020*, volume 12038 of *Theoretical Computer Science and General Issues*, pages 369–381, Cham, Switzerland, 2020. Springer International Publishing. doi: 10.1007/978-3-030-40608-0_26.

Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. Prentice-Hall, Upper Saddle River, NJ, second edition, 2009.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report R-257, University of Illinois, Urbana, Illinois, March 1966.

George Katsirelo, Nina Naraodytska, and Toby Walsh. The weighted CFG constraint. In Laurent Perron and Michael Alan Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2008*, volume 5015 of *Lecture Notes in Computer Science*, pages 323–327. Springer Berlin / Heidelberg, 2008. doi: 10.1007/978-3-540-68155-7_31.

Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In Claude Elwood Shannon and John McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–42. Princeton University Press, 1956. doi: 10.1515/9781400882618-002.

Dan Klein and Christopher D. Manning. Parsing and hypergraphs. In Harry Bunt, John Carroll, and Giorgio Satta, editors, *New Developments in Parsing Technology*, volume 23 of *Text, Speech and Language Technology*, pages 351–372. Springer Dordrecht, 2004. doi: 10.1007/1-4020-2295-6_18.

Robert Knast. A semigroup characterization of dot-depth one languages. *RAIRO – Informatique théorique*, 17(4):321–330, 1983. doi: 10.1051/ita/1983170403211.

Donald Ervin Knuth, James Hiram Morris, and Vaughan Ronald Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, August 1977. doi: 10.1137/0206024.

Andreas Krebs, Kamal Lodaya, Paritosh K. Pandya, and Howard Straubing. Two-variable logics with some betweenness relations: Expressiveness, satisfiability, and membership. *Logical Methods in Computer Science*, 16(3):1–41, September 2020. doi: 10.23638/LMCS-16(3:16)2020.

Kenneth Krohn, Richard Mateosian, and John Rhodes. Methods of the algebraic theory of machines: Decomposition theorem for generalized machines; Properties preserved under series and parallel compositions of machines. *Journal of Computer and System Sciences*, 1(1):55–85, 1967. doi: 10.1016/S0022-0000(67)80007-2.

Dakotah Lambert. Grammar interpretations and learning TSL online. In *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 81–91, August 2021a. URL `https://proceedings.mlr.press/v153/lambert21a.html`.

Dakotah Lambert. Relativized adjacency. *Journal of Logic, Language and Information*, 2021b. Accepted with minor revisions.

Dakotah Lambert and James Rogers. A logical and computational methodology for exploring systems of phonotactic constraints. In *Proceedings of the Society for Computation in Linguistics*, volume 2, pages 247–256, New York City, New York, 2019. doi: 10.7275/t0dv-9t05.

Dakotah Lambert and James Rogers. Tier-based strictly local stringsets: Perspectives from model and automata theory. In *Proceedings of the Society for Computation in Linguistics*, volume 3, pages 330–337, New Orleans, Louisiana, 2020. doi: 10.7275/2n1j-pj39.

Dakotah Lambert, Jon Rawski, and Jeffrey Heinz. Typology emerges from simplicity in representations and learning. *Journal of Language Modelling*, August 2021a. URL https://jlm.ipipan.waw.pl/index.php/JLM/article/view/262.

Dakotah Lambert, Jonathan Rawski, and Jeffrey Heinz. Typology emerges from simplicity in representations and learning. *Journal of Language Modelling*, 9(1):151–194, August 2021b. doi: 10.15398/jlm.v9i1.262.

Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. Springer Berlin / Heidelberg, 2004. doi: 10.1007/978-3-662-07003-1.

Silvain Lombardy and Jacques Sakarovitch. Sequential? *Theoretical Computer Science*, 356(1–2): 224–244, May 2006. doi: 10.1016/j.tcs.2006.01.028.

M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, New York, 2005.

Connor Mayer and Travis Major. A challenge for tier-based strict locality from Uyghur backness harmony. In Annie Foret, Greg Kobele, and Sylvain Pogodalla, editors, *Formal Grammar 2018*, volume 10950 of *Lecture Notes in Computer Science*, pages 62–83. 2018. doi: 10.1007/978-3-662-57784-4_4.

John J. McCarthy. Feature geometry and dependency: A review. *Phonetica*, 45(2–4):84–108, 1988. doi: 10.1159/000261820.

Adam G. McCollum, Eric Baković, Anna Mai, and Eric Meinhardt. Unbounded circumambient processes in segmental phonology. *Phonology*, 37(2):215–255, August 2020. doi: 10.1017/S095267572000010X.

Kevin McMullin. *Tier-Based Locality in Long-Distance Phonotactics: Learnability and Typology*. PhD thesis, University of British Columbia, 2016.

Robert McNaughton. Algebraic decision procedures for local testability. *Mathematical Systems Theory*, 8(1):60–76, March 1974. doi: 10.1007/bf01761708.

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

Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960. doi: 10.1109/TEC.1960.5221603.

Don Dalzell Miller and Alfred Hoblitzelle Clifford. Regular $\mathcal{D}$-classes in semigroups. *Transactions of the American Mathematical Society*, 82(1):270–280, May 1956. doi: 10.2307/1992989.

Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, June 1997. URL https://aclanthology.org/J97-2003.

Anil Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544. American Mathematical Society, August 1958. doi: 10.1090/s0002-9939-1958-0135681-9.

John von Neumann. On regular rings. *Proceedings of the National Academy of Sciences*, 22(12): 707–713, 1936. doi: 10.1073/pnas.22.12.707.

José Oncina, Pedro García, and Enrique Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):448–458, May 1993. doi: 10.1109/34.211465.

Daniel Nathan Osherson, Michael Stob, and Scott Weinstein. *Systems That Learn*. MIT Press, 1986.

Jean-Pierre Pécuchet. Automates boustrophedon, semi-groupe de Birget et monoïde inversif libre. *RAIRO – Informatique théorique*, 19(1):71–100, 1985. doi: 10.1051/ita/1985190100711.

Micha A. Perles, Michael O. Rabin, and Eliahu Shamir. The theory of definite automata. *IEEE Transactions on Electronic Computers*, 12(3):233–243, June 1963. doi: 10.1109/PGEC.1963.263534.

Jean-Éric Pin. Syntactic semigroups. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, pages 679–746. Springer-Verlag, Berlin, 1997. doi: 10.1007/978-3-642-59136-5_10.

Jean-Éric Pin and Pascal Weil. Polynomial closure and unambiguous product. *Theory of Computing Systems*, 30(4):383–422, August 1997. doi: 10.1007/bf02679467.

Michael Oser Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959. doi: 10.1147/rd.32.0114.

Jonathan Rawski and Hossep Dolatian. Multi-input strictly local functions for tonal phonology. In *Proceedings of the Society for Computation in Linguistics*, volume 3, pages 245–260, New Orleans, Louisiana, 2020. URL `https://scholarworks.umass.edu/scil/vol3/iss1/25`.

James Rogers. A model-theoretic framework for theories of syntax. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 10–16, Santa Cruz, CA, 1996. Association for Computational Linguistics. doi: 10.3115/981863.981865.

James Rogers. Strict LT_2 : regular :: local : recognizable. In Christian Retoré, editor, *Logical Aspects of Computational Linguistics: First International Conference, LACL '96 (Selected Papers)*, volume 1328 of *Lecture Notes in Computer Science*, pages 366–385, Berlin, 1997. Springer-Verlag. doi: 10.1007/BFb0052167.

James Rogers. *A Descriptive Approach to Language-Theoretic Complexity*. (Monograph.) Studies in Logic, Language, and Information. CSLI Publications, 1998.

James Rogers and Dakotah Lambert. Some classes of sets of structures definable without quantifiers. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 63–77, Toronto, Canada, July 2019a. Association for Computational Linguistics. doi: 10.18653/v1/W19-5706.

James Rogers and Dakotah Lambert. Extracting Subregular constraints from Regular stringsets. *Journal of Language Modelling*, 7(2):143–176, September 2019b. doi: 10.15398/jlm.v7i2.209.

James Rogers and Geoffrey K. Pullum. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information*, 20(3):329–342, June 2011. doi: 10.1007/s10849-011-9140-2.

James Rogers, Jeffrey Heinz, Gil Bailey, Matt Edlefsen, Molly Visscher, David Wellcome, and Sean Wibel. On languages piecewise testable in the strict sense. In Christian Ebert, Gerhard Jäger, and Jens Michaelis, editors, *The Mathematics of Language: Revised Selected Papers from the 10th and 11th Biennial Conference on the Mathematics of Language*, volume 6149 of *LNCS/LNAI*, pages 255–265. FoLLI/Springer, 2010. doi: 10.1007/978-3-642-14322-9_19.

James Rogers, Jeff Heinz, Margaret Fero, Jeremy Hurst, Dakotah Lambert, and Sean Wibel. Cognitive and sub-regular complexity. In Glyn Morrill and Mark-Jan Nederhof, editors, *Formal Grammar 2012*, volume 8036 of *Lecture Notes in Computer Science*, pages 90–108. Springer-Verlag, 2012. doi: 10.1007/978-3-642-39998-5_6.

Ichirō Sakai. Syntax in universal translation. In *1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608, London, 1962. Her Majesty's Stationery Office.

Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. doi: 10.1017/CBO9781139195218.

Marcel-Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, April 1965. doi: 10.1016/s0019-9958(65)90108-7.

Marcel-Paul Schützenberger. Sur certaines opérations de fermeture dans les langages rationnels. *Symposia Mathematica*, 15:245–253, 1975.

Imre Simon. Piecewise testable events. In Helmut Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer-Verlag, Berlin, 1975. doi: 10.1007/3-540-07407-4_23.

Noah A. Smith and Mark Johnson. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics*, 33(4):477–491, December 2007. doi: 10.1162/coli.2007.33.4.477.

Magnus Steinby. Rectangular algebras as tree recognizers. *Acta Cybernetica*, 22(2):499–515, January 2015. doi: 10.14232/actacyb.22.2.2015.15.

Price Stiffler, Jr. Extension of the fundamental theorem of finite semigroups. *Advances in Mathematics*, 11(2):159–209, 1973. doi: 10.1016/0001-8708(73)90007-8.

Howard Straubing. Finite semigroup varieties of the form $V * D$. *Journal of Pure and Applied Algebra*, 36:53–94, 1985. doi: 10.1016/0022-4049(85)90062-3.

Pascal Tesson and Denis Thérien. Diamonds are forever: The variety DA. In Gracinda M. S. Gomes, Jean-Éric Pin, and Pedro V. Silva, editors, *Semigroups, Algorithms, Automata and Languages*, pages 475–499. World Scientific, 2002. doi: 10.1142/9789812776884_0021.

Denis Thérien and Thomas Wilke. Over words, two variables are as powerful as powerful as one quantifier alternation: $\mathrm{FO}^2 = \Sigma_2 \cap \Pi_2$. In *STOC '98: Proceedings of the Thirtieth Annual ACM Symbosium on Theory of Computing*, pages 234–240, New York, NY, 1998. Association for Computing Machinery. doi: 10.1145/276698.276749.

Wolfgang Thomas. Classifying regular events in symbolic logic. *Journal of Computer and Systems Sciences*, 25:360–376, 1982. doi: 10.1016/0022-0000(82)90016-2.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968. doi: 10.1145/363347.363387.

Борис Авраамович Трахтенброт. Конечные Автоматы и Логика Одноместных Предикатов. Сибирский Математический Журнал, 3(1):103–131, Февраль 1962.

Paola Valdivia, Paolo Buono, Catherine Plaisant, Nicole Dufournaud, and Jean-Daniel Fekete. Analyzing dynamic hypergraphs with parallel aggregated ordered hypergraph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(1):1–13, January 2021. doi: 10.1109/TVCG.2019.2933196.

Odile Vaysse. Addition molle et fonctions p-locales. *Semigroup Forum*, 34:157–175, December 1986. doi: 10.1007/BF02573160.

Charles Yang. Negative knowledge from positive evidence. *Language*, 91(4):938–953, 2015. doi: 10.1353/lan.2015.0054.

Daniel Haven Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February 1967. doi: 10.1016/S0019-9958(67)80007-X.

Yechezkel Zalcstein. Locally testable languages. *Journal of Computer and System Sciences*, 6(2): 151–167, April 1972. doi: 10.1016/S0022-0000(72)80020-5.

Bohdan Zelinka. Graphs of semigroups. *Časopis pro pěstování matematiky*, 106(4):407–408, 1981. doi: 10.21136/CPM.1981.108493.