Finite-state Machine Construction Methods and Algorithms for Phonology and Morphology

by

Mans Hulden

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF LINGUISTICS

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 0 9

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Mans Hulden
entitled Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: Nov 12 2009
Michael Hammond

Date: Nov 12 2009
Lauri Karttunen

Date: Nov 12 2009
Adam Ussishkin

Date: Nov 12 2009
Andrew Wedel

Date: Nov 12 2009
Erwin Chan

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: Nov 12 2009
Dissertation Director: Michael Hammond

STATEMENT BY AUTHOR

ACKNOWLEDGEMENTS

There are many people I am indebted to for their direct and indirect contributions to this dissertation. I have been exceptionally lucky to have Mike Hammond as my advisor. His broad range of interests in linguistics and computation combined with warm, unwavering support for all the deviant research projects I have proposed during my graduate student career is largely what has made me finish this work.

Lauri Karttunen has—both before and after he became the outside committee member—gone far beyond any reasonable expectations of helpfulness, support, and advice. Andy Wedel and Adam Ussishkin have, since I joined the graduate program, tried to make me understand phonology and linguistic theory and responded to my objections with more patience than I ever would have. Another lucky episode was that Erwin Chan, who happened to be both very well informed on my topic and willing to help me, joined us at Arizona and my committee toward the end my stay.

Kimmo Koskenniemi was the person who first introduced me to and got me hooked on finite state machines at the University of Helsinki. His subsequent support and guidance throughout my graduate studies has made the transition from being completely green on the topic to my current state of off-green highly enjoyable. Anssi Yli-Jyrä, also at the University of Helsinki, has never missed an opportunity to tap into his encyclopedic knowledge and provide a deluge of feedback and advice which has certainly left its mark on my work.

Iñaki Alegria and the IXA research group in the Basque Country were willing to adopt and test my code in their own projects very early, despite knowing all the perils involved. Through their hospitality and willingness to provide me with excellent grammars garnered with pintxos and txakoli, they have surely helped me more than I could help them.

Dale Gerdemann at Tübingen, who has an uncanny eye for finding errors in logic, reasoning, and code, voluntarily decided to put large parts of *foma* in his crosshairs, which I am indebted for. He also prompted me to develop some of the algorithms in chapter 5.

Ron Kaplan generously took the time to peruse an earlier version of this manuscript, give me detailed feedback and provide me with valuable information I never could have figured out on my own.

I'm also grateful to Ken Beesley for his brave attempts to debug both the ideas and explanations of them in chapter 8. The chapter would have been far better had Ken written it. But that's not allowed.

I have been spoiled in my graduate career by the number of brilliant, helpful faculty members in linguistics at the University of Arizona. A special mention of thanks is due to Terry Langendoen for opening up new avenues for me in formal language theory and linguistics. Mary Willie deserves a medal of bravery for trying to do what was doomed to fail: to teach me Navajo and make me understand its morphology. I wasn't cut out for it. I also owe thanks to Amy Fountain for watching over my shoulder and sending me psychic waves of solidarity throughout this project. What is true of the faculty is also true of the staff in the department. Without Jennifer Columbus this project would have finished several years behind schedule: she knows I'm not exaggerating. Thanks.

My fellow students are all leaving or have already left Arizona, and that has made it easier to wrap up. If any of my buddies Yosuke Sato, Shannon Bischoff, Jerid Francom, or Mercedes Tubino Blanco were still around, I would simply never graduate because I could hang out with them. I've traveled around the world with them, and I would do it again, including being robbed of money in Mexico, robbed of sleep on the night train from Tangiers, robbed of a rental car in Barcelona, and getting lost somewhere in the Alps. Academia, boring?

Finally, insight into my all-encompassing debt to Vilja can be gained by consulting in detail her forthcoming book tentatively titled: "Enduring your SO's Dissertation: A guide to editing, proofreading, providing moral support, exercising financial restraint, providing psychological counseling, performing scientific peer review, and serving as academic crisis manager."

TABLE OF CONTENTS

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

List of Tables

LIST OF FIGURES

LIST OF FIGURES—*Continued*

List of Algorithms

Abstract

This dissertation is concerned with finite state machine-based technology for modeling natural language. Finite-state machines have proven to be efficient computational devices in modeling natural language phenomena in morphology and phonology. Because of their mathematical closure properties, finite-state machines can be manipulated and combined in many flexible ways that closely resemble formalisms used in different areas of linguistics to describe natural language. The use of finite-state transducers in constructing natural language parsers and generators has proven to be a versatile approach to describing phonological alternation, morphological constraints and morphotactics, and syntactic phenomena on the phrase level.

The main contributions of this dissertation are the development of a new model of multitape automata, the development of a new logic formalism that can substitute for regular expressions in constructing complex automata, and adaptations of these techniques to solving classical construction problems relating to finite-state transducers, such as modeling reduplication and complex phonological replacement rules.

The multitape model presented here goes hand-in-hand with the logic formalism, the latter being a necessary step to constructing the former. These multitape automata can then be used to create entire morphological and phonological grammars, and can also serve as a neutral intermediate tool to ease the construction of automata for other purposes.

The construction of large-scale finite-state models for natural language grammars is a very delicate process. Making any solution practicable requires great care in the efficient implementation of low-level tasks such as converting regular expressions, logical statements, sets of constraints, and replacement rules to automata or finite transducers. To support the overall endeavor of showing the practicability of the logical and multitape extensions proposed in this thesis, a detailed treatment of efficient implementation of finite-state construction algorithms for natural language purposes is also presented.

Part I FOUNDATIONAL FINITE-STATE TECHNIQUES AND ALGORITHMS

1. INTRODUCTION

Listen:
Slaughterhouse-Five
KURT VONNEGUT

Finite-state machines are objects that are found in nearly every serious application in computational linguistics and natural language processing. Today, whether a system is designed for syntactic parsing, semantic annotation, morphological parsing, constraint grammar parsing, dependency parsing, speech recognition or synthesis, it is almost certain that it contains significant components that rely on finite-state technology. Don Knuth, in the Art of Computer Programming (Knuth, 1998) reported an estimate that 25 percent or more of the time spent by computer programs in general is spent on sorting lists. One could argue that a similar relationship holds for natural language processing systems and finite-state technology: that is, a significant portion of computational systems that deal with natural language have subcomponents built on finite-state technology in one form or another.

The reason for the ubiquity of the technology is quite clear: finite state automata and transducers are extremely flexible and efficient at performing a variety of pattern matching and string translation tasks that often serve as the backbone for more sophisticated systems. These supporting tasks range from the mundane to the extremely complex and include tokenization, shallow parsing, disambiguation, morphological parsing, phrase structure analysis, spelling correction, and the like. Sometimes, as is the case with morphological and phonological parsing and analysis, the best-performing large-scale systems rely almost purely on finite-state models.

As research in finite-state technology continues, allowing for simple construction of finite-state subsystems of ever-increasing complexity, we are also likely to see many systems that were previously based on other technology move into the finite-state domain.

This dissertation is devoted entirely to exploring fundamental questions regarding the use of finite-state devices for modeling and processing natural language. Toward this end, we consider a variety of questions: efficiency of construction, correctness of algorithms, fundamental decidability properties of the technology, extensions of formalisms with which to construct finite-state machines, and the adaptation of linguistic formalisms to finite-state technology. Throughout, we aim at solidifying and extending the applicability of finite-state technology to natural language problems.

In particular, the work presented here contributes significant improvements to the following areas of linguistically oriented finite-state problems:

- How to design efficient fundamental algorithms that concern finite-state machines that serve linguistic purposes
- How to construct complex finite-state machines through abstract descriptions that are suitable for linguistics
- How to model nonlinear phenomena and nonconcatenative morphologies with finite-state devices

The first question, designing efficient fundamental algorithms, is addressed in Part I of the dissertation. The types of finite-state machines that one is likely to find in use for linguistic processing are very different from the machines one finds in, for example, programming language compilers and elsewhere. In finite-state natural language processing one generally encounters machines of considerable size, both in terms of alphabet size and the number of states: alphabets with thousands of symbols and machines with millions of states are not uncommon. On the other hand, these machines also tend to be very sparse when represented as graphs, that is, there are on the average very few transitions per state in relation to the alphabet size. As a result, standard algorithms for automaton construction (as found in, for instance, computer science textbooks) are not sufficient for the needs of finite-state language processing. With this in mind, a complete set of fundamental automaton construction algorithms designed to be efficient and to specifically address the needs of linguistic processing is presented.

In the second part, we address some problematic issues in existing techniques in finite-state transducer based morphology and phonology. Finite-state devices have been criticized for the difficulty of capturing reduplication processes in word formation, a fairly often-occurring phenomenon cross-linguistically. To address this issue, we develop a notation and algorithm for incorporating reduplication phenomena into finite-state transducer morphologies. The second part is concluded by an extensive investigation into the formal properties and decision algorithms regarding finite-state transducers. Here, new algorithms for the investigation of properties of finite-state transducers are developed, including tests for functionality, tests for equivalence of functional transducers, tests for ambiguity, and extraction of unambiguous and ambiguous parts of finite-state transducers. We conclude the second part with an investigation of different types of finite-state transducer models and their suitability for modeling natural language.

In the third part, we turn to the question of abstract notations of regular languages and relations. We develop two new formalisms to accommodate the needs of finite-state language processing: a logical formalism that uses standard first-order quantifiers to express constraints and relationships over strings and substrings of the type that regular expressions and previous notations in general are ill-equipped to do. We also develop a notation and compilation procedure for multitape automata. Multitape automata are attractive devices because of their potential to express nonlinear phenomena and relationships of the type that are called for in morphology and phonology. Unfortunately, no simple method has been presented to construct, use and manipulate such devices, and so they have found little actual use. We present a new formalism to describe multitape automata as well as a method by which such descriptions can be compiled and incorporated in standard finite-state linguistic models and used for parsing and generation. We also show how the results concerning logical and multitape formalisms can be used to develop actual large-scale grammars of phonology and morphology.

$$\begin{tikzcd}
\text{asunnonvälittäjiltäkö} \arrow[d, leftrightarrow, "\text{FST}"] \\
\text{asunto +N +Sg +Genitive välittäjä +N +Pl +Ablative +kO}
\end{tikzcd}$$

FIGURE 1.1. *The paradigm task of morphological analysis.*

1.1. Background

The paradigm task that finite-state models most elegantly solve is that of morphological analysis. This is the task of, given some orthographic or phonetic representation of a word as a string, producing an analysis of that word of the type illustrated for a Finnish inflected noun compound in figure 1.1. What makes finite-state technology so successful with this task is the combination of several factors.

First, there is a strong preference among linguists to describe word-formation processes in the direction of generation. That is, in characterizing morphology and phonology it is almost always perceived to be easier to begin with some abstract form, postulated by the linguist, and describe how legitimate words are constructed by applying a series of changes to this underlying form. The changes in question usually pertain to morphologically and phonologically conditioned processes familiar to linguists: elision, epenthesis, assimilation, lengthening, shortening, etc. of sounds in certain environments. For perhaps cognitive reasons, it is much more difficult to describe such processes in the reverse direction, to provide a set of rules by which an orthographic or phonetic word can be converted into some abstract analysis by a series of steps. Thus, grammars defined in the direction of generation of the type linguists feel comfortable with are not always straightforward to parse with a general model of computation.

1.1.1. An illustration

To illustrate the difficulty of morphological and phonological analysis by other methods and the elegance of finite-state-based solutions, let us look at a concrete example of a tiny fragment of Finnish noun formation, and see how we would run into difficulties in trying to capture its analysis with a general computational model.

Finnish nouns can be formed by concatenating the stem of a noun in the nominative case with a number of affixes reflecting number (singular or plural) and various case forms. A simplified model of the noun is

$$\texttt{Stem} \quad + \quad \texttt{[Sg/Pl]} \quad + \quad \texttt{Case} \tag{1.1}$$

where `Sg` corresponds to nothing, and `Pl` is marked by the vowel `i`. There are fifteen case forms to choose from in Finnish, but let us narrow the discussion down to two: `Nominative` which corresponds to nothing, and `Adessive` which is marked by the morpheme `lla`.

Hence, to form the nominative singular form of the noun `ranta` ('beach') we form:

$$\begin{array}{llll} (1) & \texttt{ranta} & \texttt{Sg} & \texttt{Nominative} \\ (2) & \texttt{ranta} & \emptyset & \emptyset \end{array} \tag{1.2}$$

Let us consider the above two correspondences in (1.2) as *string pairs*. And let us consider our task of word generation to be the set of correct mappings from (1) to (2), and the task of word parsing to be the set of correctly mapping (2) to (1), according to the generalizations given.

It is fairly obvious that we can by standard procedural mechanisms using any programming language produce an algorithm that maps (1) to (2). To generate a legitimate word-form—going from (1) to (2)—we select a noun (such as `ranta`) from a lexicon of nouns, adjoin legal affixes in the proper order, and then map the affixes to their phonological representations. This produces exactly a mapping from (1) to (2). We could also do this in the inverse direction and reconstruct (2) given (1) by similar methods.

But let's look at a more complex word-pair:

(1)	ranta	Pl	Adessive
(2)	ranno	i	lla

(1.3)

Here, things are not quite as simple. To form the adessive plural form, there are two morphophonological changes that we need to know about in addition to simply concatenating the morphemes described earlier. First, whenever a noun stem that has two syllables in Finnish ends in a, that a changes into o in the plural form. Second, a t preceded by n changes into n when it forms part of a closed syllable (as in ran.$\underline{\text{n}}$oil.la). Because of these several phenomena going on at the same time, in order to handle the mapping from (1) to (2), it seems it would be wise to introduce some intermediate steps where we change the word from the analysis to the surface form in little steps, such as:

(1)	ranta	Pl	Adessive	
(2)	ranto	Pl	Adessive	(Change a to o before Pl)
(3)	ranto	i	lla	(Change Pl to i and Adessive to lla)
(4)	ran.to	i	l.la	(Mark syllable boundaries)
(5)	ran.no	i	l.la	(Change t to n if followed by consonant before .)
(6)	ranno	i	lla	(Remove syllable boundaries)

(1.4)

Now we have broken down the word-formation process into six steps instead of two, taking into account the alternation rules we need to get the correct forms.

But as soon as we get this far, problems arise with general models of computation. How do we now go from (6) to (1)? Going from (1) to (6) still poses no problem if we wanted to model this in any generic programming language—a simple set of six procedures that changes strings into other strings as described above will do the job. The problem lies in 'undoing' the phonological changes the word was subjected to: when receiving as input rannoilla, producing ranta Pl Adessive. Considering that a more complete grammar of Finnish—or indeed almost any natural language—would contain at least 50 such rules, the complexity of thinking about the task in terms of 'undoing' phonological changes is daunting.[1]

This is where the finite-state transducer model shows its strength. Finite-state transducers, relatively simple devices that map strings to other strings, enjoy the crucial mathematical properties of being closed under composition and inversion. That means that if we can describe the changes that occur in steps (1)–(6) as individual finite-state transducers, we can compose these transducers and form another transducer that models the entire derivation in one step. This transducer can then, because of its invertibility, be used directly to perform the mapping from (1) to (6) and (6) to (1) without any extra machinery.

1.1.2. Abstract notation

The above example illustrates another circumstance about finite-state methods that is important: we seldom need to consider the actual low-level object of a finite-state transducer and what it is made of if we can construct the required transducers from abstract, linguistic descriptions. In fact, being able to construct automata and finite-state transducers from an abstract linguistic notation is a prerequisite for success—it is practically impossible to design real grammars from the atomic properties of automata—states and transitions.

So, in the above argument about the suitability of finite-state transducers for the task of modeling phonological alternation, all we really need to know is that if such a device can encode the string changes operating in (1)–(6), and if it can be composed and inverted, it can model the process. And it can do so bidirectionally; that is, it can both parse and generate word forms even if the process description itself is unidirectional. This also reflects the state of maturity of the technology—people who work with it very rarely think of the devices in terms of their constituents: states and transitions.

Kaplan and Kay, in their 1994 paper, state:

[1]Even though, as Richard Sproat notes in Sproat (1992, p.152): "Finnish morphology is really very simple."

> The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs—the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine. The only hope of success in this domain lies in developing an appropriate set of high-level algebraic operators for reasoning about languages and relations and for justifying a corresponding set of operators and automata for computation.
>
> (Kaplan and Kay, 1994, p.376)

In a way, this statement encapsulates the problem the present work hopes to alleviate. On the one hand, abstract, intuitive notation is essential if one is to express the ideas one wishes to address with finite-state tools. On the other hand, these abstract formulations must then be convertible to finite-state automata accurately and efficiently; otherwise, we will not obtain a functional system. To address the first point, we provide a logical formalism that is easy to grasp and check for correctness, and that can be directly converted to regular expressions (and, therefore, finite-state automata). Similarly, we offer multitape representations of regular relations, which facilitate the task of describing non-linear linguistic phenomena. On the latter point, we develop various methods for improving the efficiency of constructing finite-state machines.

1.1.3. A brief history

From the very early days of computational linguistics, finite state models of language have been used for many practical tasks in natural language processing. The first instance of an actual system developed is possibly the one described later by Joshi and Hopely (1997), which was a finite-state transducer based syntactic parser called Uniparse, implemented at the University of Pennsylvania in 1958 under the direction of Zellig Harris. Later, in the early 1960s, as much research effort was being poured into formally characterizing the nature of natural language phenomena, interest quickly shifted away from finite-state-based models as they were deemed less powerful than was required. In fact, finite-state models were quickly rejected as a candidate for modeling natural language syntax in Chomsky (1957), a circumstance that may have diminished interest in such models. Although investigations of finite-state models and implementations continued, the interest was more marked outside linguistics, such as in the field of computer science where finite-state based solutions were in heavy use as components for compilers for programming languages, among other things.

The impetus for a renewed interest in finite-state models arrived in the early 1980s as the problem of parsing or decomposing the morphological structure of a complex word—in a language with more intricate morphology than English—was understood to be an important first step in any natural language processing system. It had been noted already in Johnson (1972) that the apparently context-sensitive grammar presented in the *Sound Pattern of English* (SPE) (Chomsky and Halle, 1968) could actually be modeled with finite-state transducers. These were string-to-string rewriting rules of the general form

$$a \rightarrow b \,/\, c \,\_\, d \tag{1.5}$$

which would apply to a string cad, changing it into cbd, but leave the as in $caad$ untouched as they do not appear in the correct environment (between a c and a d).

Also, Langendoen (1981) pointed out, using linguistic arguments, that word-formation processes were essentially finite state.[2] Despite these and other similar observations, the first comprehensive work in this area was developed in Kaplan and Kay (1994) (published

[2]"I know of no attested natural language the word-formation component of whose grammar must be more powerful [than finite-state]" (Langendoen, 1981).

in 1994, but developed much earlier) and Koskenniemi (1983), who, at the same time as he developed the 'two-level' formalism, presented a wide-coverage implementation for Finnish morphology with the system, in effect showing the viability of finite-state based models as a basis for complex descriptions of phonology and morphology.

Soon after the publication of Koskenniemi's 'two-level model' a number of applications to different languages appeared and a large body of research is to be found from the 1980s and early 1990s focusing on morphological and phonological parsing tasks and descriptions through finite-state devices. Also, the original two-level formalism was developed in different directions thorough various augmentations. Later, the SPE-influenced approach of cascaded replacement rules joined together by composition as outlined in Kaplan and Kay (1994) was reinvigorated through a body of work done at Xerox/PARC and elsewhere (Karttunen, 1996; Kempe and Karttunen, 1996; Mohri and Sproat, 1996; Karttunen, 1997; Beesley and Karttunen, 2003).[3] As a number of technical solutions had been developed and implemented to address some of the early problems with finite-state transducer-based models such as treatment of nonconcatenative morphologies, reduplication, long-distance constraints, and the size of automata and transducers, it was becoming clear through empirical observation that, indeed, finite-state based models were probably sufficient to handle most phenomena in morphology and phonology.[4]

1.1.4. Models of phonology and morphology

As mentioned, there are two different and intertwined major strands in the research and application of finite-state computational morphology and phonology. The first is exemplified by the above example, where we assumed that to perform morphological analysis we begin with a lexicon of stems which we subject to a sequence of rewrite rules, which finally give

[3]While much of the published work related to Kaplan & Kay's approach stems from the 1990s, the techniques were employed in a number of commercial applications in the 1980s prior to the publication of the paper (Ron Kaplan, p.c.)

[4]In fact, Lauri Karttunen in 2003 was more direct and called (computational) morphology a "solved problem." (IGERT Workshop on the Cognitive Science of Language, Department of Cognitive Science, Johns Hopkins University.)

us as its output, the correct word form. This is indeed a very common model of describing word formation. It is often associated with the formalism presented in *The Sound Pattern Of English*, or SPE (Chomsky and Halle, 1968), but the descriptive method seems to go back at least to the 4th century BCE Sanskrit grammarian Pānini (Kiparsky, 2009). The other 'parallel' approach is the two-level morphology developed in Koskenniemi (1983). Both these models have as a prerequisite the ability to construct descriptions of phonological changes through finite-state transducers. In the rewrite-rule tradition, these rules are composed together to form a chain of successive changes, and in the two-level tradition, one builds transducers that encode constraints across two levels—the analysis and the word form.

Although the rewrite model and two-level model in figure 1.2 (a) and (c) are the most common ways of developing morphological and phonological analyzers, finite-state methods do not force a commitment to either way of thinking about the problem. As linguistic theories about phonology and morphology have been developed that apparently deviate greatly from these systems, they have usually been found to be exactly equivalent or easily expressible through the same mechanisms (Karttunen, 1998; Gerdmann and van Noord, 2000; Karttunen, 2003; Roark and Sproat, 2007). The model in figure 1.2 (b)—a composite finite-state transducer that maps surface words to analyses and vice versa—can thus be produced from a variety of different grammatical formalisms provided one has access to the appropriate tools and algorithms for such a conversion.

For example, Optimality Theory, a linguistic theory that is nonderivational—where the theory explicitly dictates that words be formed without any 'intermediate' steps—has been modeled in Karttunen (1998) and Gerdmann and van Noord (2000) as the composition of rules that are very much like the SPE-rules presented above. In other words, through a series of intermediate steps.

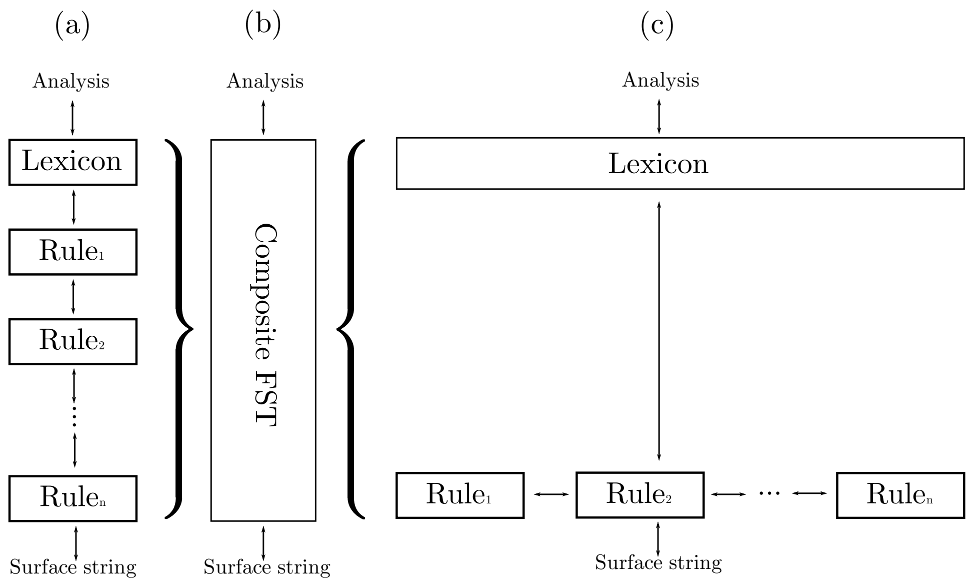


FIGURE 1.2. *Different FST models for morphological/phonological analysis and generation.*

1.2. Problems addressed

Despite the successes of the finite-state approach to modeling morphology and phonology, there are a number of issues, both computational and linguistic, that remain problematic with this approach.

1.2.1. Linguistically useful idioms

Although a ready-made finite-state transducer or automaton is simple to use as a black-box device for performing various parsing and analysis tasks in linguistics, the question of how to construct such a complex device out of simple linguistic generalizations is of foremost importance. It is of little comfort that in principle any known phenomenon in phonology and morphology can be modeled by a finite-state device unless one has a congenial notation with which to do the modeling. Apart from the already well-established two-level rules and string-to-string rewrite rules, we introduce a variety of new operations and show how such operations are to be compiled into automata and transducers. These new techniques include a logical formalism (chapter 6), multitape automata operations (chapter 7), operators for modeling string copying and reduplication phenomena (chapter 4), and a number of extensions to the string-rewriting formalisms (chapter 8).

1.2.2. Efficiency

Although the end product of compiling a linguistic grammar may be a relatively simple finite-state transducer, there are a myriad of intermediate stages involved in compiling such transducers from high-level descriptions. Compiling a single rewrite rule into a transducer may invoke hundreds, if not thousands, of calls to underlying low-level automata construction algorithms such as the classic determinization and minimization procedures. It is therefore of paramount importance to the feasibility of constructing complex finite-state models that the underlying algorithms are efficiently designed.

As we shall see, it is also often the case that the algorithms relating to automata outlined for purposes such as compiler construction found in classical computer science sources require serious revision when dealing with natural language problems. The fundamental algorithms and the modifications called for when dealing with the large (but often sparse) automata common in natural language processing are discussed in chapter 3.

As the available technology has become more flexible, finite-state technology is making inroads into areas outside morphology and phonology and often include components designed for shallow syntactic and semantic analysis as well; this is a trend one can expect to continue as the technology develops and the ability to handle larger and more complex systems improves. Such a trend directly leads to an increase in the demands for efficient implementation of the fundamentals in finite-state technology. Modeling syntactic phenomena requires much larger and more complex automata than is the case with morphology and phonology.

A related efficiency issue is that of containing nondeterminism: a large number of the expressions we wish to compile into automata and transducers rely on nondeterministic descriptions during their construction. Eventually, of course, the automata built during such intermediate stages need to be determinized, which can produce exponentially large intermediate results and render even the compilation of a relatively simple expression impossible in practice. This is often the case even though the final product would be small. While there is no general method to avoid such combinatorial explosion, there are a number of ways of refactoring nondeterministic expressions which work well in practice and can often avoid the astronomical growth of pathological intermediate results. This question is a pervasive theme, but is given special attention in chapters 3 and 6.

1.2.3. Correctness

Some automata-construction tasks, in particular those that deal with complex string rewriting modalities, have proven to require quite a number of elaborate intermediate stages in their construction. This often leads to a situation where the correct behavior of a long algorithm for producing a certain type of finite-state machine is difficult to prove—the situation is analogous to that of algorithms intended for implementation with ordinary programming languages. A step toward the direction of verifiability is taken in chapter 6 which develops an alternative formalism useful for expressing complex constraints and simplifying expressions that would otherwise be tedious to prove correct.

1.2.4. Problems specific to morphology

Some of the problems relating to the construction of morphological analyzers have lingered for a while—Sproat (1992) in quite a thorough monograph on the then-current state of the technology of computational morphology outlined some shortcomings in the area of morphology that clearly needed further work. Some of the issues he raised are still relevant. Perhaps the major points in that work which remain valid (and which is addressed in subsequent chapters) are the following:

- The assumption that morphology is mostly concatenative

- The assumption of item-and-arrangement-style morphology
- The lack of mechanisms for handling string copying and reduplication phenomena

These three points will all be discussed upon in different parts of the dissertation. The treatment of nonconcatenative morphology is addressed in two chapters: chapter 4 which presents a new model for treatment of reduplication, and chapter 9 which shows how to deal with nonlinear phenomena in general with multitape automata.

1.3. Structure

Much of the material in this dissertation has appeared in print (often in much abbreviated form) in various places. Some of these publications reflect earlier stages of the research and include Hulden (2006); Hulden and Bischoff (2007, 2008); Hulden (2009a,b,c,d,e).

In the first part we shall examine basic algorithms concerning the construction of finite-state automata and transducers. Most of this work (in chapter 3) is concerned with adapting and modifying existing or classical algorithms to the needs and peculiarities of natural language processing. In this chapter we also present algorithms for constructing complete finite-state-based morphological and phonological tools, including lexicon compilation.

In part two, we are concerned with substantial extensions to the classical finite-state construction methods. Chapter 4 is concerned in its entirety with a method for treating reduplication in finite-state morphologies. This has traditionally been the weak point of finite-state-based treatment of word formation, and the lack of effective ways to deal with reduplicated words has been frequently criticized. In chapter 5 we examine the properties of finite-state transducers in detail, focusing on algorithms and questions that are important from a natural language processing point of view. We also develop a taxonomy of transducers and analyze various types of restricted transducer models from the perspective of decidability questions and closure properties.

In part three, we depart from the classical models of finite-state transducer based morphology and develop a new formalism based on predicate logic and multitape automata, as well as applications based on these. Chapter 6 develops a novel logical formalism for specifying and compiling into automata such languages that are difficult to describe with existing methods. In chapter 7 we lay the foundations for an extension to the now-traditional two-level and rewrite formalisms, that of multitape automata, and show how such objects can be constructed and manipulated entirely through the basic single-tape automaton construction methods. In chapters 8 and 9 we apply the theory we develop for multitape automata in two ways. In the former, we provide an extensive and detailed construction method for building finite-state transducers that encode various types of rewrite rules that may be needed in linguistics applications. In the latter chapter, we show how the multitape automata can be used to directly build complex grammars, in effect, offering a third alternative to the prevailing models of two-level and rewrite grammars. In that chapter we give particular emphasis to such phenomena that are difficult to capture through the traditional methods. To illustrate the method, we provide examples from a nonconcatenative grammar that handles Arabic verbal morphology.

1.4. A note on implementation

When working with the body of research presented here, one of my goals has been to make the leap from theory to practice short. To this end, I have developed a freely available finite-state toolkit, *foma*, with which all of the algorithms and methods presented here have been implemented. The appendix contains some implementation examples with cross-references to the algorithms and formulas as well as an overview of the functionality of the toolkit. The development of an actual implementation of the material covered here has served a number of purposes. First, it has aimed to make sure that the contributions presented here go beyond the proof-of-concept level—actually testing the algorithms against a number of grammars, large and small, provides an additional level of comfort as regards their correctness. Also, as any practicing programmer knows, claims made on paper regarding ‘efficient algorithms’ should often be taken with a grain of salt unless practical implementations based on the fundamental ideas are made available for testing and critique. There are too many factors that invoke cases where standard asymptotic analysis of a given algorithm says very little about its practical usefulness—huge constants, ‘average cases’ which are never seen, biases toward dense graphs or sparse graphs when one assumes the opposite, to name a few. Second, having at my disposal a large number of more or less complete grammars developed over the years by a number of industrious people has made it much easier to test and develop the work further. This would not have been the case without a complete toolkit: a patchwork of a few toy implementations of algorithms here and there does not make it possible to compile entire grammars and hence gain the valuable information that is to be observed through the complex interaction between a large number of components. Third, there is a sizable community that has been willing to use the software for their own research purposes and grammar implementations. I have received much feedback from people who have discovered errors and inconsistencies in the toolkit—often regarding important cases that would never have occurred to me to test—and my hope is that this information has translated directly into improving the results presented here.

2. NOTATION

We give a brief overview of the notation and concepts used in subsequent chapters. We largely adhere to a fairly standard notation as found in e.g. Hopcroft and Ullman (1979) and Roche and Schabes (1997). On occasion, we will—for the sake of simplicity and clarity and for reasons relating to the special subject matter we are dealing with—introduce some conventions not widely used. These cases will be emphasized in order to avoid confusion.

2.1. Finite-state machines

Definition 2.1. *A deterministic finite-state automaton (DFA) is a 5-tuple* $(Q, \Sigma, \delta, s_0, F)$ *where*

(1) Q is a finite set of states

(2) Σ is a finite alphabet

(3) δ is a partial mapping from $Q \times \Sigma$ to Q

(4) $s_0 \in Q$ is the designated initial state

(5) $F \subseteq Q$ is a set of final states

Definition 2.2. *A non-deterministic automaton (NDFA) is exactly as above, with the exception that* δ *is a partial mapping from* $Q \times \Sigma \cup \{\epsilon\}$ *to* $\mathcal{P}(Q)$*, the power set of* Q*.*

Definition 2.3. *An* ϵ*-free automaton is a (possibly) nondeterministic automaton where no* ϵ*-transitions are present*

We say a word $w = w_0 w_1 \ldots w_n$ is accepted by a finite automaton iff there exists a sequence of states $s_0 s_1 \ldots s_n$ such that $s_n \in F$ and transitions

$$\delta(s_0, w_0) = s_1, \ldots, \delta(s_{n-1}, w_n) = s_n$$

In the event that the transition function is specified for all $Q \times \Sigma$ we call the automaton *complete*, otherwise it is *incomplete*.

The languages that can be defined by a finite automaton are the regular languages.

Definition 2.4. *A finite-state transducer is a 5-tuple* $(Q, \Sigma, \delta, s_0, F)$ *where*

(1) Q is a finite set of states

(2) Σ is a finite alphabet

(3) δ is a partial mapping from $Q \times (\Sigma \cup \{\epsilon\})$ to $Q \times (\Sigma \cup \{\epsilon\})$

(4) $s_0 \in Q$ is the designated initial state

(5) $F \subseteq Q$ is a set of final states

A finite-transducer accepts an input string under the same conditions that a finite automaton does, and possibly additionally outputs strings in Σ^* as per the transition function.

Determinism in a finite-state transducer may be interpreted in several different ways. We shall reserve the term for implying that a transducer is deterministic in the DFA-sense. This is the case if the underlying graph is deterministic when transitions are interpreted as single symbols over $(\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\})$. In other words, a finite transducer is deterministic in the DFA sense if there are no two transitions from a state with the same label pair $x{:}y$. Naturally, the precence of the symbol pair $\epsilon{:}\epsilon$ implies nondeterminism. We say a transducer is epsilon-free if there are no $\epsilon{:}\epsilon$-transitions.

Completeness in a finite-state transducer in our terminology implies that set of transition pairs $x{:}y$ that occur in any state is present in every state.

We deviate slightly from the standard definitions in the literature by declaring that the input and output alphabets in a finite-state transducer are always the same, i.e. Σ. Also, we will mostly consider so-called letter transducers, which is the case where every transition in the finite-state transducer is a single-symbol input output pair. In some specific circumstances we will consider a more extended model where (3) in the above is a mapping from $Q \times (\Sigma \cup \epsilon)$ to $Q \times \Sigma^*$, but in general we will restrict ourselves to letter transductions.

The relation that an arbitrary finite-state transducer can encode is called a *regular relation*.

In many contexts we make no or little distinction between a finite-state transducer and an automaton. We assume that if the context is unclear, an automaton can always be interpreted as a transducer that, in addition to accepting strings in its domain, maps these strings to themselves. If we want to be explicit about this, we denote an automaton $\mathcal{A}$ which is to be interpreted as an identity transducer as $Id(\mathcal{A})$.

We alse use the term ‘acceptor’ or ‘recognizer’ to indicate a finite-state machine (FSM) where every transition performs an identity transduction.

2.1.1. Finite-state machines as graphs

For many purposes, it is convenient to assume a more graph-oriented view of the finite automaton and transducer. In this case, we say a finite automaton is a directed graph with a set of vertices and edges (V, E), where an edge $e \in E$ may carry labels drawn from $\Sigma \cup \{\epsilon\}$ or label pairs $u{:}l$ representing symbols in the alphabet, or ϵ. In a graph representation of a finite-state machine there is a designated initial vertex and a set of final vertices, as per the above definitions. Under this view a finite-state automaton, or acceptor, is a graph where every edge is a pair $a{:}a$ for $a \in (\Sigma \cup \epsilon)$; a finite-state transducer may contain edges $a{:}b$ where a and b are drawn from the same set but may be unequal.

Definition 2.5. *A state s in a finite-state machine is accessible if there exists a sequence of edges from s_0 to s. A finite state machine is accessible if every state is accessible.*

Definition 2.6. *A state s in a finite-state machine is coaccessible if there exists from vertex*

s to a sequence of edges to some vertex $s_f \in F$. A finite-state machine is coaccessible if every state is coaccessible.

If a finite-state machine is both accessible and coaccesible, we say it is *trim*.

Similar to graph definitions, we say a finite-state machine is cyclic if it contains a loop. If not, it is acyclic, and hence represents a finite number of strings. Cyclicity and acyclicity is always determined with respect to the reduced graph representing a finite-state machine. A machine that accepts a finite number of strings is not cyclic despite the fact that its transition function may be completed by adding a dead state and that contains self-cycles.

2.1.1.1. Meta-symbols

A notational convention which has proven useful, especially in natural language applications, is to reserve special symbols in the alphabet as placeholders for symbols that are not explicitly declared (Beesley and Karttunen, 2003). We shall make frequent use of this device as it contributes to making expressions and finite-state machines more concise. The two symbols are @ and ?. A @-transition in a finite automaton refers to any symbol outside the current alphabet; for automata ?-transitions are not relevant. For a transducer, an edge labeled @:@ represents the identity mapping a:a for any symbol not explicitly declared in the alphabet, i.e. $a \notin \Sigma$, while ?-labels, which can appear either as the input label, the output label, or both, represents any symbol not in the declared alphabet. A transition a:? then denotes the translation of a symbol a to any symbol not in the alphabet Σ. Also, the combination ?:? represents the nonidentity mapping of any symbol not in Σ to some other symbol also not in Σ. In this latter case, the input and output must be different.

In drawing diagrams of finite-state machines we follow the usual convention that final states are denoted by a double circle and that the initial state is always s_0 in the state numbering. Figure 2.1 shows a transducer that changes at least one symbol in the input word to some other symbol, illustrating both the special alphabet symbols @ and ? and

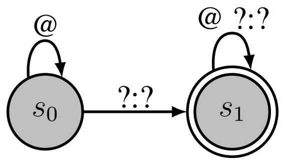

$$\begin{tikzcd}
s_0 \arrow[loop above, "@"] \arrow[r, "?:?"] & s_1 \arrow[loop above, "@\ ?:?"]
\end{tikzcd}$$

FIGURE 2.1. *A finite-state transducer that changes at least one symbol in the input to some other symbol.*

finite-state machine diagrams. We also simplify redundant pairs $a{:}a$ to a in the illustrations and it is understood that a single symbol represents an identity pair.

2.1.1.2. Minimality

For each regular language, there exists a unique minimal (in the number of states) canonical deterministic machine representing that language, based on the Myhill-Nerode equivalence classes of its states (Myhill, 1957; Nerode, 1958). If a finite-state machine is the minimal machine representing the language, we say it is minimized. For regular relations, no such canonical form of finite transducers exists. However, when interpreting a transducer as an automaton over an alphabet of label pairs, the canonical minimization again becomes available. In the case of a transducer, when we say it is minimal, minimality in this DFA-sense is implied, similar to determinism in the DFA-sense.

2.2. Regular expressions

As finite-state automata define the same languages as regular expressions do, we will make little distinction between the two and use one or the other mode of expressing languages depending on the context. Likewise, finite-state transducers correspond to regular relations—the set of relations definable by the operations of concatenation, Kleene star, union, and cross product—and we will follow a similar practice as regards them.

Apart from the standard regular expressions of concatenation ($T\ T'$), Kleene star (T^*), and union ($T \cup T'$) and the other boolean operations we will take advantage of a number of extensions. Some of them can be directly derived from more primitive expressions, while some require dedicated algorithms (presented in chapter 3) for constructing automata and transducers that involve such expressions. An overview of the expressions used are given in table 2.1. The nonstandard ones that deserve comment are as follows. The shuffle product of two languages, also called asynchronous product, denoted $T \parallel T'$, is the set of words that it is possible to construct using two words $w_1 \in T$ and $w_2 \in T'$ where each letter in w_1 and w_2 is used and their internal order is preserved. There are two primitives that can be used to construct a relation from regular languages: through the symbol pair notation $a{:}b$ and the cross product $T \times T'$ operation. The cross product of two languages T and T' is the regular relation where each string in T maps to each string in T'. Similarly, given a regular relation, we denote the regular languages in the domain and range of the relation with dom(T) and range(T). The ignore operation T/T' denotes the set of strings from T where some string from the language T'^* is inserted between every position.

As with automata vs. transducers, we make little distinction between a language and a relation. A regular language L can be interpreted as the identity relation over every string in L.

2.3. Summary

This chapter has presented an overview of the notation that will be followed in the remaining chapters. There are two noteworthy exceptions to standard notation of finite automata and transducers in the literature. First, we tend to mostly assume a graph-oriented view of finite-automata and transducers since this simplifies many of the algorithmic descriptions and makes the notation more compact. Especially for the low-level algorithms discussed in chapter 3 it is often convenient to employ a vocabulary of vertices, edges, and edge labels as opposed to transition functions. Also, we have chosen to include two meta-symbols in the

ϵ	The empty string
$\emptyset$	The empty set (language)
Σ	The alphabet/any symbol
$a{:}b$	A symbol pair
$T\,T'$	Concatenation
T^*	Kleene Closure
T^+	Kleene Plus
$T \cup T'$	Union
$T \cap T'$	Intersection
T/T'	Ignore
$T \parallel T'$	Shuffle (asynchronous product)
$T - T'$	Set subtraction
$\neg T$	Complement
T^k	k-ary concatenation
$dom(T)$	Domain extraction
$range(T)$	Range extraction
$T \circ T'$	Composition
$L \times L'$	Cross product of languages

TABLE 2.1. *Basic regular expressions.*

core notation. These denote unknown symbols on automaton transitions and transductions and allow us to avoid lengthy alphabet declarations that would otherwise be necessary in conjunction with operations on automata and transducers.

3. FUNDAMENTAL ALGORITHMS FOR FINITE-STATE AUTOMATA AND TRANSDUCERS IN NLP

3.1. Introduction

This chapter will focus on fundamental finite-state transducer construction methods. The majority of the basic techniques discussed in this chapter are well known, and on occasion, only a brief summary of some the actual techniques will be given. Instead, we will focus on such aspects as may not be common knowledge, or algorithms that require significant adaptation of existing methods to withstand the computational strain that natural language processing systems put of FSM construction methods. The purpose of the chapter is twofold: to comprehensively summarize the necessary tools needed for understanding the details in subsequent chapters, and to provide insight into the efficient implementation of the fundamental finite-state machine algorithms.

The underlying problem with developing large-scale finite-state solutions to natural languages is that in many cases, choosing the obvious-looking algorithmic solution to compiling FSMs results in excessive time and space requirements. Natural language problems usually contain many innocent-looking subproblems that must be compiled into finite-state machines; some of these subproblems will quickly cause the entire process to be uncompilable because of an unnecessary exponential explosion in the size of intermediate results. Often the solution required to make an otherwise uncompilable grammar to compile in fractions of a second is simply refactoring a complex expression in some way, reordering a set of operations, or using a different algorithm for some underlying construction. However, it is not obvious which tack to choose to design the most efficient compilation process. In many cases, the underlying question is not to make the process efficient, but to make it feasible: even the shortest regular expression can cause exponential growth when compiled into a deterministic finite-state machine. The fundamental question is to avoid this in the intermediate compilation results of a system if at all possible.

Avoiding all the pitfalls is not always necessarily aided by the vast literature on the general topic of finite-state automata and transducers. The literature abounds with conflicting results on efficiency matters and difficult-to-interpret best practices. Experiments regarding efficiency of algorithms are often made with regard to some average case which is not at all the average case in natural language applications. A case in point is that of finite-state machine determinization and minimization. A survey of the literature quickly reveals that most experiments with practical algorithms are made with very small alphabets (< 30 symbols), whereas natural language processing applications call for efficient handling of alphabets with thousands of symbols. Also, in the computer science literature, finite-state machines are often assumed to be very dense as graphs, whereas in natural language applications exactly the opposite is often the case: NLP automata are large, but sparse, and make use of large alphabets.[1]

We will first outline the primitive operations for constructing finite-state machine transducers in sections 3.2, 3.3, 3.4, 3.5 and 3.6. These sections primarily develop modifications to well-known algorithms that are motivated by efficiency concerns. Following this, we will briefly outline the construction of a number of nonprimitive operators in sections 3.7 and 3.8. Next, we will discuss efficiency considerations of the core algorithms of finite-state machine determinization and minimization in sections 3.9 and 3.10. We will also briefly outline algorithms for compiling continuation-class lexicons in section 3.11. We will conclude with some general remarks on efficiency of construction of natural language systems in section 3.12.

[1]In morphological applications this sparsity is often a result of constraining a grammar against a lexicon. Transducers that encode string-to-string replacement rules may be fairly dense, but are generally composed against a lexicon which tends to keep composite transducers sparse.

3.2. From regular expressions to automata and transducers

Fundamental to the endeavor of constructing automata and transducers for models of natural language is the ability to do so from regular expressions. A variety of methods are available to this end, the perhaps the most popular ones being the Thompson construction (Thompson, 1968), construction of automata via Glushkov automata (Glushkov, 1961), via follow automata (Ilie and Yu, 2003), and via Antimirov automata (Antimirov, 1996). We shall here focus on the Thompson construction as a starting point for compiling natural language systems. Because of its simplicity, it is amenable to adaptations and additions of regular expression operators of various kinds. Also, many of the special regular operators that are necessary for NLP purposes are most easily described through nondeterministic automata, something which the Thompson construction also produces.

3.3. Basic operations

We shall assume as a starting point the basic 'Thompson' construction of nondeterministic automata from regular expressions (Thompson, 1968). To construct transducers with that method, instead of automata, we simply modify the atomic symbol construction (see figure 3.1). Here, (a) the empty string is represented as the FSM with a single, final state with no transitions; (b) an atomic symbol pair $a_i{:}a_o$ is representable as the automaton with the single path $a_i{:}a_o$; (c) concatenation of L_1 and L_2 is created by constructing a composite automaton L_3 by adding transitions from all final states in L_1 to the initial state in L_2, and making all states in L_1 nonfinal and where the initial state of L_1 is the initial state of L_3; (d) Kleene closure, L_1^*, is constructable by adding a new initial state which is also final with ϵ-transition(s) to the formerly initial states and adding ϵ-transitions from all final states in L_1 to the new initial state; and (e) union $L_1 \cup L_2$ is represented by adding an initial state to L_3 with ϵ-transitions to the initial states of L_1 and L_2.

Additionally, the reversal of a transducer may be accomplished by designating the set of final states initial and vice versa, and replacing each transition $\delta(p, x, q)$ with $\delta(q, x, p)$.

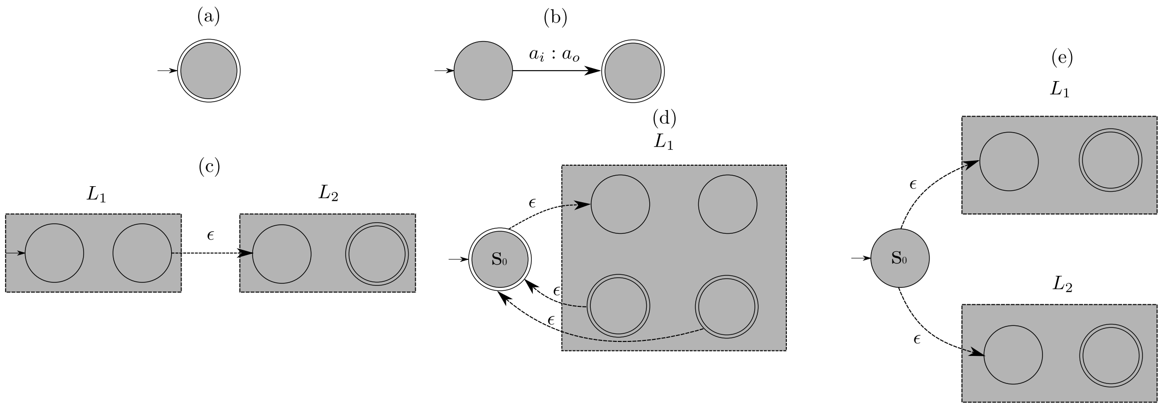


FIGURE 3.1. *A Thompson construction for transducers.*

As mentioned, we will make no distinction between a symbol a and the identity pair $a{:}a$. In regular expressions, symbols such as a will be taken to mean $a{:}a$, unless occurring as part of a symbol pair $(a{:}b)$. Whether a finite-state machine is an automaton or transducer should be evident from the context. An automaton (or recognizer) is then a special type of transducer that consists of identity pairs only on its transitions. The pair $\epsilon{:}\epsilon$ is a valid identity pair and is the only one corresponding to an empty transition in the standard interpretation of automata.

The Thompson construction yields non-deterministic ϵ-containing automata, which in many cases we will want, or need to make deterministic and minimal. The operations will be considered in later sections, after the construction operations regarding regular languages and relations have been introduced.

3.3.1. Alphabet treatment

The notational device alluded to in chapter 2, where we maintain placeholder meta-symbols **?** and @, significantly eases the compilation of complex expressions to automata as alphabets need never be declared beforehand.[2] A @:@ transition signifies any identity pair not in the currently declared alphabet, and the **?**-symbol on one side of a symbol pair signifies

[2]These meta-symbols are due to Ron Kaplan at Xerox/PARC.

any symbol also not in the alphabet. The combination **?:?** is reserved for the non-identity relation of any symbol pair where each symbol is outside the alphabet.

What makes the use of these meta-symbols particularly transparent is that the fundamental algorithms for transducer construction—with minor exceptions—need no modification to accommodate the use of the symbols. During construction, these can be treated as ordinary symbols in the alphabet. Instead, we only need to make sure that, before each operation where two transducers are input arguments, their respective alphabets are merged or 'harmonized', converting some of the unknown symbols to known symbols if they are present in one of the two machines being combined. The procedure for merging two alphabets and modifying the respective finite-state machines is given in algorithm 3.1.

Assuming this precaution is taken, any FSM involving these special symbols can be treated with the ordinary algorithms. We adopt the convention that the atomic regular expression Σ is equivalent to the FSM with the single path consisting of @. Thus, regular expressions involving the symbol Σ have a dynamic counterpart in the automaton representation. The special symbol @ on a transition in a FSM will expand its semantics whenever operations are made with the finite-state machine. The label pair $\Sigma{:}\Sigma$ is represented as a FSM with two paths, one denoting @ and the other **?:?**. This is necessary to capture the dual semantics of the regular expression $(\Sigma{:}\Sigma)$: such an expression denotes the union of the set of all identity translations and nonidentity translations. Note that $(\Sigma{:}\Sigma)$ is the only place in the notation where a single symbol a is not identical in meaning to the symbol pair $a{:}a$.

3.4. Boolean operations

Of the Boolean operations, Thompson's construction customarily only includes the union of two automata. It shall therefore be necessary to also include the other Boolean operations, intersection and set subtraction, to our construction. All of these—including the union—are traditionally described through a product construction method. This is given in

Algorithm 3.1: MERGEALPHABETS

```
Input: $FSM_1 = (Q_1, \Sigma_1, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma_2, \delta_2, t_0, F_2)$
1  begin
2      Calculate the set $N_1 = \{s|s \in \Sigma_2 \wedge s \notin \Sigma_1\}$
3      Calculate the set $N_2 = \{s|s \in \Sigma_1 \wedge s \notin \Sigma_2\}$
4      foreach $i \in \{1, 2\}$ do
5          foreach transition $\delta_i(p, @, q)$ in FSM$_i$ do
6              add transitions $\delta_i(p, n, q)$ foreach $n \in N_i$
7          end
8          foreach transition $\delta_i(p, a{:}?, q)$ in FSM$_i$ do
9              add transitions $\delta_i(p, a{:}n, q)$ foreach $n \in N_i$
10         end
11         foreach transition $\delta_i(p, ?{:}a, q)$ in FSM$_i$ do
12             add transitions $\delta_i(p, n{:}a, q)$ foreach $n \in N_i$
13         end
14         foreach transition $\delta_i(p, ?{:}?, q)$ in FSM$_i$ do
15             add transitions $\delta_i(p, n_1{:}n_2, q)$ foreach $(n_1, n_2) \in N_i$
16             add transitions $\delta_i(p, ?{:}n, q)$ foreach $n \in N_i$
17             add transitions $\delta_i(p, n{:}?, q)$ foreach $n \in N_i$
18         end
19     end
20 end
```

algorithm 3.2. The algorithm is the on-demand implementation of the product of the set of states where each machine FSM_1 and FSM_2 are traversed in parallel, generating pairs of states. The operation at hand (union, intersection, or subtraction) determines which states are designated as final states in the composite machine.

Algorithm 3.2: PRODUCTCONSTRUCTION

```
Input: $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$, OP $\in \{\cup, \cap, -\}$
Output: $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$
1  begin
2      Agenda $\leftarrow (s_0, t_0)$
3      $Q_3 \leftarrow (s_0, t_0)$
4      $u_0 \leftarrow (s_0, t_0)$
5      index $(s_0, t_0)$
6      while Agenda $\neq \emptyset$ do
7          Choose a state pair $(p, q)$ from Agenda
8          foreach pair of transitions $\delta_1(p, x, p')$ $\delta_2(q, x, q')$ do
9              Add $\delta_3((p, q), x, (p', q'))$
10             if (p',q') is not indexed then
11                 Index $(p', q')$ and add to Agenda and $Q_3$
12             end
13         end
14     end
15     foreach State $s$ in $Q_3$ = $(p, q)$ do
16         Add $s$ to $F_3$ iff $p \in F_1$ OP $q \in F_2$
17     end
18 end
```

In the algorithm, it is assumed that in a transition specified as $\delta_1(p, x, p')$, x refers to any symbol pair occurring on a transition in a transducer. The input arguments must be ϵ-free: i.e. ϵ:ϵ-transitions must not be present.

The way algorithm 3.2 specifies the product construction assumes implicitly that each automaton is complete for the set of possible symbol pairs. In practice we will want to deal with trim automata, and so do not have access to complete transition functions. For the intersection operation this is irrelevant since if there does not exist a common transition for FSM_1 and FSM_2 and some state pair (p, q), such a transition should not exist in the composite machine either. For the union and subtraction case we need to capture the possibility that one of the machines does not have a transition (which in a completed machine would lead to a sink state), and thus simulate the behavior of the sink state. Of course, if both machines lack transitions at some state pair (p, q) (where one may be a simulated sink state) we need not simulate sink states of both machines and simply do not create a transition in the composite machine.

3.4.1. Complement

With the boolean operations we can construct the complement $\neg L$ for a regular language (recognizer) L by $\Sigma^* - L$. This operation is not well defined for regular relations (transducers); however, we can define the *path* complement of a transducer to be $((\Sigma{:}\Sigma) \cup (\Sigma{:}\epsilon) \cup (\epsilon{:}\Sigma))^* - T$. This denotes the set of all transduction paths, except the ones represented by T.

3.4.2. Efficiency of product construction

For practical purposes, line 7 in the product construction needs to be implemented efficiently: that is, retrieving the next pair of transitions compatible with states p and q should preferably be accomplished in $O(1)$-time. This requires efficient access to the transition labels of at least one machine—the labels in the other machine may be traversed iteratively. Usually this presupposes an indexing of the transitions.

Asymptotically, the most favorable indexing would be to entirely index the transitions on either FSM_1 or FSM_2, so that labels in one of the machines could be found in $O(1)$-time. However, given the potential size of natural language transducers and the typical alphabet size, these indices can be quite large, i.e. taking up $|\Sigma||V|$-space (for the set of vertices in the smaller machine). Consider indexing a transducer with 1,000,000 states and 1,000 symbols—quite a normal circumstance for natural language applications. This requires 1G units of space using standard methods. Also, it is not foreseeable that the indexing has any effect: when performing an operation on two transducers, it cannot in the general case be known that the result would not be empty (no state pairs would be built beyond the initial state), in which case allocating and creating the index would be a waste.

Some natural alternatives to the full indexing are:

- Maintaining the set of transitions in sorted order for either the input or output symbols (or pre-sorting them before each operation), and traversing the transitions for each machine synchronously when looking for a transitions in the pair (p, q)
- Forgetfully indexing and re-indexing one of the FSMs for each time a pair (p, q) is retrieved from the Agenda in the product construction
- Using a hash instead of indexing, by which the space requirements are cut down to $min(|E_1|, |E_2|)$, where E_1 and E_2 represent the set of edges in the two machines.
- By default not indexing, but using a threshold of average transition outdegree beyond which some indexing/sorting is performed

For most natural language applications, one can by and large choose any of these indexing techniques without undue overhead. Hashing is likely to beneficial for very dense graphs. Experiments with all of these techniques indicate that for the majority of cases, very little is gained by indexing at all since usually one of the machines used as input for a product construction tends to be sparse. However, to guard against degenerate behavior in extreme cases, one of the above techniques becomes necessary.

3.4.3. Validity of product construction on transducers

A word may be in order about Boolean operations on transducers. As is well known (see chapter 5 for more detail), finite-state transducers are not closed under the operation of intersection or subtraction in general. However, when allowing these operations on transducers, the interpretation of subtraction and intersection should be in regard to the path languages described by the transducers in question. In effect, the path language is the transduction performed by a transducer that takes into account the precise order in which atomic transduction occur. For example, the two transductions $a{:}\epsilon\ \epsilon{:}b$ and $a{:}b$ define the same relation, but do not have the same path language because of the different order of the translations of symbols.

Algorithmic operations on the path language, i.e. the set of concatenated individual atomic relations $a{:}b$ described by a finite-state transducer, do not always coincide with the corresponding logical operations. Only in the case of union is this true. Evidently, the path union of two transducers always coincides with the logical union of their respective relations. Nevertheless, Boolean operations on the path that a transducer describes may be very useful in a variety of circumstances. We will return to this in the context of creating the cross product of two regular languages L_1 and L_2.

3.5. Transducers from automata

We shall only define two primitive constructions by which transducers can be built from automata. The symbol pair $a{:}b$, already defined as part of the Thompson construction, and the cross product construction $L_1 \times L_2$, which yields a transducer encoding all the mappings (w_1, w_2) where $w_1 \in L_1$ and $w_2 \in L_2$. Other, more complex transducer construction methods can always be derived from these operations. In fact, the operation of cross product subsumes the symbol pair operation, and we could maintain simplicity by only defining one primitive for transducer construction. However, as there are in principle many different ways to encode as a transducer the single-symbol relation $a \times b$, we reserve the notation $a : b$ to single-symbol pairs that implicitly state that the encoding of $a{:}b$ as a transducer contains a single transition $a{:}b$, and not any other sequence, such as $a{:}\epsilon\ \ \epsilon{:}b$.

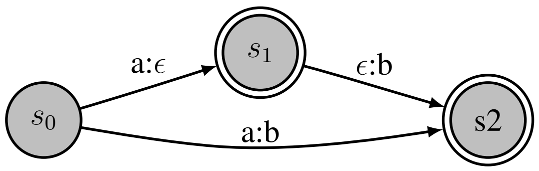


FIGURE 3.2. *Multiple paths produced by the naive cross product algorithm.*

3.5.1. Cross-product of languages

The fundamental idea of creating a transducer T that encodes the cross product of languages L_1 and L_2 is again the product construction. This algorithm is listed separately in algorithm 3.3. The only actual modification to the basic product of the two automata is the additional feature of handling non-even-length words. This requires that as we traverse the two automata in parallel and construct the composite transducer encoding the cross product, we need to account for the fact that words in L_1 may be of different length than words in L_2. This entails allowing one of the automata to halt at final states, while the other automaton continues its path to reach another final state. When the other automaton continues its path, the halted one naturally contributes with the empty string ϵ. This is encoded in lines 14–19 of the algorithm.

The basic cross product algorithm is suboptimal in the sense that it will produce multiple paths and alignments for identical relations. For example, the simple cross product of $(a \times (b \cup \epsilon))$ will produce the transducer in figure 3.2 with two alternate ways of producing the mapping from a to b.

To solve this problem, what we can do is first perform the naive cross product of L_1 and L_2 and then intersect its output with a filtering transducer $F_\times$ that removes the offending paths, illustrated in figure 3.3. The paths that are removed are such where a) ϵ symbols are aligned non-leftmost and b) ϵ:x pairs are followed by y:ϵ pairs or vice versa.

Algorithm 3.3: CROSSPRODUCT

```
Input: $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$
Output: $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$
1  begin
2      Agenda $\leftarrow (s_0, t_0)$
3      $Q_3 \leftarrow (s_0, t_0)$
4      $u_0 \leftarrow (s_0, t_0)$
5      index $(s_0, t_0)$
6      while Agenda $\neq \emptyset$ do
7          Choose a state pair $(p, q)$ from Agenda
8          foreach pair of transitions $\delta_1(p, x, p')$ $\delta_2(q, y, q')$ do
9              Add $\delta_3((p, q), x{:}y, (p', q'))$
10             if (p',q') is not indexed then
11                 Index $(p', q')$ and add to Agenda and $Q_3$
12             end
13         end
14         if $p \in F_1$ then
15             foreach $\delta_2(q, y, q')$ add transitions $\delta_3((p, q), \epsilon{:}y, (p, q'))$
16         end
17         if $q \in F_2$ then
18             foreach $y$ in $\delta_1(p, x, p')$ add transitions $\delta_3((p, q), x{:}\epsilon, (p', q))$
19         end
20     end
21     foreach State $s$ in $Q_3$ = $(p, q)$ do
22         Add $s$ to $F_3$ iff $p \in F_1 \wedge q \in F_2$
23     end
24 end
```

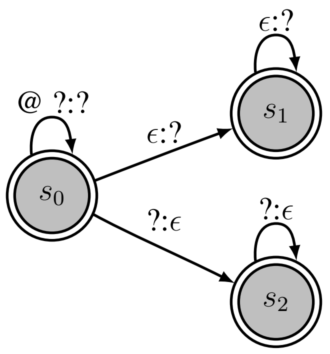


FIGURE 3.3. *The filter $F_\times$ for achieving aligned cross products.*

$$(L_1 \times L_2)_f = (L_1 \times L_2) \cap F_\times \tag{3.1}$$

Naturally, finite-state transducers are not closed under intersection in the general case, but here we are not interested in the intersection of the relations that the transducers encode, but rather, the intersection of their paths, something which the product construction intersects correctly.

All the paths in the filter transducer are such that any actual symbol-to-symbol pairs are aligned leftmost in a string of symbol pairs, after which either a sequence of $\epsilon{:}a$ or $a{:}\epsilon$ follow, but not both. In effect, intersecting the result of a naive cross product with the filter transducer removes the paths where epsilon inputs or outputs are nonfinal. The filter transducer $F_\times$ can also be described with the regular expression

$$(\Sigma{:}\Sigma)^*((\Sigma{:}\epsilon)^* \cup (\epsilon{:}\Sigma)^*) \tag{3.2}$$

Applying the filtering mechanism may cause the resulting transducer to grow by maximally a factor of 3 since we are further intersecting the result of the naive cross product with a 3-state transducer.

To illustrate the effect of the filter transducer, consider the two transducers in figure

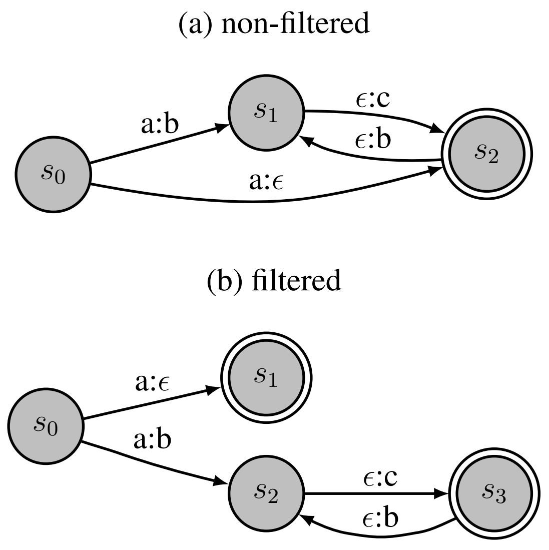


FIGURE 3.4. *Two strategies for performing the cross product of two languages* $(a \times (bc)^*)$.

3.4: both represent the relation $(a \times (bc)^*)$, but one allows arbitrary alignment of ϵ-symbols while the other has been intersected with $F_\times$.

In what follows we will always assume the *aligned* cross product construction to be used whenever $(L_1 \times L_2)$ is encountered, and that the alignment is precisely the one produced by the filtering mechanism presented. Other alignments are possible—for instance, all ϵ transitions could be aligned to the left instead of the right, etc. etc. However, as we have chosen a particular alignment to represent the cross product relation, it is obvious that the separate operation $a{:}b$—the symbol pair—could be dispensed with, and so only one primitive for building transducers from automata would be necessary.

3.6. Composition

The basic composition algorithm is also an on-demand product construction where we create new transitions from the composite states of two transducers. In effect, we traverse both transducers in parallel and output new transitions in the composite machine whenever

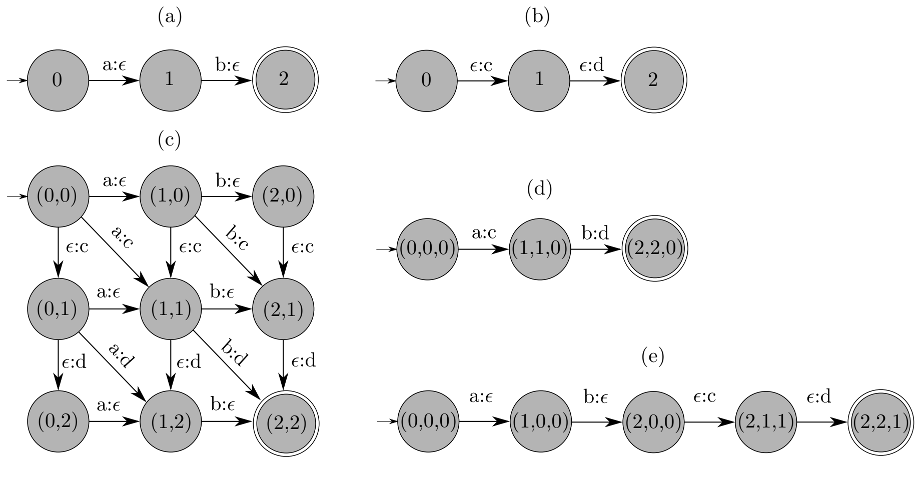


FIGURE 3.5. *Various composition strategies.*

the intermediate tapes (the output symbol of FSM_1 and the input symbol of FSM_2) match. However, an additional detail must be attended to—the possibility that FSM_1 in state p either emits an ϵ in which case FSM_2 must stay in state q while FSM_1 is allowed a move, or that FSM_2 consumes an ϵ, in which case FSM_1 must wait. These additions are captured in lines 13–24.

Note that the result of a composition operation may be nondeterministic in the FSA sense since we may create composite transitions $x{:}z$ in several ways: by combining $(x{:}y, y{:}z)$, or $(x{:}w, w{:}z)$, etc. Also, the operation may introduce ϵ-transitions by creating a composite transition from $(\epsilon{:}x, x{:}\epsilon)$, yielding $\epsilon{:}\epsilon$.

3.6.1. Multiple equivalent paths in composition

The composition algorithm 3.4 by default will yield multiple equivalent paths for certain types of transducers. In figure 3.5 (c) a composite transducer created from (a) and (b) is illustrated where the same translation can be performed through multiple paths in the composite machine. Although not strictly incorrect, this is a source of great inefficiency in any larger system that tends to compound over a long chain of compositions.

The potential creation of multiple paths is exactly the same problem that occurs with

Algorithm 3.4: COMPOSITION

```
Input: $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$
Output: $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$
begin
    Agenda $\leftarrow (s_0, t_0)$
    $Q_3 \leftarrow (s_0, t_0)$
    $u_0 \leftarrow (s_0, t_0)$
    index $(s_0, t_0)$
    while Agenda $\neq \emptyset$ do
        Choose a state pair $(p, q)$ from Agenda
        foreach pair of transitions $\delta_1(p, x{:}y, p')$ $\delta_2(q, y{:}z, q')$ do
            Add $\delta_3((p, q), x{:}z, (p', q'))$
            if (p',q') is not indexed then
                Index $(p', q')$ and add to Agenda and $Q_3$
            end
        end
        foreach transition $\delta_1(p, x{:}\epsilon, p')$ do
            Add $\delta_3((p, q), x{:}\epsilon, (p', q))$
            if (p',q) is not indexed then
                Index $(p', q)$ and add to Agenda and $Q_3$
            end
        end
        foreach transition $\delta_2(p, \epsilon{:}z, p')$ do
            Add $\delta_3((p, q), \epsilon{:}z, (p, q'))$
            if (p,q') is not indexed then
                Index $(p, q')$ and add to Agenda and $Q_3$
            end
        end
    end
    foreach State $s$ in $Q_3$ = $(p, q)$ do
        Add $s$ to $F_3$ iff $p \in F_1 \wedge q \in F_2$
    end
end
```

the cross product construction. Unfortunately, it cannot be solved post-composition as in the cross product case. It must be addressed at the time of composition, since after the operation, we are unable to distinguish how the different combinations of ϵ-symbols contributed to the final product.

The way to address this during composition is to keep track of the operation by guiding the algorithm toward a preference of ϵ consumption and controlling the order in which either machine is allowed to stay put in the product construction. We shall here outline two strategies of doing so.

3.6.1.1. Solution 1: tri-mode composition

The first variant is to guide the composition process toward a preference of synchronized ϵ alignment. That is, given a state p in T_1 and a state q in T_2, we shall avoid creating paths from (p, q) to (p', q') in the composite machine that alternate between halting the input and output machine in consecutive moves. What this means is that we disallow alternation between not consuming a symbol pair on the input side and output side, unless there exists an intervening move where symbols on both sides are consumed. To this end, instead of constructing state pairs in the product construction, we construct state triplets (p, q, m), where m is an integer in $\{0, 1, 2\}$ indicating that the composition is in a synchronized state (0), that the input side has consumed a symbol pair while the output has not contributed to the emission (1), and that the output has consumed a symbol, while the input has not emitted anything (2). The initial state is $(0, 0, 0)$ in the composite machine.

The mode indicator in a composite state triplet governs the order in which either of the argument transducers to composition must behave with respect to waiting as follows:

- In modes $\{0, 1\}$ transitions $(p \overset{x:\epsilon}{\rightarrow} p')$ create the composite transition to $(p', q, 1)$ with $x{:}\epsilon$ (FSM_2 waits)
- In modes $\{0, 2\}$ transitions $(q \overset{\epsilon:z}{\rightarrow} q')$ create the composite transition to $(p, q', 2)$ with $\epsilon{:}z$ (FSM_1 waits)

- In mode 1, FSM_1 is not allowed to wait
- In mode 2, FSM_2 is not allowed to wait

The machine in figure 3.5 (d) is the result of composing (a) and (b) with this strategy.[3] Note that some of the states in figure 3.5 (c) will still be created even with this approach. They will be non-coaccessible, however, and need to be pruned away. The illustration in figure 3.5 (d) represents the result after it has been made trim.

3.6.1.2. Solution 2: bi-mode composition

The other strategy of avoiding multiple paths in the composite machine is to, instead of stating a preference toward synchronization, disallow composite move sequences where T_1 waits followed by T_2 waiting. To this end, the composition algorithm must also disallow creating composite transitions $x{:}z$ from $(p \overset{x:\epsilon}{\rightarrow} p'), (q \overset{\epsilon:z}{\rightarrow} q')$. Otherwise we could create multiple paths through both synchronization and having T_2 wait followed by T_1 waiting.

In this case, we only need to distinguish between two internal states in the composition algorithm with m being $\{0, 1\}$. The m-value 1 would indicate that T_1 has been forced to wait. In this case we create the constraints that

- In modes $\{0, 1\}$ transitions $(q \overset{\epsilon:z}{\rightarrow} q')$ create the composite transition to $(p, q', 1)$ with $\epsilon{:}z$ (FSM_1 waits)
- In mode 1 FSM_2 is not allowed to wait

3.6.2. Efficiency and size of single paths

Given that we have at least two strategies to avoid multiple equivalent paths during composition, it is natural to ask which one is more efficient in the average case, either with respect

[3]This is very similar to a method introduced by Mohri et al. (1996) for weighted automata where the ϵ-symbols in the two machines T_1 and T_2 to be composed are first made distinct, yielding T_1' and T_2', after which a filter transducer F is introduced between the composition, and composition proceeds as $T_1' \circ F \circ T_2'$.

File	#states 2-state	#states 3-state	time(s) 2-state	time(s) 3-state
Lingala	34,294	34,294	0.860	0.830
Sanskrit	108	108	0.160	0.130
Porter	334	330	0.360	0.370
Engsyll	285	309	0.090	0.090
ArabicStems	871	871	0.050	0.050
FinnOTCounting	389	389	0.310	0.300
FinnOTMatching	245	245	7.100	6.270
BasqueErregelak	4,313	3,970	1.490	0.980
SpanishVerbs	14,933	14,931	3.920	4.050
EngPhon	52,234	52,234	4.130	4.210

TABLE 3.1. *Comparison of two composition strategies.*

to the sizes of resulting machines or the time taken by the algorithm. Table 3.1 shows this information with respect to 10 natural language grammars that involve numerous compositions of transducers. Although the illustration in figure 3.5 can lead one to believe that the tri-mode composition strategy would be more efficient with respect to the size of the results because of the more esthetic alignment of machine (d) versus machine (e), this advantage seems to be negligible for practical purposes. In the majority of cases, the final result is identical: however, table 3.1 reveals cases in which either strategy can result in smaller transducers.[4] The timing results are also inconclusive as regards the comparative efficiency of the two strategies.

[4]Lingala is an implementation of realizational morphology by Lauri Karttunen; Sanskrit is Richard Sproat's grammar given in Roark and Sproat (2007) as implemented by Dale Gerdemann; Porter is a porter stemmer by Jason Eisner; Engsyll is an English syllabifier described in Hulden (2006); ArabicStems is the stem compilation script described in chapter 9; FinnOTCounting is an Optimality Theory implementation of Finnish stress by Lauri Karttunen; FinnOTMatching is the same grammar but implemented with a different approach by Dale Gerdemann with the method described in Gerdmann and van Noord (2000); BasqueErregelak is the set of phonological alternations in a large Basque grammar by Izaskun Etxeberria and Iñaki Alegria (Alegria et al., 2009).

3.7. Extended operations

Apart from the more complex operators already introduced, there are a few remaining useful primitive operations that require state or transition manipulation.

3.7.1. Inverse

The inverse of a relation can be accomplished by simply swapping the input and output labels in a transducer. This is an operation that maintains FSA determinism and minimality.

3.7.2. Domain and range

The domain and the range can be extracted from a transducer, yielding an identity transducer (or recognizer) over the input or output language, respectively. This requires substituting transitions in the transition function $\delta(p, x{:}y, q)$ with $\delta(p, x{:}x, q)$ (for the domain) and $\delta(p, y{:}y, q)$ for the range. The special symbol **?**, if found, needs in each case to be replaced with @. The result may be nondeterministic and may contain ϵ-moves.

3.7.3. Asynchronous (shuffle) product

The shuffle of two regular languages L_1 and L_2 ($L_1 \parallel L_2$) consists of all the words it is possible to construct by picking a word from L_1 and one from L_2 and interleaving the two resulting strings in an arbitrary way. This is also called asynchronous product because of the way it is calculated, given two finite-state machines FSM_1 and FSM_2. The algorithm (3.5) is related to the product algorithm, with the difference that composite moves occur asynchronously: either a move is made by FSM_1 or FSM_2 in the composite machine, but not both.

Note that the result of the algorithm may be nondeterministic in the FSA sense. Obviously, we may construct a state pair (p, q) in the composite machine where there are several

Algorithm 3.5: AsynchronousProduct

```
Input: $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$
Output: $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$
begin
    Agenda $\leftarrow (s_0, t_0)$
    $Q_3 \leftarrow (s_0, t_0)$
    $u_0 \leftarrow (s_0, t_0)$
    while Agenda $\neq \emptyset$ do
        Choose a state pair $(p, q)$ from Agenda
        foreach transition $\delta_1(p, x, p')$ do
            Add $\delta_3((p, q), x, (p', q))$
            if (p',q) is not indexed then
                Index $(p', q)$ and add to Agenda and $Q_3$
            end
        end
        foreach transition $\delta_2(q, x, q')$ do
            Add $\delta_3((p, q), x, (p, q'))$
            if (p,q') is not indexed then
                Index $(p, q')$ and add to Agenda and $Q_3$
            end
        end
    end
    foreach State $s$ in $Q_3$ = $(p, q)$ do
        Add $s$ to $F_3$ iff $p \in F_1 \wedge q \in F_2$
    end
end
```

outgoing transitions on the same symbol a created from the original transitions $\delta(p, a, p')$ and $\delta(q, a, q')$.

The shuffle product can easily describe insertions of material in languages. For example, the single-symbol insertion of a into an arbitrary position in the language L is simply $(L \parallel a)$.

As the shuffle is constructed through the product construction, shuffle can also operate on transducers. For instance, the transducer that maps strings in a language L to themselves, with possibly arbitrary sequences of a inserted can be defined by $(L \parallel (\epsilon{:}a)^*)$.

3.7.4. Ignore

The 'ignore' operation (L_1/L_2) is closely related to the shuffle operation and is useful for many constructions. It constructs the composite language L_3 where all words are of the form $L_2^* w_1 L_2^* w_2 L_2^* \ldots L_2^* w_n L_2^*$, for $w_1 w_2 \ldots w_n \in L_1$.[5]

Algorithm 3.6: IGNORE

```
Input: $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$
Output: $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$
begin
    $FSM_3$ = Copy($FSM_1$)
    Create $|Q_1|$ copies of $FSM_2$
    foreach state $s_i \in Q_3$ do
        Add a transition $\delta_3(s_i, \epsilon, t_i)$
        foreach final state $s_f$ in the ith copy of $FSM_2$ do
            Add a transition $\delta_3(s_f, \epsilon, s_i)$
        end
    end
end
```

The algorithm for the ignore operation is given in algorithm 3.6 and illustrated in figure 3.6. The method is straightforward: for each state in FSM_1, we create an ϵ path to a copy

[5]The notation for the ignore operation is borrowed from Xerox's xfst.

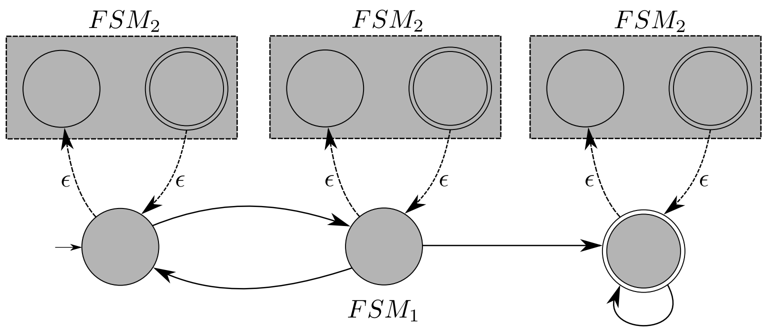


FIGURE 3.6. *Implementing the ignore operation with ϵ-transitions to copies of FSM$_2$.*

of FSM_2, from the final states of which we create an ϵ-transition back to that state. The result is always nondeterministic and contains ϵ-symbols.

Note that single-symbol ignores $L_1/a = L_1 \parallel a^*$.

As with the shuffle product, ignore may be applied to transducers as well as automata. That is, the construction method per se does not prohibit that the input arguments be non-identity mappings.

3.8. Nonprimitive operations without transducers

A large number of additional operations may be defined in terms of the primitives. We shall here focus on a few that are useful and particularly difficult to define.

3.8.1. Context restriction

It is often useful to be able to state an existential constraint on where substrings from a certain language can occur. We denote this by

$$X \Rightarrow L \,\_\, R \tag{3.3}$$

The above statement refers to the subset of Σ^* where each instance of any substring that belongs to the language X is preceded by L and succeeded by R. This can be defined in terms of the basic and Boolean operations:

$$\neg\big((\neg(\Sigma^*\, L)\; X\; \Sigma^*)\;\cup\;(\Sigma^*\; X\; \neg(R\,\Sigma^*))\big) \tag{3.4}$$

This operation is not well-defined for non-identity transducers.

It is assumed that if either or both arguments L and R are empty, they denote ϵ. This allows us to express *if-then* type statements, where *if(P,Q)* is taken to denote the language where each substring belonging to P is followed by an instance of a string from Q. That is

$$P \Rightarrow\ \_\ Q \tag{3.5}$$

This notation and its possibilities will be explored further in chapter 6.

3.8.2. Nonprimitive operations with composition and domain/range extraction

Now we have reached a point where enough primitive operations on regular languages and transducers have been defined so that in the future we can construct new ones in terms of the primitives already defined.

In particular, the combination of composition, cross product, and domain/range extraction is an extremely efficient tool for constructing complex language operations of the type that would be very difficult to do by only resorting to low-level manipulations of states and transitions.

Let us first look at some regular language constructs made simple by the combination of these three operations.

3.8.2.1. Quotients

The left or right quotient of a language is a familiar operation from formal language theory. The left quotient of a languages L_1 and L_2 is defined as

$$L_1 \backslash^L L_2 = \{w | \exists x((x \in L_1) \wedge (xw \in L_2))\} \tag{3.6}$$

Informally, this is the set of suffixes one can add to L_1 and get strings in L_2. Thus, for example $(ab)\backslash^L(abc^+)$ = $c+$.

Defining $L_1 \backslash^L L_2$ in terms of cross product, composition and range extraction is simple:

$$L_1 \backslash^L L_2 = \text{range}\big(L_2 \circ ((L_1 \times \epsilon)\ \Sigma^*)\big) \tag{3.7}$$

Similarly, the right quotient of two languages L_1 and L_2 is defined as

$$L_1 /^R L_2 = \{w | \exists x((x \in L_2) \wedge (wx \in L_1))\} \tag{3.8}$$

Informally, this is the set of prefixes one can add to L_2 to get a string in L_1. For example $(a^*ba^*)/^R(a^*b)$ = a^*.

This can be defined as:

$$L_1 /^R L_2 = \text{range}\big(L_1 \circ (\Sigma^*\ (L_2 \times \epsilon))\big) \tag{3.9}$$

3.8.2.2. Ignores

The low-level ignore operation L_1/L_2 which was defined earlier through transition/state manipulation techniques can also be defined in terms of the primitive operators.

$$L_1/L_2 = \text{range}\big(L_1 \circ (\Sigma \cup (\epsilon \times L_2))^*\big) \tag{3.10}$$

The logic in the above definition is that we compose L_1 with a transducer that accepts single symbol identity relations, or cross products of ϵ and L_2 in arbitrary numbers and in any order. Then we extract the range of this relation.

Now, earlier we had defined ignore with a low-level algorithm. Why not defer the definition of ignore until the other primitives were defined and ignore could be defined in terms of these, as above? The only real advantage we get from defining the ignore operation as a low-level one is the ability for it to operate on transducers, i.e. T_1/T_2 becomes possible to compile into a transducer which it would not be with the composition/range extraction method.[6] Also, having the two definitions of ignore serves an illustrative purpose of the relative simplicity of defining operators with transducer-based techniques as opposed to low-level techniques.

Another variant of the ignore operation, also present in Beesley and Karttunen (2003), is the ignore-internally operation, where $L_1 \ ./. \ L_2$ denotes the language L_1 with arbitrary strings from L_2 intervening, except that the first and last symbols must belong to L_1. For example, the language $(abc)./.x$ contains strings such as abc,$axbc$,$abxc$,$axbxc$, but not e.g. $xabc$.

This operation is much more inconvenient to define through low-level manipulation of states and transitions, and we offer only the alternative definition similar to the one of ignore above.

$$L_1 \ ./. \ L_2 = \text{range}\Big(L_1 \circ \big((\Sigma \ (\Sigma \cup (\epsilon \times L_2))^*\Sigma) \cup \Sigma \cup \epsilon\big)\Big) \tag{3.11}$$

3.8.2.3. Substitutions and homomorphisms

Another group of (regular) language operations that are easily defined with composition and range extraction are substitutions, homomorphisms, and inverse homomorphisms.

A substitution is the act of replacing words in a language L of the form $w_1w_2\ldots w_n$ with words of the form $w'_1w'_2\ldots w'_n$. A substitution h maps each symbol in L to a set of words in Σ^*—i.e. every symbol in L is mapped to a language. A homomorphism is

[6]Whether the ignore operator is well-defined for transducers is questionable. Consider the relations $T_1 = (a{:}b)$ and $T_2 = (a{:}\epsilon)\ (\epsilon{:}b)$. Now clearly $T_1 = T_2$ despite their possibly different transducer representations. However, $T_1/x \neq T_2/x$. (Dale Gerdemann, p.c.)

a special case of substitution where $h(x)$ is always a single string in Σ^* for each symbol $x \in \Sigma$.[7]

For example, define the homomorphism $h(a) = 10$ and $h(b) = 22$. Then $h(aab) = 101022$.

Both substitutions and homomorphisms of regular languages can be defined in the same way for some set of mappings h:

$$h(L) = \text{range}\big(L \circ ((a_1 \times h(a_1)) \cup \ldots \cup (a_n \times h(a_n)))^*\big) \tag{3.12}$$

where a_i and the corresponding $h(a_i)$ is the mapping of the symbol to the language in question.

For the above example $h(a) = 10$ and $h(b) = 22$ over the alphabet $\{a, b\}$, this becomes

$$h(L) = \text{range}(L \circ ((a \times 10) \cup (b \times 22))^*) \tag{3.13}$$

For the inverse homomorphism, we use the same definition and only invert the auxiliary transducer that L is composed with:

$$h^{-1}(L) = \text{range}\Big(L \circ \big(((a_1 \times h(a_1)) \cup \ldots \cup (a_n \times h(a_n)))^*\big)^{-1}\Big) \tag{3.14}$$

3.8.2.4. String manipulation problems

The general approach of using composition, range extraction and cross product can be extended beyond the definition of new operators. The technique is versatile enough that we can perform tasks that ordinarily require dedicated algorithms to accomplish. Some string manipulation problems that normally call for a depth-first/breadth-first search on the finite-state machine or even dynamic programming algorithms to solve can be expressed through a judicious combination of the transducer operators. Let us examine a few examples.

[7]Sometimes two different alphabets are assumed: one for the domain Σ and one for the image Γ of the substitution/homomorphism. Here we assume that they operate on a single alphabet Σ.

3.8.2.5. The shortest string problem

Consider a language L that contains an arbitrary number of strings. Suppose we wanted to extract the set of shortest strings, or one of these strings encoded in the language. If the language is represented as an automaton, the low-level approach would be to perform a breadth-first-search on the graph, starting at the initial state and stopping at the first final state encountered, or the set of first equidistant final states. However, the problem can also be expressed by the introduction of an intermediate regular relation. We can define the set of shortest strings in L as a regular language L_s by the expression

$$L_s = L - \ \Sigma^+ \text{ range}(L \circ (\Sigma{:}\Sigma)^*) \tag{3.15}$$

The logic is straightforward: the subexpression $L \circ (\Sigma{:}\Sigma)^*$ maps words in L to any word of the same length over the alphabet Σ. Concatenating with Σ^+ yields the set of strings over Σ such that each word is at least as long as some word in L. Subtracting this from L obviously leaves only the set of shortest strings in L.

Although the above expression performs the job correctly, it can be further optimized (for actual compilation) to avoid generating as an intermediate language the set of all strings that are longer than strings in L.

$$L_s = L - \Sigma^+ \text{ range}\big((\text{range}(L \circ (\Sigma{:}a)^*) \circ (a{:}\Sigma)^*)\big) \tag{3.16}$$

Here, the subexpression range$(L \circ (\Sigma{:}a)^*)$ is the language that contains the same words as L, except every symbol is an a. Composing this language with $(a{:}\Sigma)^*$ we map it to the language that contains any string over Σ that is of the same length as some word in L. Concatenating this with Σ^+ we can be sure to obtain all words in L longer than some other word in L. Note that the choice to use a as an intermediary symbol to map every word in L to is arbitrary: it is simply an auxiliary and can be any symbol in the alphabet, including a symbol that occurs in L. Although slightly more complex, this expression is vastly more efficient for actual use.

3.8.2.6. Edit distance

In a similar fashion, we can solve edit distance string problems. Consider the problem of, given some word w which is not in L, finding all the words in L that are of edit distance n from w.[8] This also appears to imply a necessary resort to algorithmic methods, but is equally solvable with the same techniques as have been developed so far. The set of words in L that are exactly one edit distance from w are definable as

$$L(w^1_{ed}) = \text{range}\big(w \circ (\Sigma^* \; (\Sigma{:}\Sigma - \Sigma \cup \Sigma{:}\epsilon \cup \epsilon{:}\Sigma) \; \Sigma^*) \circ L\big) \tag{3.17}$$

This can be generalized to any edit distance n by

$$L(w^n_{ed}) = \text{range}\big(w \circ (\Sigma^* \; (\Sigma{:}\Sigma - \Sigma \cup \Sigma{:}\epsilon \cup \epsilon{:}\Sigma) \; \Sigma^*)^n \circ L\big) \tag{3.18}$$

which constructs the set of words in L exactly n edit operations away from w.

3.8.2.7. Subsequence and substring problems

Before we move on to other topics, let us close the illustration of the capabilities and expressive power of constructing transducer-based solutions by illustrating two string-related problems whose solution can be calculated by simple transducer operations. This illustration will serve as a basis for the more advanced techniques which we will encounter in chapters 7 and 9.

The two problems in question are the longest common subsequence problem, and the longest common substring problem. Although not directly related to morphological and phonological applications, they concisely illustrate the flexibility of the combination of composition and domain/range extraction operations.

The longest common substring problem is that of finding the longest common substring shared by two words s and t. The related longest common subsequence problem (LCS) asks

[8]The measure of edit distance implies a cost: changing a symbol in w, inserting a symbol to w, and substituting one symbol in w to some other symbol all cost one unit.

the same question relating to subsequences. A substring of a string w is obtained from w by deleting a prefix and suffix of any length. A subsequence of w is obtained by deleting any number of symbols anywhere in w.

For example, assume $s = abcaa$ and $t = dbcadaa$. The longest common substring is bca obtained from s by $a\underline{bca}a$ and t by $d\underline{bca}daa$. The longest common subsequence is $bcaa$, obtained from s by $a\underline{bcaa}$ and t by $d\underline{bca}d\underline{a}a$ or $d\underline{bca}da\underline{a}$ or $d\underline{bc}ad\underline{aa}$.

Both of these problems are commonly attacked by dynamic programming techniques (Bergroth et al., 2000). Here, we will treat both by simple transducer operations.[9]

As a first step, we give the regular language definition of the set of substrings of some language L, which can be obtained by

$$\text{Substring}(L) = \text{range}(L \circ (\Sigma{:}\epsilon)^*\Sigma^*(\Sigma{:}\epsilon)^*) \tag{3.19}$$

This is done exactly following the definition of substring above: by deleting an arbitrary amount of material from the beginning and end of a string. In a similar way, we can define the set of subsequences of some language L as

$$\text{Subsequence}(L) = \text{range}(L \circ (\Sigma \cup (\Sigma{:}\epsilon))^*) \tag{3.20}$$

Here, we delete an arbitrary amount of material from anywhere in L, as per the definition.

We can now, in a similar manner as in section 3.8.2.5 define the set of longest words in L.

$$\text{Longest}(L) = L - \text{range}\big((\text{range}(L \circ (\Sigma{:}a)^*(\Sigma{:}\epsilon)^+) \circ (a{:}\Sigma)^*)\big) \tag{3.21}$$

[9]As our purpose here is not to suggest a practical method for solving these problems but to illuminate the capability of finite-state transducers we shall leave it unproved that these problems are also solved in polynomial time by the transducer approach, exactly as by dynamic programming algorithms.

The idea is the analogue of extracting the set of shortest strings in L: we subtract from L all those strings that are longer than some other string in L.[10]

Now, given these definitions, the set of longest common substrings of s and t as the expression

$$\text{LCSubstr}(s,t) = \text{Longest}(\text{Substring}(s) \cap \text{Substring}(t)) \tag{3.22}$$

and likewise the set of longest common subsequences of s and t:

$$\text{LCSubseq}(s,t) = \text{Longest}(\text{Subsequence}(s) \cap \text{Subsequence}(t)) \tag{3.23}$$

This can of course be generalized to an arbitrary number of sequences $s_1 \ldots s_n$ by

$$\text{LCSubstr}(s,t) = \text{Longest}(\text{Substring}(s_1) \cap \ldots \cap \text{Substring}(s_n)) \tag{3.24}$$

and likewise the set of longest common subsequences of s and t:

$$\text{LCSubseq}(s,t) = \text{Longest}(\text{Subsequence}(s_1) \cap \ldots \cap \text{Subsequence}(s_n)) \tag{3.25}$$

3.9. Determinization

The backbone of inductive finite state machine construction is the ability to determinize non-deterministic automata and create ϵ-free machines from ϵ-containing machines. When relying on a Thompson-style construction where non-deterministic, ϵ-containing machines are the result of many operations, the importance of efficient determinization increases. This is a necessary intermediate operation for many subsequent operations, and will need to be performed very frequently during FSM construction. For this reason, it is important that the algorithm that performs determinization be as efficient as possible. Although one

[10]Although, if L contains infinitely long words, the expression will return the empty language. This is not a problem for solving the LCS problems since we know that the input words are finitely long and hence their substrings and subsequences must be so as well.

cannot in the general case avoid an exponential increase in size when going from a non-deterministic to a deterministic machine, it is important that the construction algorithm be such that its running time is as close as possible to linear in the size of the output automaton.

The classical subset construction algorithm (Rabin and Scott, 1959) is given here in an abstract form in algorithm 3.7.

Algorithm 3.7: SUBSETCONSTRUCTION

```
   Input: $FSM_1 = (Q_1, \Sigma, \delta_1, S_0, F_1)$
   Output: $FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2)$
 1 begin
 2     $t_0 \leftarrow$ INDEX($\epsilon -$ CLOSURE($S_0$))
 3     Agenda $\leftarrow \epsilon -$ CLOSURE($S_0$)
 4     while Agenda $\neq \emptyset$ do
 5         $S \leftarrow$ AGENDAPOP()
 6         foreach symbol pair $X$ with a transition in $S$ do
 7             T $\leftarrow$ $\epsilon$-CLOSURE(move($S$,$X$))
 8         end
 9         if INDEX($T$) $= \emptyset$ then
10             Index T
11             Add T to Agenda
12             Add INDEX(T) to $Q_2$
13             if any $(s \in T) \in F_1$ then
14                 Add T to $F_2$
15             end
16         end
17         Add transition $\delta_3$(INDEX($S$), $X$, INDEX($T$))
18     end
19 end
```

In the algorithm S and T denote state sets. The function $\epsilon -$ CLOSURE(S) returns the set of states reachable by ϵ-moves from the set of states S. The function INDEX(S) associates an integer with a set of states. This integer is used when assigning state numbers to the output FSM. The initial state in the deterministic machine is the $\epsilon -$ CLOSURE of the set of initial states in the nondeterministic machine.

3.9.1. Hashing of sets

As we traverse the non-deterministic FSM with all possible symbol moves 'in parallel' and create, potentially, the power set of the set of states, some efficiency concerns arise as regards the storage and manipulation of these state sets.

In normal circumstances a bit vector approach is recommended for storing the sets of states (Leslie, 1995). As the transducers that can be encountered may be very large, however, we quickly reach a point where such a storing state sets with vectors is no longer feasible. Much space can be saved by storing the sets of states as integer sets. But if this is done, their treatment needs to be efficient. When sets are a collection of integers, it is important that the task of ascertaining whether a set is previously encountered is performed efficiently. A hashing method is suitable here. In particular, a hash where a set of integers hash to the same value regardless of the order the hash functions accesses the members of the set allows one to disregard sorting the set. This saves time when hashing needs to be performed frequently.

3.9.2. General efficiency concerns

Line 6 in the algorithm, where we retrieve the next symbol that has a transition in the set of states S deserves special attention. In general when we are dealing with sparse automata, there may be many symbols in Σ that do not have transitions, even for a collection of states S. Therefore, line 6 (and 7) which fetches the next outgoing transition from any state in the set S should operate in $O(|S|)$-time. If the entire set does not have an outgoing transition on a symbol pair X_i, no time should be spent in checking this and moving to symbol pair X_{i+1}. Achieving this is aided by pre-sorting the outgoing transitions from each state before determinization, in which case a pointer may be maintained, one for each set in the state, and the next available transition may be calculated quickly from these pointers.

If the ϵ transitions are frequent, memoization of $\epsilon - \textsc{Closure}$ may be necessary. There are various methods of doing so, and differences may depend on idiosyncrasies of the machines at hand. Results in Van Noord (2000) seem to indicate that it is advisable to store such functions where $\epsilon - \textsc{Closure}(S)$ returns not the entire set of states reachable from any state in S, but only those states that have outgoing transitions on actual symbols, as output automata may be smaller by such an approach.

3.10. Minimization

Along with determinization, the ability to efficiently minimize automata and transducers is of foremost importance in systems that construct finite-state machines of any complexity. Minimization is not only necessary for the final result in a chain of operations that result in a FSM, but a crucial operation that needs to be performed at frequent intervals during construction.

Table 3.2 illustrates this by a sample of some natural language script files that compile a finite-state transducer and that all use complex construction techniques such as composition of a variety of replacement rules. The relative timing results show that although compiling a finite-state transducer from the scripts requires a large number of calls to the minimization algorithm, not calling the minimization algorithm at all results in very inefficient compilation. In fact, some scripts failed to terminate in the 7,200 seconds allotted to it. Notably, the ‘Porter’ script took 0.360s to compile with minimization calls, while the nonminmizing approach did not terminate in the allotted two hours. The general pattern has a straightforward explanation: failing to minimize FSMs after each call to a determinization or a product construction algorithm will have exponential effects that compound very quickly in any complex system.

In what follows, we shall refer to minimization of finite-state transducers and automata alike. Naturally, for finite-state transducers there does not exist a canonical minimal form as there does for finite-state automata (see chapter 5). For finite-state automata, one can always compute the Myhill-Nerode equivalence (Myhill, 1957; Nerode, 1958) for an automaton FSM to yield the minimal automaton FSM_{min} representing that language, but

File	time w/ min(s)	no min(s)	#calls to minimize()
Lingala	0.860	3.010	2,372
Sanskrit	0.160	0.270	2,891
Porter	0.360	∞	3,346
Engsyll	0.090	43.340	317
ArabicStems	0.050	1.280	1,604
FinnOTCounting	0.310	8.900	1,922
FinnOTMatching	6.270	∞	3,679
BasqueErregelak	0.980	21.190	4,038
SpanishVerbs	3.920	16.220	27,452
EngPhon	4.130	∞	3,346

TABLE 3.2. *Differences in compiling natural language grammars with and without intermediate minimization of FSMs.*

for finite-state transducers, no such canonical form exists. However, if we interpret a finite-state transducer as an automaton where the alphabet consists of label pairs $\Gamma \subseteq \Sigma \times \Sigma$, applying a Myhill-Nerode equivalence does not change the relation encoded by the transducer, and often results in great space gains.

3.10.1. Choice of algorithm

There exists a large body of research concerning minimization algorithms for finite automata. An overview and taxonomy of finite-state minimization techniques is found in Watson (1995). The vast majority of the different minimization algorithms are based on the general idea of Moore minimization, given in Moore (1956). This in illustrated in algorithm 3.8. This approach involves calculating the set of equivalent states by first partitioning the set of states into the final and nonfinal states, and then iteratively refining the partitioning based on evidence that two states are not equivalent. Finally, the minimal automaton is generated by picking a representative state from each partition. Variants of this approach include: Hopcroft's algorithm (Hopcroft, 1971), the Hopcroft-Ullman algorithm (Hopcroft and Ullman, 1979, p.70) and the Aho-Sethi-Ullman algorithm (Aho et al., 1986, p.142).

Algorithm 3.8: GENERICMOOREMINIMIZATION

```
   Input: $FSM = (Q, \Sigma, \delta, s_0, F)$
1  Partition $Q$ into $\Pi = \{F, Q - F\}$
2  repeat
3      foreach Group $G$ in $\Pi$ do
4          split $G$ such that states $s$ and $t$
5          are in the same group if and only if
6          for all $a \in \Sigma$
7          transitions $\delta(s, a)$ go to the same group as $\delta(t, a)$
8      end
9  until no splitting occurs
```

Apart from this, there are a few algorithms which some research has found to be efficient that are not based on the general technique of Moore minimization: Brzozowski's algorithm (Brzozowski, 1963), and the Watson-Daciuk algorithm (Watson and Daciuk, 2003). Brzozowski's algorithm is extremely simple to implement given that one has access to a determinization algorithm, and simply consists of twice determinizing and reversing the automaton in question:[11]

$$\text{Determinize}(\text{Reverse}(\text{Determinize}(\text{Reverse}(\mathcal{FSM})))) \tag{3.26}$$

Another simplifying facet of this algorithm is that its output is always a trim automaton and does not need to be made accessible or coaccessible before or after minimization.

The Watson-Daciuk algorithm is based on the opposite initial assumption of Moore-based algorithms—assuming that every state is distinct and then merging equivalent states. This leads to partial results being usable, in contrast to Moore algorithms where the run must terminate before a partitioning is valid.

All of these algorithms, except Hopcroft's algorithm or the Aho-Sethi-Ullman algorithm have $O(n^2)$ or higher complexities, n being the number of states in the FSM.

[11]Note that the determinization algorithm must be able to handle multiple initial states (as final states are turned into initial states in the reversal). Otherwise, an extra state must be added with epsilon transitions, and the resulting Brzozowski maneuver requires one more determinization.

The worst-case time complexities of these algorithms suggest that either the Aho-Sethi-Ullman algorithm ($O(n\ log\ n)$) or Hopcroft's algorithm ($O(n\ log\ n)$) stand out as viable candidates for efficient minimization. However, the research appears to be inconclusive in this respect: there any many sources that report remarkable savings over Hopcroft's algorithm in the average case using algorithms such as Moore's generic algorithm (Bassino et al., 2009), which is quadratic in the worst case, the Watson-Daciuk algorithm (Watson and Daciuk, 2003; Almeida et al., 2007), which is superquadratic but still polynomial is the worst case, and Brzozowski's algorithm (Watson, 1995; Almeida et al., 2007), which is exponential in the worst case. In particular, the claim that Brzozowski's algorithm would be superior to Hopcroft's for many practical purposes is frequently encountered (see e.g. Watson (1995); Champarnaud et al. (2002); Watson and Daciuk (2003); Almeida et al. (2007); Castiglione et al. (2008) among others).

These previous results have not been reproduced in this study: as regards the algorithms developed in this chapter, Hopcroft's algorithm, assuming some care is taken in the implementation, has outperformed all of the alternatives in every test case found.[12] For this reason we shall focus on Hopcroft's minimization algorithm in the following.[13]

However, there are some caveats to the efficiency of Hopcroft's algorithm. First, the algorithm is extremely delicate and requires great care in the implementation to avoid it degenerating into an $O(n^2)$-algorithm. This is witnessed by the large number of publications that focus on reinterpreting the original description of the algorithm and describing it in a way where the asymptotic time analysis would be simpler, as in Gries (1972);

[12]There is one exception to this: when compiling a small class of pathological regular expressions that are exponentially smaller in the reverse direction such as $(a \cup b)^* a(a \cup b)^n$, assuming the automaton representing the regular expression is kept nondeterministic, the Brzozowski maneuver outperforms the sequence of determinization and Hopcroft minimization by a constant factor of 2.

[13]This potentially widespread underestimation of Hopcroft's algorithm has not gone completely unnoticed. Kiraz and Grimley-Evans write the following note after comparing two minimization implementations: "Although Brzozowski's algorithm is exponential in the worst case there are some indications that it is in practice more efficient than Hopcroft's algorithm when applied to the kind of automata used in practice. However, the observed quadratic behaviour of FIRE Lite's and Grail's implementations of Hopcroft's algorithm raises the worry that Hopcroft's algorithm may not previously have been given a fair trial: the figures seem to confirm that Hopcroft's algorithm is difficult to implement correctly" (Kiraz and Grimley-Evans, 1998).

Knuutila (2001); Berstel and Carton (2005). Second, Hopcroft's original description of the algorithm is quite unintuitive, which has also prompted a number of more descriptive publications on the topic. Third, Hopcroft's algorithm is designed to work with complete deterministic automata—which natural language automata in general are not. The asymptotic $O(|\Sigma|n\ log\ n) = O(|\Sigma||V|\ log\ |V|)$ complexity can be quite large if the alphabet is large. Especially when working with natural language automata which are incomplete, and have large alphabets but few transitions in proportion to the number of states, this involves a great penalty. Nontrivial modifications of the algorithm are required to cope with this scenario. We shall focus on all of these questions in turn: describing the algorithm, presenting some data structures that are required for the proper asymptotic behavior, and presenting modifications that are required to make the algorithm $O(|E|\ log\ |V|)$, where $|E|$ is the number of actually occurring transitions in an incomplete automaton, as opposed to $O(|\Sigma||V|\ log\ |V|)$, as the original description gives.

3.10.2. Hopcroft's algorithm

As mentioned, Hopcroft's algorithm is a variant of the more generic Moore state equivalence calculation given above. Let us first examine the canonical version of the algorithm, given in algorithm 3.9. This is necessary to develop the arguments of correctness surrounding the modifications to the algorithm to handle sparse automata.

In contrast to most adaptations of the Moore partition splitting procedure, Hopcroft's algorithm examines transitions in the reverse direction when splitting groups into subgroups. Initially, only the final and nonfinal states are in distinct groups, and for each symbol in the alphabet, a pairing consisting of a group and a symbol is put on an agenda. Then, the following procedure is iterated until the agenda is empty: select a symbol and group pair, and examine all the incoming transitions to that group with the symbol; for each set of incoming transitions where the source of the transition is not equally present in every state of the source group, split the source group. If a group B that is split into two new groups

Algorithm 3.9: HOPCROFTCANONICAL

```
Input: $FSM = (Q, \Sigma, \delta, s_0, F)$
Output: The set of equivalence classes $\Pi$
begin
    $\Pi$ = $\{F, Q - F\}$
    foreach $a \in \Sigma$ do
        Agenda $\leftarrow min((F, Q - F), a)$
    end
    while Agenda $\neq \emptyset$ do
        Choose a pair $(C, a)$ from Agenda
        Refine $\Pi$
        foreach $B \in \Pi$ split by $(C, a)$ into $B'$ and $B''$ do
            foreach $a \in \Sigma$ do
                if $(B, a)$ is on Agenda then
                    Replace $(B, a)$ with $(B', a)$ and $(B'', a)$
                else
                    Add $min((B', a), (B'', a))$ to Agenda
                end
            end
        end
    end
end
```

B' and B'' was not on the agenda from before, for each symbol a and possible group pair of the two new groups, select the smaller one and place it on the agenda. If a group B that is split is already on the agenda, place all the possible pairs (B', a) and (B'', a) for each symbol a on the agenda.[14]

For the sake of clarity, let us illustrate a run through the canonical Hopcroft algorithm (algorithm 3.9).

Consider the automaton in figure 3.7. The initial partitioning on line 2 of the algorithm will be $\{0, 1, 2, 3\}$ and $\{4\}$, the final and nonfinal states. Now, on lines 3–5, we choose the smaller of each block and inverse symbol combination to add to the agenda. We hence add to the agenda $(\{4\}, a)$ and $(\{4\}, b)$ as the block $\{4\}$ has fewer incoming transitions for a (0) than the other block (5), and the block $\{0, 1, 2, 3\}$ has 4 incoming transitions on b and the block $\{4\}$ only has one. We now choose a block to split on line 7 (note that the algorithm says nothing about the order in which elements are chosen from the Agenda). Let us choose $(\{4\}, a)$. That block has no incoming transitions on a and there is nothing to split. We now choose $(\{4\}, b)$. The block has one incoming transition on b, from state 3. The new partition Π on line 8 then becomes $\{0, 1, 2\}, \{3\}, \{4\}$. Since the block that was split was not on the agenda, we need to, for each symbol, add the smaller of $\{0, 1, 2\}$ and $\{3\}$ to the agenda on lines 10–15. For both symbols we add the same block: $(\{3\}, a)$ (with 0 incoming transitions) and $(\{3\}, b)$ (with one incoming transition). Again we choose a block and a symbol from the agenda, this time, say, $(\{3\}, a)$ which has no incoming transitions, so there is nothing to split. We now choose $(\{3\}, b)$, which has an incoming transition from state 1, and hence we split $\{0, 1, 2\}$ into $\{0, 2\}$ and $\{1\}$ on line 8. The split block was not on the agenda, so we only add the smaller for each symbol again on lines 10–15: $(\{0, 2\}, a)$ and $(\{1\}, b)$. Neither of these blocks has any incoming transitions on the respective symbols, and the algorithm will terminate as they are removed

[14]Selecting the 'smaller' group means: select the one with fewer incoming transitions to that group on symbol a.

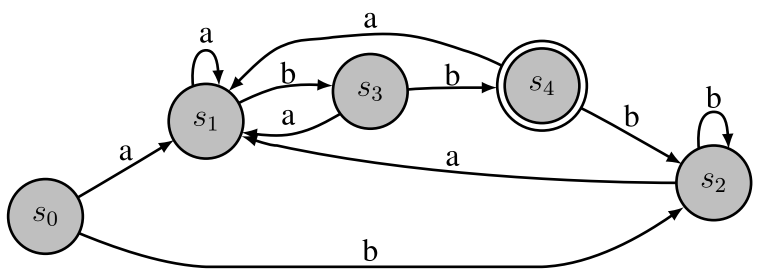


FIGURE 3.7. *A non-minimized FSM.*

next from the agenda. Hence, the final partitioning is $\{0,2\},\{1\},\{3\},\{4\}$ and states 0 and 2 are equivalent.

3.10.3. Minimizing incomplete automata

There is a fundamental efficiency problem when Hopcroft's algorithm in its canonical form is applied to natural language processing applications. In the above form, Hopcroft's algorithm requires that the automata that it operates on be *complete*; for every symbol in the alphabet and for every state there must be a transition. Hopcroft's algorithm will in general not give correct results if run on incomplete automata.

In NLP applications we are typically dealing with incomplete automata that are very sparse graphs, as seen in table 3.3 that shows the sizes of some large natural language lexicons compiled both as automata (L) and transducers (T). For those lexicons the average number of outgoing transitions per state is very low, only 8.6 for the transducers. In that table, for example, the Basque (T) transducer lexicon has an average 1.06 outgoing transitions per state, whereas, if it were complete, it would have 937 outgoing transitions per state. Obviously, if we had to make this type of a transducer's transition function complete before minimizing it (complete in the DFA sense), it would entail a great computational cost: for the Basque transducer, we would have to add a sink state and 2,324,156,862 transitions on top of the already existing 2,635,270 to make it complete.

Now, evidently it is not necessary to actually store this large number of transitions to a sink state if we want to minimize a sparse automaton. Since the set of missing transitions (to a sink state) in an incomplete automaton is the complement of the set of actually existing transitions (for each state individually), it is possible to avoid constructing these transitions and to calculate them on the fly. Although this is a possibility, the complexity of the data structures required to achieve this efficiently is such that we shall consider an alternative option.

This alternative option is as follows. We make two modifications to the Hopcroft algorithm. First, instead of choosing a symbol and a block to place on the agenda, we place entire blocks on the agenda, and iterate over the whole block and every symbol in Σ in one compound step. Second, we initialize Hopcroft by not only placing the smaller of F and $Q - F$, but by placing both blocks on the agenda. The algorithm with the modifications is given in algorithm 3.10.[15]

Let us now examine two aspects of the modifications: that they indeed produce the correct result, and that we are dealing with an $O(|E|log|V|)$ algorithm.

To reason that the changes in question indeed give the desired result, we need not reanalyze the algorithm, but rather, lean on the prior knowledge that Hopcroft's algorithm in its canonical form yields the correct minimal partitioning, and does so in $O(|\Sigma||V|\ log\ |V|)$-time.

Let us examine the minimization of an automaton FSM_c that is complete, where we add three blocks to the agenda and Π initially: the final states, the nonfinal states, except a possible sink state, and the sink state. Minimizing with an initial agenda that is a finer

[15]The discovery of being able to minimize incomplete automata through these relatively simple modifications to Hopcroft's algorithm was quite accidental and serendipitous. It was in fact the first implementation of Hopcroft's algorithm I did based my interpretation of the description of it given in Aho et al. (1974). That description leaves open many crucial details, such as the actual block handling, splitting strategy, and initialization, and also mentions nothing about the requirement that the automata must have a complete transition function. Only my subsequent reading of two articles (Béal and Crochemore, 2008; Valmari and Lehtinen, 2008) that make substantial modifications to Hopcroft's algorithm (and different from the modifications presented here) in order to deal with incomplete transition functions made me aware of the fact that I had implemented Hopcroft's algorithm in a non-standard way, and that this non-canonical method was indeed already able to correctly minimize incompletely specified deterministic automata.

Algorithm 3.10: HOPCROFTOPTIMIZED

```
Input: $FSM = (Q, \Sigma, \delta, s_0, F)$
Output: The set of equivalence classes $\Pi$
begin
    $\Pi$ = $\{F, Q - F\}$
    Agenda $\leftarrow \{F, Q - F, \text{index} = 0\}$
    while Agenda $\neq \emptyset$ do
        Choose a block $C$ from Agenda
        $i \leftarrow$index($C$)
        foreach symbol $a_j$ where $i \leq j \leq |\Sigma|$ and exists a transition on $a_j$ to $C$ do
            Refine $\Pi$ with $(C, a_j)$
            foreach $B \in \Pi$ split by $(C, a_j)$ into $B'$ and $B''$ do
                if $B \neq C$ then
                    if $B$ is on Agenda then
                        Replace $B$ with $B'$ and $B''$ in Agenda
                    else
                        Add $min(B, B'')$ to Agenda with index 0
                    end
                else
                    selfsplit $\leftarrow$ TRUE
                    Add $min(B', B'')$ to Agenda with index 0
                    Add $max(B', B'')$ to Agenda with index $j$
                end
            end
            if selfsplit then
                selfsplit $\leftarrow$ FALSE
                break
            end
        end
    end
end
```

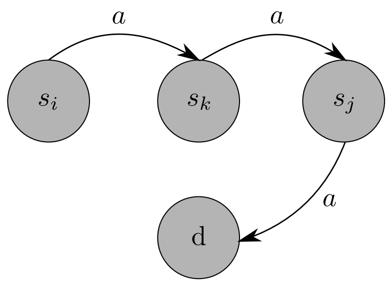


FIGURE 3.8. *The effect of splitting on sink states in Hopcroft's algorithm.*

refinement than normally naturally yields the correct result so long as the initial partitioning is correct.

If we can show that minimizing the same automaton from where the sink state has been removed, along with all its transitions (call this machine FSM) and where the initial agenda consists of the final states and the nonfinal states, then minimization of this machine without the dead state operates correctly in algorithm 3.10.

First, consider the case where FSM_c has no sink state. Here the cases FSM and FSM_c are obviously the same, and there is nothing to show. Now, consider the case where FSM_c has a sink state. As mentioned, for FSM_c we initialize Hopcroft's agenda and partitioning to the three groups $\{Q - F - d\}$, $\{F\}$, $\{d\}$. Let us examine the refinement on (Σ, d). Consider the discrimination that the splitter on the complete automaton FSM_c will perform on its initial run on (Σ, d): the contribution of splitting on d and a symbol a is to distinguish between two states s_i and s_j iff s_i or s_j (but not both) have a transition to d with a (see figure 3.8). Suppose s_j has a transition on a symbol a to d and s_i does not. If this were not the case, the two states would be indistinguishable by d and the symbol a, and will remain so when minimizing the incomplete automaton FSM. Now, if d in FSM_c distinguishes between s_i and s_j, we can show that the same algorithm will always distinguish between s_i and s_j also without the dead state in the machine FSM. Consider the state s_i in FSM. By assumption, s_i must have a transition to some other state s_k since it did not have a transition to d in the complete automaton. However, s_j cannot also have a transition to s_k on a since otherwise the automaton would not have been deterministic, and we assume the input is deterministic. Now, in minimizing the incomplete automaton FSM note that since all the states in the automaton (except d which has been removed) are put on the agenda as part of some block, we must eventually encounter and split on the block that contains s_k and the symbol a. Hence, we will always split on a block that contains s_k which in turn will place the two states s_i and s_j in different partitions. Hence, we can refine all the partitions correctly without ever splitting on a sink state d if both $Q - F$ and F are put on the agenda initially.

Let us move to the second point about the time complexity of Hopcroft's algorithm being invariant even though we add both the final and nonfinal states to the agenda initially. Consider an automaton FSM with some final states F and nonfinal states $Q - F$. Now, consider the isomorphic automaton FSM_i where all states are final. To the automaton FSM_i, add a new symbol α with transitions from all the corresponding final states in FSM to some nonfinal state in FSM_i. Now, if we initialize Hopcroft in FSM_i to only the final states F_i, and split on (Q_i, α) first, after the first iteration, we will find that the partitioning Π is exactly the original Q and $Q - F$, and further iterations of Hopcroft will proceed exactly as they would with FSM and adding both F and $Q - F$ to the agenda. Hence, given that the original asymptotic time complexity of Hopcroft must apply to the automaton FSM_i, it must be the same for FSM even though both F and $Q - F$ are initially on the agenda. Therefore, the same time complexity must hold in general whenever the algorithm is initialized with both blocks F and $Q - F$.

3.10.4. Avoiding the alphabet constant

A further optimization that is desirable is to avoid the constant time for each symbol in the alphabet in the main loop of algorithm 3.10 on line 7. It is quite important to not spend any time on iterating over groups that have no incoming transitions on a symbol a. In other words, the selection of the next symbol to split on should take $O(1)$-time in all cases, and the algorithm should waste no time looping over symbols that have no incoming transitions to C, even though every checking of such a circumstance could be done in $O(1)$-time.

In the experimental results that follow, we have named Hop_1 the canonical algorithm where automata must be completed before they are minimized. Hop_2 is the optimized version of the algorithm where the innocuous-looking check on line 7 takes $O(|B|)$-time for each symbol in Σ ($|B|$ being the size of the block) regardless of whether the block has any incoming transitions on that symbol. Hop_3 is the further optimized algorithm where this constant is avoided and where fetching the next symbol with actual incoming transitions for the block C takes $O(1)$-time.

3.10.4.1. Data structures

Since the actual data structures that are required in the proper implementation of Hopcroft's algorithm differ from what the algorithmic exposition would lead one to believe, some comments regarding this may be appropriate here. As pointed out in Gries (1972), many of the loops, if implemented literally as stated, will actually yield an $O(|V|^2)$ algorithm and there are a number of traps to avoid in the implementation. First, when refining Π on line 8 it is crucial to not be forced to inspect any of the other groups except the ones that truly have transitions to C. Doing otherwise will immediately cause quadratic behavior. Strategies to avoid this can be illustrated through the data structure given in figure 3.9. Here, we maintain lists that represent the Agenda, the current partitioning P, and the set of states E. Each state in the set of states can access the partition it belongs to as well as the other states in the same partition. As we go through a block C in *Refine* on line 8 and find a transition from some state s in a block B to the block C we can simply access s's partition where we maintain a counter of how many states in the group have a transition to the block C. After we are done looking at the inverse transitions, we know which groups to split based on whether the counter is equal to the number of states in the block B or not. Of course, none of the blocks that did not have transitions to C should have their counters

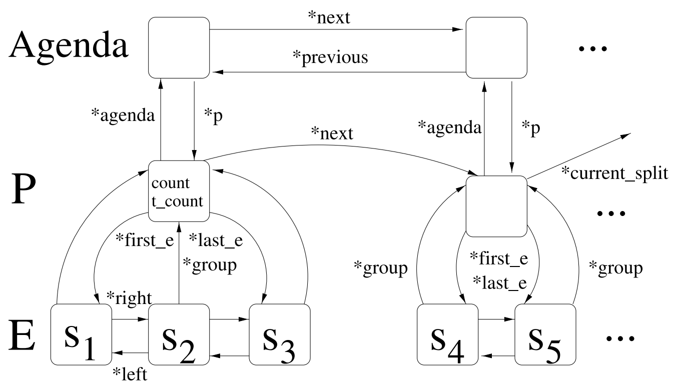


FIGURE 3.9. *An illustration of a data structure for correct implementation of Hopcroft minimization.*

inspected to avoid quadratic behavior. This can be avoided by creating a temporary list when the inverse transitions are inspected that stores the partitions which had incoming transitions to C which is iterated over afterwards to check the counters.

3.10.5. Comparison of algorithms

In order to illustrate the necessity of the optimizations over the canonical version of the Hopcroft minimization algorithm, some experiments were run on minimizing large finite automata and transducers for NLP use. The experiment consisted in compiling four lexicons, which are finite-state transducers and timing the subsequent minimization. Also, an alternative was tested where from each nonminimized lexicon only the range was extracted (the actual surface words) and the resulting automaton was determinized, after which the minimization was timed. These are given in table 3.3. The names of the languages in question in the two sets in the table are suffixed with L if it is an automaton that is minimized, and suffixed with T if it is a transducer. For the automata, the alphabet size is given in $|\Sigma|$. For the transducers, $|\Sigma|$ represents the number of actually occurring different label pairs $a{:}b$ in transitions. The different algorithms are Hop$_1$: the canonical Hopcroft algorithm where automata/transducers must be completed before minimizing; Hop$_2$: the optimized version

	Hop_1	Hop_2	Hop_3	Brz	$\|\Sigma\|$	#States	#Trans	$\text{\#States}_{min}$
English L	5.180	0.756	**0.752**	6.308	43	199,238	332,428	187,965
Basque L	18.333	2.198	**1.684**	25.738	48	320,214	483,702	75,590
Greenlandic L	11.461	2.636	**1.568**	∞	151	142,959	1,400,330	96,406
North Sami L	17.885	1.864	**0.456**	16.913	243	158,070	366,177	84,812
English T	31.810	2.360	**0.612**	2.012	268	199,009	331,639	188,181
Basque T	∞	37.198	**1.820**	8.041	937	2,483,236	2,635,270	2,482,805
Greenlandic T	297.353	25.974	**6.165**	140.43	2287	228,956	6,700,241	154,826
North Sami T	166.302	10.173	**0.404**	3.988	1923	161,437	352,117	88,024

TABLE 3.3. *Comparative timing results of different minimization algorithms.*

File	Hop_3(s)	Brz(s)	#calls to minimize()
Lingala	0.860	1.900	2,372
Sanskrit	0.160	0.450	2,891
Porter	0.360	3.180	3,346
Engsyll	0.090	0.850	317
ArabicStems	0.050	0.340	1,604
FinnOTCounting	0.310	94.320	1,922
FinnOTMatching	6.270	∞	3,679
BasqueErregelak	0.980	101.820	4,038
SpanishVerbs	3.920	9.060	27,452
EngPhon	4.130	1497.7	3,346

TABLE 3.4. *Comparison of two minimization algorithms.*

of Hopcroft without the alphabet optimization given in 3.10.4; Hop_3: the optimized version of Hopcroft where the alphabet optimization is included; and Brz, Brzozowski's algorithm.

Additionally, because of the reports that Brzozowski's algorithm would perform very efficiently in practice, and often better that Hopcroft's algorithm, we tested running the scripts used in table 3.2 using both Hop_3 and Brz as the minimizer in the compilation process. These results are given in table 3.4. Again, ∞ implies more than 2 hours of running time. To make the comparison fair, the intermediate automata were never determinized before minimizing when running Brz, as this is unnecessary.

As can be seen from the timing results, each of the successive optimization efforts is quite valuable. In particular, running the canonical version of Hopcroft's algorithm (Hop_1)

will lead to very inefficient minimization because of the $O(|\Sigma||V|\ log\ |V|)$ behavior, as opposed to the $O(|E|\ log\ |V|)$ behavior of Hop$_3$. The intermediate variant Hop$_2$ begins to suffer as large alphabets are encountered.

3.10.6. Comparison with other implementations

As a number of other tools are available for processing finite-state automata, we include here the results of some experiments run with our implementation of Hopcroft's algorithm together with the same results with other finite-state tools.

Table 3.5 shows minimization time in seconds with other available finite-state software with the same automata and transducers as were given in table 3.3. The different tools are as follows: *fst* is the finite-state toolkit developed at PARC/Xerox, *OpenFST*[16] is a finite-state toolkit developed by researchers at Google research and New York University's Courant Institute, *SFST* is the Stuttgart Finite State Transducer Tools[17], *AT&T* is the toolkit developed at AT&T[18]. D$_{HOP}$, D$_{ASU}$, D$_{W-D}$ are implementations of the Hopcroft algorithm, the Aho-Sethi-Ullman algorithm, and the Watson-Daciuk algorithm (Watson and Daciuk, 2003) by Jan Daciuk[19]. All the tools are written in C/C++. For the timing results, the UNIX command `time` was used. Each automaton/transducer was first converted to the native format of the different tools. Loading time of the files containing the finite-state machines are not included in the timing results.

Since the source code for OpenFST and SFST are available, it is known that OpenFST uses a variant of Hopcroft's algorithm. SFST, which fails to finish on the Greenlandic automaton, uses Brzozowski's algorithm. The same effect is seen in our implementation of Brz in table 3.3. Although *xfst* is a closed-source tool and the details of the implementation are not known, its API is available where comments allude to that Hopcroft's algorithm is

[16]http://www.openfst.org
[17]http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html
[18]http://www.research.att.com/˜fsmtools/fsm/
[19]http://www.eti.pg.gda.pl/˜jandac/

Tool	Hop_3	fst	OpenFST	SFST	AT&T	D_{HOP}	D_{ASU}	D_{W-D}
Version		2.12.10	1.1	1.3	4.0			
English L	**0.752**	1.220	1.688	14.021	1.992	47.219	3.880	655.90
Basque L	**1.684**	3.116	2.552	56.276	3.536	41.852	23.646	103.63
Greenlandic L	**1.568**	1.840	2.952	∞	29.942	66.661	12.553	1311.4
North Sami L	**0.456**	0.640	1.156	44.527	1.336	62.750	14.853	143.65
English T	**0.612**	1.052	1.652	4.208	3.324	122.137	20.305	651.15
Basque T	**1.820**	4.600	13.181	28.410	32.918	∞	∞	∞
Greenlandic T	**6.165**	7.041	13.649	119.008	44.671	1154.640	576.236	∞
North Sami T	**0.404**	0.552	1.112	56.276	2.732	507.617	127.480	143.24

TABLE 3.5. *Comparative timing results of minimization in different FSM applications.*

being used for minimization. It is then noteworthy that the three fastest implementations in table 3.3 all use a variant of Hopcroft's algorithm.

3.10.7. Minimization of acyclic automata

Acyclic automata come up frequently in natural language applications, when compiling finite-state machines representing lexicons, in particular. This class of machines is well known to be minimizable in linear time (Revuz, 1992). The question is, then, whether one should analyze automata before minimizing them and apply a different algorithm for acyclic ones than for cyclic ones. Obviously, looking at the worst case time complexity, this effort appears to be worthwhile.

However, it has been our observation that no acyclic automata appear to minimize in more than linear time using Hopcroft's algorithm. Figure 3.10 shows the result of minimizing a number of automata with 100,000 to 10 million states with our implementation of Hopcroft's algorithm. All the automata were fairly dense over a 6-symbol alphabet, each with an average of 4.56 outgoing transitions per state (nearly complete). As can be seen, the behavior does not depart markedly from the linear curve plotted next to it. Similar results were consistently obtained with other acyclic data sets.

The data sets we have seen may be too limited to conjecture about linear behavior of

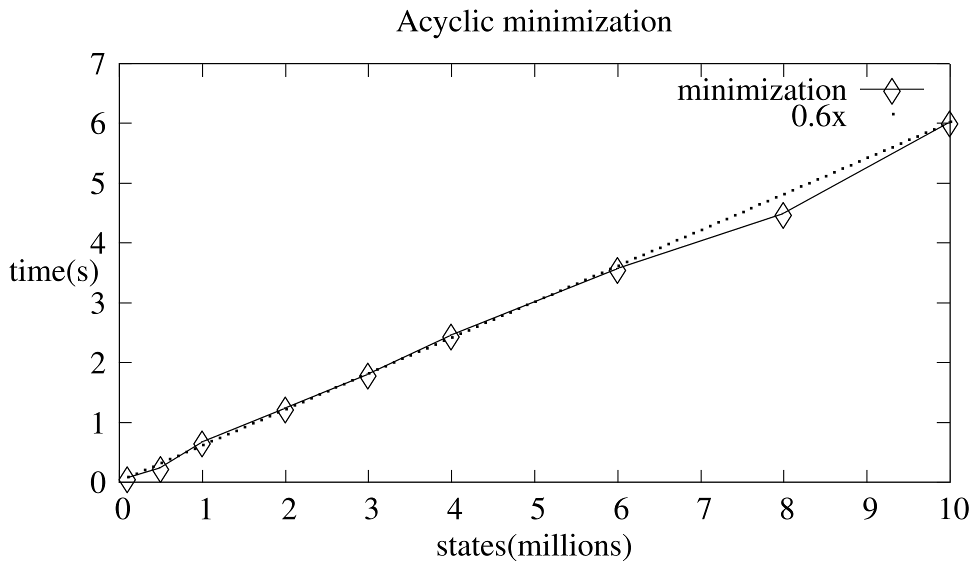


FIGURE 3.10. *Minimizing acyclic automata with Hopcroft's algorithm.*

Hopcroft's algorithm when dealing with acyclic automata. However, it has been shown that even for cyclic automata, to induce nonlinear behavior of the algorithm requires either very special types of automata, or 'unlucky' selections of the block to partition on next (Berstel and Carton, 2005; Castiglione et al., 2008). This certainly makes plausible the prospect that the algorithm works in linear time for at least the majority of acyclic machines. From our experience, the algorithm indeed gives linear time performance for acyclic automata in practice, but whether this is true in the general case seems to be an open question. Nevertheless, the immediate conclusion is that, unless other pressing concerns prompt it, the inclusion of special minimization algorithms to handle acyclic machines may be unnecessary.

3.11. Lexicon compilation

For the sake of completeness, we shall here describe a compilation method for so-called continuation-class lexicons, a useful formalism that serves as an intermediary in many practical finite-state morphological applications. The notation largely follows Karttunen (1993); Beesley and Karttunen (2003), which in turn is loosely based on the formalism developed by in Koskenniemi (1983) and also Antworth (1991). Although the formalism lacks some of the features that might be desired in developing morphological grammars for languages with nonconcatenative morphologies, it is a convenient notation for most concatenative morphologies, still widely used for describing morphotactic alternations suitable for linking with a two-level or rewrite-style grammar system. The formalism provides for a simple way to model concepts such as morpheme classes, derivation, agglutination, and compounding. It is also called *lexc*, based on a utility of the same name by Xerox/PARC.

3.11.1. Format of continuation class lexicons

The formalism itself is rather simple. We define a set of sublexicons, their contents, and their concatenative order with respect to other sublexicons. In practice, these sublexicons generally correspond to morphemes classes. Generally, each sublexicon has a unique name, and each entry in a sublexicon is followed by a specification of which sublexicons may concatenatively follow that entry. The general format is a collection of statements

```
LEXICON Name
Entry1 NextLexicon;
...
EntryN NextLexicon;
```

which declares a sublexicon, its name, entries, and for each entry, which sublexicon entries may follow that entry.

For example, the sublexicon called **Noun** with two entries is specified as

```
LEXICON Noun
cat N;
dog N;
```

indicates that each entry in this sublexicon may be followed (concatenatively) by entries in the lexicon called **N**.

We assign two special lexicons for denoting the first lexicon in concatenative order **Root**, and the final lexicon, which has no entries **#**. Thus, a *lexc* specification like

```
LEXICON Root
cat #;
```

corresponds to a finite-state machine that only accepts the word **cat**. We may also allow arbitrary regular expression entries in lexicons surrounded with special marker symbols $<$ and $>$.

```
LEXICON Verb
<s w i:a m> #;
```

Also, we can allow direct transduction specifications

```
LEXICON Verb
swim:swam #;
```

which is equivalent to specifying a regular-expression entry as the cross product of **swim** and **swam**. Also, empty entries are allowed, such as

```
LEXICON Suffix
s #;
  #;
```

which defines two suffixes in the **Suffix** sublexicon, one being the empty string, and both of which may only be followed by the end-of-word.

In general, we assume that the entries are arbitrary objects that are somehow individually representable as a finite-state transducers.

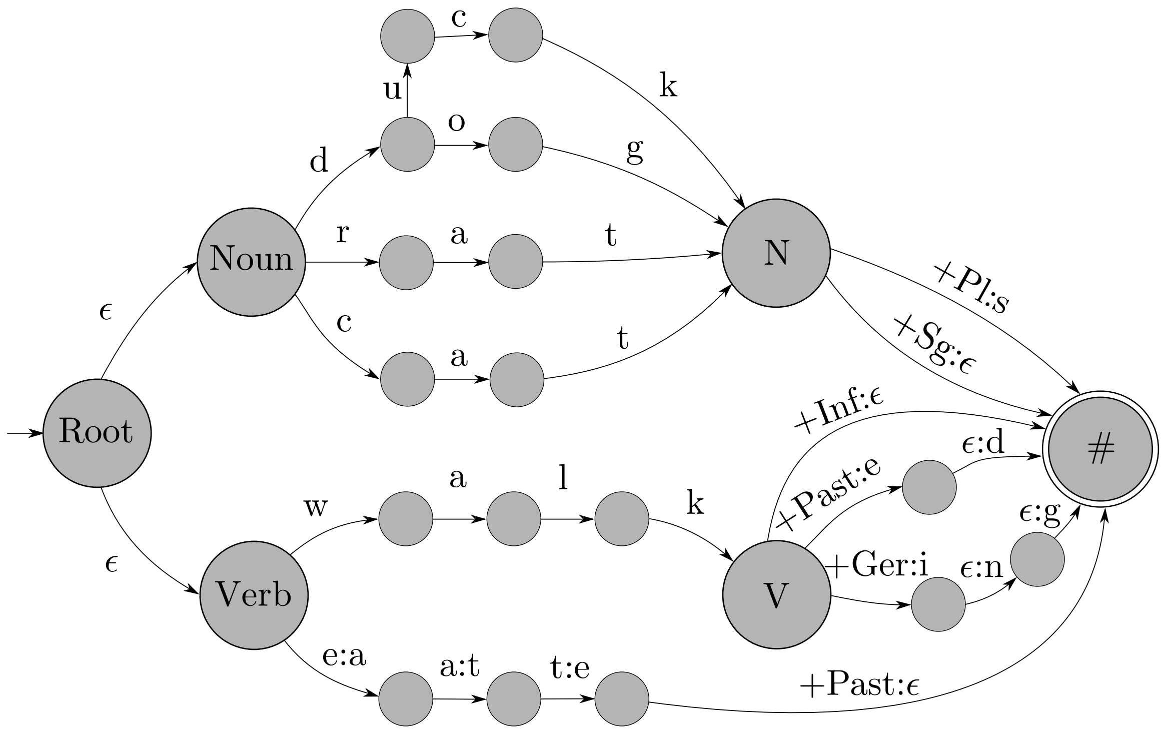
3.11.2. Compilation

The compilation procedure for such continuation-class lexicons is modeled exactly on the formalism it is specified—that is to say, we can construct an almost isomorphic nondeterministic machine that represents the set of sublexicons and their possible connections. We begin by constructing states that correspond to the sublexicons. That is, for each sublexicon **Root**, $\mathbf{Lex}_1$, …, $\mathbf{Lex}_n$, **#**, we construct a state. Now, for each entry W that is a string (or symbol-pair string) in the source lexicon $\mathbf{Lex}_i$, going to the target lexicon $\mathbf{Lex}_j$, we build a trie extending from the source lexicon state with the final transition in the trie going to the target lexicon state. When adding a word to the set of already existing words outgoing from some lexical state, we need to depart from the trie for the last symbol, or whenever a previous prefix has a transition to another lexical state.

Figure 3.11 shows a small lexicon and the nondeterministic transducer that is constructed from it using algorithm 3.11.

Notice that the result of the lexicon construction may be nondeterministic even if no regular expression entries are allowed (which are automatically created with ϵ-transitions). Since we have the ability to declare lexicon entries that consist of the empty string, the construction method will directly add an ϵ-transition from the lexicon source state to the lexicon target state. Also, two entries with the same prefix but that are of different length in the same sublexicon will generally produce nondeterminism as the longer string will need to depart from the trie at some point using the same symbol as the shorter string, or vice versa, depending on which entry was added first.

There are special methods that allow for direct construction of minimal acyclic au-



```
Lexicon Root    Lexicon Noun    Lexicon Verb
Noun;           cat  N;         walk         V;
Verb;           dog  N;         eat+Past:ate #;
                duck N;
                rat  N;

Lexicon N       Lexicon V
+Pl:s #;        +Inf:0   #;
+Sg:0 #;        +Past:ed #;
                +Ger:ing #;
```

FIGURE 3.11. *A continuation-class lexicon and the corresponding non-determinized, non-minimized FSM.*

Algorithm 3.11: ADDLEXENTRY

```
Input: Entry, Lex$_s$, Lex$_t$
1  begin
2      if entry = regex then
3          Add transition Lex$_s \xrightarrow{\epsilon}$ FSM(Entry)$_{s0}$ (the initial state)
4          Add transitions FSM(Entry)$_{sF} \xrightarrow{\epsilon}$ Lex$_t$
5      end
6      if entry = string-pair then
7          CurrState $\leftarrow$ Lex$_s$
8          $i \leftarrow 1$
9          while $i < |Entry|$ do
10             if $i < |Entry| - 1$ then
11                 if exists transition $\delta(Currstate, w_i, s_t)$ and $s_t$ is not lexicon state then
12                     CurrentState $\leftarrow s_t$
13                 else
14                     TargetState $\leftarrow$ NEWSTATE()
15                     add transition $\delta(\text{CurrentState}, w_i, \text{TargetState})$
16                 end
17             else
18                 add transition $\delta(\text{CurrentState}, w_i, \text{Lex}_t)$
19             end
20             $i \leftarrow i + 1$
21         end
22     end
23 end
```

File	lexical states (foma)	lexical arcs (hfst-2.1)	#lex entries in total
English	1.780s	11.360s	146,539
Basque	13.790s	52.560s	93,932
Greenlandic	9.810s	174.66s	122,806
North Sami	1.720s	13.700s	105,503

TABLE 3.6. *Comparison of two sublexicon compilation strategies.*

tomata from lexicons (Daciuk et al., 1998), but such methods cannot accommodate specifications that include cyclic lexicons (such as in unlimited compounding), and other features that are desirable from a natural language modeling point of view, and which are made possible with the formalism at hand.

Also, the intermediate nondeterminism is not a general concern in practice: as table 3.6 shows, very large lexicons may be compiled quite comfortably with the method, even though they need to be determinized and minimized after the initial construction steps. Apart from our implementation, the table also contains a comparison with an alternative construction method implemented in the *hfst* toolkit[20] for such lexicons which is not based on creating lexical states, but lexical transitions.

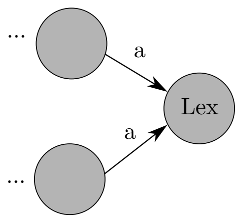
3.11.3. Efficiency concerns

A technique that may be useful to speed up compilation of lexicon compilation is to pre-minimize parts of the resulting machine before determinizing and minimizing the entire machine. This method consists of locating identical suffixes in the initially constructed machine that target the same lexical states but where the transitions arrive from different states, such as

[20] `http://hfst.sourceforge.net`

File	w/ sublexicon minimization	no sublexicon minimization	#sublexicons
English	1.780s	1.470s	29
Basque	13.790s	7.880s	421
Greenlandic	9.810s	18.880s	304
North Sami	1.720s	1.410s	870

TABLE 3.7. *Effectiveness of sublexicon minimization.*



Obviously, such state pairs can be collapsed into a single state. This collapsing of states may proceed recursively from a lexicon state backwards until no collapsible states are encountered.

The same case is seen in figure 3.11 with the two at-suffix paths arriving in lexical state **N**: the four states in question are collapsible into two.

In general, the construction method may produce a large number of such collapsible states. In order to perform such a pre-minimization quickly, the lexical states need to store their string suffixes in a hash, or some such data structure, which can then be efficiently consulted in such a minimization operation.

The effect of this pre-minimization is not always beneficial, however. Only with very large lexicons where words in sublexicons share similar suffixes does the method seem to pay off with respect to only determinizing and minimizing the resulting machine with standard techniques. Table 3.7 gives an indication of the type of gains that are to be expected: negative gains with small lexicons that otherwise compile quickly, but quite substantial gains with large cumbersome ones.

3.12. General efficiency concerns

Let us briefly return to more general concerns of efficiency that are not directly related to the underlying algorithms, but to the way in which a complex transducer is built from component machines. In many cases we have several different ways in which a equivalent transducer can be built by ordering operations or replacing a set of operations with another equivalent set. Subtle differences in one approach may often lead to significant efficiency gains over some other seemingly equivalent method.

3.12.1. Coaccessibility

As product constructions (including composition) may create machines with a large number of non-coaccessible states, and as none of the other algorithms (determinization and mininization) automatically remove these states, it is assumed that such an operation of trimming is performed whenever results may be non-coaccessible.[21] Trimming is a simple algorithm that can be performed in $O(|E|)$-time by standard depth-first-search methods: we employ a reverse depth-first search (DFS) on the graph that the machine represents, starting from the set of final states, and remove any states not encountered along with transitions to them.

3.12.2. Factoring expressions

Many times in the course of compilation of large batch of operations, achieving the desired result hinges on the proper factorization of commutative and associative operations.

The classical case where factorization of subexpressions is important is the one where we have three transducers/automata of different size R, S, T and want to perform a product construction on them, say intersection, such as:

[21] Composition, in particular, tends to produce the maximum of $|V_1||V_2|$ states as a result, but where only a fraction of these states are coaccessible.

$$R_{large} \cap S_{large} \cap T_{small} \tag{3.27}$$

Here, the implied factorization to first intersect R and S could be extremely inefficient, since the intermediate result which grows as the product $|R||S|$ could be very large. Instead, performing

$$(R_{large} \cap T_{small}) \cap S_{large} \tag{3.28}$$

is much more efficient in the general case, following the dictum that when constructing complex machines, intermediate results should be kept as small as possible.

There are more subtle incarnations of this general observation. For instance, in morphological applications, when dealing with a lexicon transducer (Lex) and a rule transducer that itself consists of individual composed transducers (Rule$_1$ ... Rule$_n$), such as

$$(\text{Lex}) \circ (\text{Rule}_1 \circ \text{Rule}_2 \circ \ldots \circ \text{Rule}_n) \tag{3.29}$$

the above factorization is often very inefficient. Although it may seem more natural to first construct the lexicon transducer and the composite rule transducer separately and subsequently composing the two, it is generally advisable to compose the lexicon against the first rule, and then the second, etc.:

$$((((\text{Lex}) \circ \text{Rule}_1) \circ \text{Rule}_2) \circ \ldots \circ \text{Rule}_n) \tag{3.30}$$

This phenomenon was first noted by Karttunen (1994) and the generalization of the effect led to the operation called ‘intersecting composition’ by which two-level grammars could be compiled more efficiently than previously.

An even more subtle illustration of the general issue is offered by time-saving refactorings in composition of two large transducers. Suppose that we indeed have a large lexicon transducer (Lex) and a composite rule transducer (Rules), and perform the composition of the two as

$$\text{Lex} \circ \text{Rules} \tag{3.31}$$

Now, in many cases it is much more efficient to actually perform

$$\text{Lex} \circ (\text{range}(\text{Lex}) \circ \text{Rules}) \tag{3.32}$$

which will evaluate to the same. We assume that the intermediate results, in particular (range(Lex) $\circ$ Rules), are determinized and minimized and made trim. The difference between the two methods is that we first extract the range of the lexicon, yielding a transducer of identity relations only, which we compose against the rules, which again is composed with the Lexicon. The reason for this seemingly backward construction is that directly composing the lexicon with the rule set creates many non-coaccessible states that need to be pruned away. To avoid this and the waste of time and memory use it leads to, we extract the range of the lexicon first and compose it against the rules which produces far less non-coaccessible states in the intermediate result.

Figure 3.12 illustrates this general effect of non-coaccessible state creation in regular composition. In that figure, only one non-coaccessible state is created during regular composition. This number may of course be vastly greater for lexicons and rule sets that are large, and in particular if the rule set transducer is dense.

Table 3.8 shows some timing results regarding compilation of actual lexica and rule sets as transducers. As can be seen from the fact that one lexicon-rule compilation is produced faster without the intermediate range extraction, the efficiency gains of the maneuver is not always true, so a general recommendation of the procedure is unwarranted. However, as can be seen from the other results, time savings with large transducers may be quite dramatic in composition.

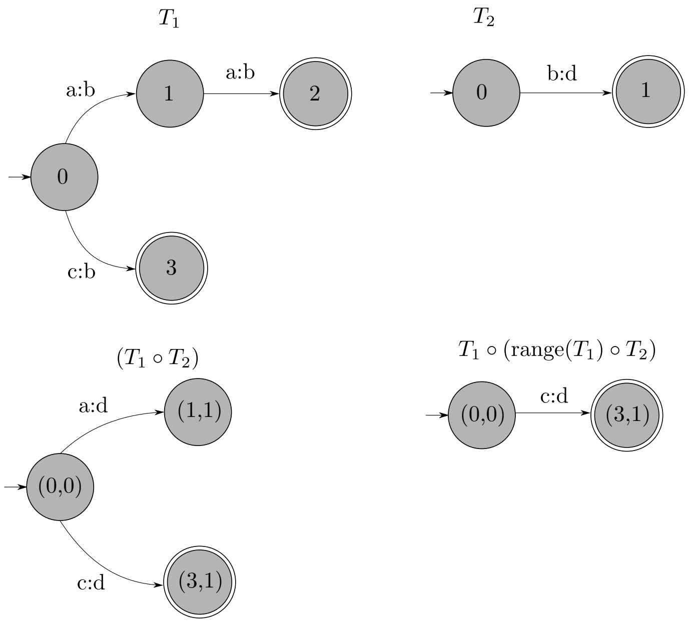


FIGURE 3.12. *The effect of pre-extracting the range of the first argument of composition.*

Language	$T_l \circ T_r$	$T_l \circ (\text{range}(T_l) \circ T_r)$	$\lvert T_l \rvert$	$\lvert T_r \rvert$
English	1.640s	2.680s	188,181	52,234
Basque	32.18s	10.880s	2,482,805	898
Greenlandic	126.05s	40.410s	154,826	12,474

TABLE 3.8. *Comparison of composition with pre-extraction of non-coaccessible states and regular composition.*

3.12.3. Bypassing determinization in product constructions

In general, there is no particular reason why the input argument machines to the product construction algorithm should be deterministic, other than that they may be minimized if they are made deterministic. Obviously, they have to be at least ϵ-free since there is no way in the product construction algorithm to represent the effect of being in multiple states in one machine with the same input string (as there is in the subset construction algorithm). This leaves open the possibility that one need only convert the arguments of any product construction input to ϵ-free machines that are not fully determinized.[22] The reason this may be beneficial is that if only a fraction of the paths in either T_1 or T_2 surface as output of the product construction, paying the cost of determinizing (and possibly minimizing) each argument beforehand may be unnecessary. Of course the resulting output may be nondeterministic, but this is the case also with the composition algorithm, even the input arguments are deterministic.

Consider two transducers T_1 and T_2, and suppose T_1 is nondeterministic in the DFA-sense for some state and input symbol $x{:}y$, and that whenever $x{:}y$ occurs in T_1, there is no corresponding $y{:}z$ transition in T_2 for any $z \in \Sigma$. Now, if we perform

$$T_1 \circ T_2 \tag{3.33}$$

obviously the $x{:}y$-transitions in T_1 will be unused. Hence, it would have been unnecessary to perform the subset construction on T_1 before performing the composition. In fact, doing so may have taken an exponential amount of time in relation to the size of T_1, the result of which would have been useless.

In this scenario, assessing or foreseeing the usefulness of bypassing determinization is much more difficult. The benefits of determinizing and minimizing the arguments of the product construction are sporadic, and it is not easy to predict for a particular pair of machine if either way of proceeding is beneficial. However, there exist pathological cases

[22]This can be performed by the subset construction algorithm by only calculating the set ϵ-closure.

where avoiding determinization of the arguments is crucial to successful compilation of a grammar, precisely because of the possibility of an intervening useless exponential-time operation of subset construction.

3.13. Discussion

This chapter has presented an overview of the fundamental algorithms and finite-state operations necessary for the extensions that will be dealt with in subsequent chapters. With the exception of methods for compiling phonological alternation rules into transducers (which is postponed until chapter 8), the methods here are also sufficient in themselves for constructing morphological and phonological parsers in the classical mode of creating a lexicon which is composed against a set of phonological alternation rules.

A great deal of emphasis has been placed on efficient implementation of the low-level operators used for constructing complex automata. In subsequent chapters we will maintain some of this emphasis, but it is clear that the key to solving large scale problems lies in the efficient construction of the primitive operations. Any neglect in this respect will propagate to the higher-level abstractions immediately. As we discuss more abstract operations later, the fundamental efficiency concern will shift from focus on data structures and details of algorithms to the avoidance of unnecessary non-determinism in the intermediate machines which could result in potentially exponential construction times.

Part II EXTENSIONS AND LIMITATIONS OF TRANSDUCER MODELS

4. REDUPLICATION AND FINITE-STATE SYSTEMS

4.1. Introduction

In this chapter we will consider an approach to including reduplication-like phenomena into the standard composition model in finite-state morphological and phonological grammar development. The primary problem of modeling reduplication in a finite-state system is that doing so indirectly requires a finite-state model of producing, recognizing, or generating identical copies of strings. In its most general and simple form this would require that one could, given a word w drawn from some set S, also recognize words of the form ww as well (and exclude words of the form vw where $|v| = |w|$ but $v \neq w$). Now, if S is not a finite set, this circumstance is inexpressible as part of a regular language or regular relation. More formally, it can be easily shown that the copy language $\{ww \mid w \in A^*\}$ over an alphabet A with two or more symbols is not a regular language.

The best we can hope for, then, is a finite-state model that can handle cases where copying, or partial copying, occurs in relation to a finite set. Even so, it is a non-trivial task to choose and implement a formalism by which one can construct automata and transducers that encode copying and partial copying and which at the same time fits in naturally with the surrounding framework that models phonology and morphology.

This chapter will address this question in the following manner. First, in section 4.2 we shall briefly survey the different types of reduplication found across languages to give an overview of the phenomena that need to be handled by any formalism that proposes to provide the tools for reduplication. In section 4.3 we will discuss various approaches found in the literature to finite-state treatment of the problem. In sections 4.4 and 4.5 we develop a method for incorporating and defining a new operator, $EQ()$, into the finite-state calculus. This operator allows us to assert that certain types of substrings be equal in content and will be our primary tool for handling reduplication. Section 4.6 then provides some examples and grammar snippets of how to employ the $EQ()$ operator to capture the variety of reduplication types attested cross-linguistically. Additional applications not directly related to reduplication, such as backreferencing and segment copying where the $EQ()$-operator can be profitably put to use, are discussed in section 4.7. Section 4.8 will present the actual algorithm by which EQ is implemented. In that section we also pursue a more formal analysis of its behavior and some decision properties related to its implementation.

4.2. Reduplication cross-linguistically

The classical case of reduplication in morphology and phonology is the phenomenon of complete reduplication—a perennial example is that of Bahasa Indonesia (4.1) or Axininca Campa (4.2), where (typically) pluralization is expressed through reduplication of a complete word:

Base Form	Reduplication	Gloss	
buku	buku-buku	'book' Pl.	
orang	orang-orang	'man' Pl. (people)	(4.1)
		(MacDonald and Darjowidjojo, 2001)	

Base Form	Reduplication	Gloss	
kawosi	kawosi-kawosi	'bathe'	(4.2)
		(Payne, 1981)	

This is a very important type of reduplication, since it appears more or less in every language (Moravcsik, 1978), although not always with an explicit grammatical function as pluralization.[1]

Another often-occurring pattern is the phenomenon where a limited amount of material is copied from a stem that may be longer than the reduplicant (Uw Oykangand):

[1]English, for instance, apart from the well known *shm*-reduplication (as in *linguistics-shminguistics*), seems to employ the device of total reduplication as a "contrastive focus" method: "I had a JOB-job once. [as opposed to an academic job]." Corpus studies have revealed that this occurs more often than one would expect: see Ghomeshi et al. (2004) for examples like this.

Base Form	Reduplication	Gloss	
elbmben	elbmbelbmben	‘red’	(4.3)
algal	algalgal	‘straight’	
		(Sommer, 1981)	

A special type of this is the case where only a part is reduplicated, with intervening material (Madurese):

Base Form	Reduplication	Gloss	
garadus	dusgaradus	‘fast and sloppy’	
abit	bitabit	‘finally’	(4.4)
		(Stevens, 1968)	

A similar example comes from Warlpiri where a limited amount of material, a ‘prosodic skeleton’ from the stem, is prefixed before the stem:

Base Form	Reduplication	Gloss	
pangurnu	pangu-pangurnu	dig/PAST	
tiirlparnkaja	tii-tiirlparnkaja	split-run/PAST	(4.5)
wantimi	wanti-wantimi	fall/NonPast	
		(Nash, 1980; Sproat, 1992)	

Also, reduplication may not always result in identical material—phonological changes may occur that result in that two (or more) sequences are similar, yet not identical, as in this example from Javanese:

Base Form	Reduplication	Gloss	
bali	bola-bali	‘return’	
iba	iba-ibu	‘mother’	(4.6)
udan	udan-udɛn	‘rain’	
		(Kiparsky, 1986; Sproat, 1992)	

The more challenging patterns occur when we find dependencies that cross and produce multiple different partial copies of the base and of affixes. We can find examples of this in the Salishan language Coeur d’Alene:

Base Form	Reduplication	Gloss	
ɛn'is	ɛ'ɛn'ɛn'is	'little ones went off one by one'	(4.7)
caq	caqcaqaqɛlipəp	'he fell on his back'	
		(Reichard, 1938)	

4.3. Previous work

Approaches to incorporating reduplication-like phenomena in finite-state systems commonly fall into two categories: those that are purely finite-state, i.e. encode all reduplication effects in a standard automaton, and those that augment the standard finite-state model in some way.

Proposals of the latter type are more common. Walther (2000) proposes a system where the transition function of a finite-state automaton is augmented with certain specific types of arcs that signal repetitions of previous subsequences in a string. This is a model that can be used as a 'run-time' solution for identifying duplicate sequences. Cohen-Sygal and Wintner (2006) develop an augmented model they call finite-state registered automata, also a 'run-time' approach, where arcs in an automaton can cause register read-write operations as well as checking the contents of stored registers. The idea is that, as a word is applied to an automaton, the run-time code stores a representation of prefixes seen in the finite registers, and at some subsequent arc traversal—presumably when a second copy of a subword is encountered—checks that the registers match the second copy.[2]

Naturally, any augmented finite-state models run the risk of not enjoying the attractive closure properties that finite-state automata and transducers do. In particular, the availability of the composition operation, the one feature that makes it possible to build a bidirectional model of morphophonology by a unidirectional description, is easily lost. It is fairly straightforward to augment a finite automaton with some type of memory that records subsequences encountered and checks the equality between other subsequences,

[2]The fundamental idea in these approaches bear strong similarities to the work of Woods (1970) on recursive transition networks.

but such augmented models do not share the operational properties of finite-state automata and transducers in general. Cohen-Sygal and Wintner (2006) assure that the formal power of their augmented automata is equal to the formal power of finite automata by limiting the number of registers and the size of their contents.[3] Hence, such augmented automata can be converted into regular finite automata, although it is unclear how this is actually done. However, this means that in such a model one must apparently declare beforehand a maximum word length that one can recognize as reduplicated.

Also, while the simple full reduplication phenomena may be modeled by such means, more complex cases such as partial reduplication and interaction with the rest of the phonology can be difficult. The reason for this is fairly straightforward: even though a formalism allows one to check the equality of substrings, or to copy substrings, in many reduplication phenomena we find that one of the reduplicants is subtly different from the other one due to interaction with the rest of the grammar.

Of the other type of an approach, where one attempts to include reduplication in the usual automaton model without modification, Beesley and Karttunen (2000, 2003) develop a system based on recursive compilation of regular expressions which they call 'compile-replace.' The core idea is to build finite-state automata that contain both strings (as normal) and regular expressions. These regular expressions are then extracted from the automata and compiled in place, and their result is inserted back into the position the regular expression held in the automaton. Thus, one can compile an automaton that describes a finite set of words W, where each word is followed by the string ˆ2, the latter string being a regular expression operator that denotes n-ary, or, in this case 2-ary, concatenation. If 'compile-replace' is run on this automaton, one can produce all words ww in the original automaton. That is, if one has a language W, represented as an automaton, consisting of the strings $\{cat$ˆ2, dogˆ$2\}$, compile-replace will produce another automaton W' that consists

[3] This is very similar to the approach of approximating context-free grammars by simulating a pushdown automaton with a fixed-size stack, where the finite size of the stack guarantees regularity.

of $\{catcat, dogdog\}$. The set of words in W must of course be finite for compile-replace to be able to construct a finite-state machine out of the expression.

There are many subtleties involved in compile-replace and the formalism has surprising expressive power for some applications, to all of which we cannot be do justice here. However, compile-replace is also quite a demanding and complex tool to handle for the designer of a finite-state grammar. Further, as we shall discuss below, it is not immediately obvious how to handle either partial reduplication or cases where copies of reduplicants are not directly adjacent or undergo phonological changes subsequently.

Finally, it is not universally agreed either from a computational or linguistic view that reduplication should be included at all in (computational) morphology. Roark and Sproat (2007), citing arguments from theoretical linguistics due to Inkelas and Zoll (2005), conclude that perhaps reduplication is not part of phonology and morphology at all, and so does not pose a problem to implementations of finite-state systems that handle word formation.

4.4. Reduplication in finite-state systems

Unfortunately, the computational problem of parsing and generating words that feature duplicated elements does not vanish by decreeing reduplication not a part of morphology or phonology proper. It remains the case that one would like to parse and generate all word-forms and continue to have an efficient general model that maps morphological descriptions to their word realizations and vice versa.

Bambara-like reduplication where entire surface forms of words appear as doubled is less problematic and there is no compelling reason to include its treatment within a finite-state system of morphology. It is simple to bypass the entire problem by modifying an otherwise complete finite-state morphological system both in the direction of generation and parsing to handle this kind of phenomenon. For example, one could simply add to the grammar a marker symbol that is understood to match some previous subword in a complex construction. So, the example given in Culy (1985), '*wulunyinina o wulunyinina*' would be generated as '$[_x$*wulu*$]_x[_y$*nyinina*$]_y$ *o XY*'. When generating such forms, one could then, as a last step not performed by transducer, replace the relevant variables with a copy of the marked regions. Likewise, one could do something similar for parsing, the problem of identifying repeated sequences contained in a string such as '*wulunyinina o wulunyinina*' being computationally straightforward. As a preprocessing step before any finite-state processing in the direction of parsing, one could simply identify doubled sequences and map them to variables as in the above, after which one could leave the rest of the task to the finite-state system. In effect, we could map arbitrary-length words such as '*wulunyinina o wulunyinina*' into '$[_x$*wulu*$]_x[_y$*nyinina*$]_y$ *o XY*' before handing over the parsing to the finite-state transducer system. Such an approach would completely separate the finite-state parts of the system and the reduplication-handling parts and, from a practical point of view, may be a reasonable candidate as an approach to handling full reduplication in a finite-state system.

A much more interesting and problematic case of reduplication, and one that perhaps presents an argument for retaining at least some reduplication processing within the finite-state grammar, occurs when reduplicated sequences are not exactly identical in content. The habitual-repetitive reduplication in Javanese seen in (4.6) is of this kind. The suggested strategy of externalizing the reduplication in cases where the base form *bali* is reduplicated to produce *bola-bali*, for example, would not be as straightforward. In such examples, the copies are of course not identical, and have undergone phonological changes which may be quite complex in nature. If we were to externalize the reduplication handling in any way, the phonological processes that operate on reduplicated word forms would also have to be externalized since the task of identifying reduplicants is no longer merely about identifying identical sequences of sounds or symbols. Many of these phonological processes that contribute to producing non-identical reduplicants are of course active also in the language in general and not only in non-reduplicated forms, and so one would be forced to duplicate at least some of the phonology in this hypothetical reduplication-identifying system. Now the simple task of identifying two subwords whose underlying forms may be the same is just as complex a task as parsing any other phonological word.

If we choose to include bounded reduplication in a finite-state grammar, there are potentially two problems involved. The first is largely an empirical one. Given that we accept some redundancy in the finite-state devices that model a reduplicating morphology and phonology, there will naturally be some growth in the state complexity of the system. How large will this growth be?

The second question concerns the descriptive device we want to use to model the reduplication: how do we express the idea that some bounded subwords in a regular language be equal in content and compile this into an automaton or transducer? This is obviously something that the standard battery of regular expressions and automata algorithms is not equipped to handle since the output of such a constraint may be non-regular. Also, this descriptive device should be flexible enough to capture most types of bounded reduplication found in natural language without undue complexity in the grammar. To illustrate this point, let us assume, for example, that we had access to a regular expression operator *Copy(L)*, which, given a finite language L would yield L' such that all words in $L' = ww$ if and only if w is a word in L. Clearly, such an operator would allow for modeling of simple cases such as full reduplication when bounded against a lexicon. But it would not elegantly handle some of the more complex cases already discussed such as partial prosodically governed reduplication in Warlpiri, or the case where phonological material intervenes between the reduplicants, found in Madurese.

4.5. Equality of substrings: the EQ operator

Instead of modeling reduplication by way of string copying, we shall here develop an alternative tool: as the central device to model the types of phenomena already illustrated, a new regular expression operator $EQ()$ is introduced. First, we shall define its operation and illustrate how it can be used in conjunction with composition to yield finite-state grammars that include reduplication. Later, in section 4.8, we shall show how such an operator is implemented in detail.

The operator EQ takes as its input three arguments, τ, $\mathcal{L}$, and $\mathcal{R}$; τ being an arbitrary regular relation (a finite-state transducer), and $\mathcal{L}$ and $\mathcal{R}$ regular languages (finite-state automata). The operation

$$EQ(\tau, \mathcal{L}, \mathcal{R}) \tag{4.8}$$

extracts from τ only those relations where any strings in the range (output) of τ which are surrounded by words from $\mathcal{L}$ and $\mathcal{R}$ are identical. That is, for all strings in range(τ), $lw_1r \ldots lw_nr$, if $l \in \mathcal{L}$ and $r \in \mathcal{R}$, $w_1 = \ldots = w_n$.

We have defined the operator in such a way that $\mathcal{L}$ and $\mathcal{R}$ can be arbitrary regular languages. For our purposes, it will suffice to limit ourselves to the case where they are single symbols: a left and right delimiter. The idea is then to express the constraint that whenever there are several substrings delimited by the left and right delimiters, all of them have to be identical in content.

For example, let τ be the infinite regular language $la^*b^*rl(a \cup b)r$ (or identity transducer). Then $EQ(\tau, l, r)$ consists of the two strings $larlar$ and $lbrlbr$ since all other strings in τ contain unequal content between instances of l and r.

We also consider an expression $EQ(\tau, \mathcal{L}, \mathcal{R})$ to be well-defined if and only if there is no nesting or overlap of $\mathcal{L}$ and $\mathcal{R}$. For example, $EQ(lalarr, l, r)$ is not well-defined.

4.6. Examples of EQ

Let us now turn to actual examples of how to fit the EQ-operator into finite-state morphological grammars that employ the standard composition-model of word-formation. The key to the process is that since EQ operates on transducers, we can enforce equality of delimited substrings at an arbitrary point of our choosing in the derivation of the surface form of a word.

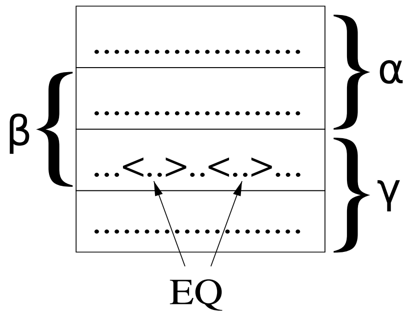


FIGURE 4.1. *Illustration of applying EQ in the middle of a chain of compositions.*

4.6.1. Notation

In what follows, we shall assume more or less the usual finite-state grammar model based composition as seen in figure 4.1, augmented with the EQ operation, which, since it is a function that takes as its primary input a transducer, and returns a transducer, can be thought of as part of a chain of compositions. In a sequence of compositions such as:

$$\alpha \circ \beta \circ \gamma \tag{4.9}$$

if we intend an $EQ()$ operation apply 'after' β (to the output of β and serve as the input to γ), it must be denoted:

$$EQ(\alpha \circ \beta, \mathcal{L}, \mathcal{R}) \circ \gamma \tag{4.10}$$

for some delimiters $\mathcal{L}$ and $\mathcal{R}$. Figure 4.1 illustrates this for the formula (4.10) where the left and right delimiters are assumed to be $<$ and $>$, respectively.

4.6.2. Total reduplication

Let us now illustrate how to use the EQ-operator in actual grammars.

We begin with the simple case of total reduplication using Bahasa Indonesia as an example, although the below description will apply to any instance of repetition compounding. An otherwise complete grammar would likely include (perhaps with more elaborated morphological tags than here) a lexicon of nouns. Presumably there would also be a tag on the lexical side expressing this fact. In effect, the job of the grammar would be to perform mappings such as:

$$\texttt{orang +Noun +Sg} \;\leftrightarrow\; \texttt{orang} \tag{4.11}$$

But we would also like to include the mapping:

$$\texttt{orang +Noun +Pl} \;\leftrightarrow\; \texttt{orang-orang} \tag{4.12}$$

without separately listing the reduplicated variant as a lexical entry. One way to produce such a grammar in a serial, composition-oriented way, would be to start with a lexicon which would include single nouns. From this lexicon, we can produce an augmented lexicon that includes *noun* + *noun* combinations, where each noun would be surrounded with delimiting brackets. This *noun* + *noun* combination would be produced from the original single-entry lexicon by the composition of the lexical transducer with a simple transducer that inserts a member of the lexicon following the original member. Naturally, at this stage, the two nouns in the *noun* + *noun* combination could be unequal. At this point, one would apply EQ to the lexicon and remove the auxiliary brackets. In effect, we would have:

$$\begin{aligned} &L_n = \texttt{orang}\ldots \\ &\tau_{lexicon} = (L_n\texttt{+Noun +Sg }) \cup ([\epsilon{:}\texttt{<}]L_n[\epsilon{:}\texttt{>}][\epsilon{:}\texttt{<}][\epsilon{:}L_n][\epsilon{:}\texttt{>}]\texttt{+Noun +Pl}) \\ &\tau_{removediacritics} = (\texttt{<} \cup \texttt{>} \cup \texttt{+Noun} \cup \texttt{+Sg} \cup \texttt{+Pl}) \rightarrow \epsilon \end{aligned} \tag{4.13}$$

and the entire process could be represented as:

$$EQ(\tau_{lexicon}, <, >) \circ \tau_{removediacritics} \tag{4.14}$$

Illustrated serially, in (4.14) we would have intermediate representations such as:

1. `orang +Noun +Pl`	*input side of* $\tau_{lexicon}$	
2. `orang<X> +Noun +Pl`	*output side of* $\tau_{lexicon}$, *X = any noun*	(4.15)
3. `orang<orang> +Noun +Pl`	*after* EQ	
4. `orangorang`	*after* $\tau_{removediacritics}$	

In step 2, which represents the output side of $\tau_{lexicon}$, we have any (possibly unequal) *noun-noun* combination, the non-matching ones being later filtered out by EQ. In practice it is more efficient for compilation not to restrict the second copy to just the nouns, but to any sequence of symbols not containing the markers $<$ or $>$, i.e. $(\Sigma - < - >)^*$. The subsequent equality operator will of course filter out non-identical sequences.

4.6.3. Partial reduplication

4.6.3.1. Copying prosodically specified templates

Turning to a more complex example, let us see how patterns in the partial reduplication in Warlpiri can be modeled by applying EQ in an appropriate way.

In the Warlpiri examples above in (4.5), the copied material consists of a skeleton $CV(C)(C)V$ pattern, the Cs in parentheses being optional. For a given base, we should therefore also allow a prefixed copy of the skeleton part of the base. Since reduplication in Warlpiri signals a number of different grammatical distinctions, we shall not be concerned with the specific tags that might be used in conjunction with reduplication, but restrict ourselves to the mechanism by which, given a base word w in a lexicon we can automatically produce pw where p matches w in the initial $CV(C)(C)V$ pattern.

This can be done in two steps: first we create a transducer τ_{markp} that surrounds the relevant part—$CV(C)(C)V$—of the base word with auxiliary brackets, $<$, and $>$, which we the compose with another transducer $\tau_{insertp}$ which optionally prefixes a $< X >$ sequence, where X is any word over the alphabet except those that contain $<$ or $>$. Clearly, these transducers can be created through standard techniques to construct transducers that encode rewrite rules. After composing the lexicon with the transducers specified above, we apply EQ. In other words:

$$EQ(\tau_{lexicon} \circ \tau_{markp} \circ \tau_{insertp}, <, >) \tag{4.16}$$

producing sequences such as:

1. `tiirlparnkaja`	*output side of* $\tau_{lexicon}$	(4.17)
2. `<tii>rlparnkaja`	*after* τ_{markp}	
3. `<X><tii>rlparnkaja`	*after* $\tau_{insertp}$*, X = any sequence*	
4. `<tii><tii>rlparnkaja`	*after* EQ	

4.6.3.2. Copying in conjunction with phonological change

A similar strategy can be used to capture the examples concerning habitual-repetitive reduplication in Javanese in (4.6), encoding the crucial fact that both copies may undergo phonological change after the reduplication has occurred.

A generalization that captures the fragment of the data on page 109 is that the last vowel in the left copy of the final syllable must be `a`, and so is changed into an `a`. Further, if the first vowel of the stem was already `a`, that vowel is raised, i.e. `a` → `o`, and `ɛ` → `e`. Here, using EQ together with composition, the approach is to first copy the stem and surround the copy with the brackets $<$ and $>$, then apply EQ, and compose this with a transducer representing the vowel changing rule. In actuality, we of course do not copy the stem, but rather optionally prefix the lexical items with the sequence $< X >$ where X is any word over the alphabet, except those containing the symbols $<$ or $>$. Call the transducer that performs this τ_{habrep}. The subsequent vowel changes can be modeled by some transducer $\tau_{vowelmelody}$.

That is, the entire habitual-repetitive operation can be described as follows, using the stem `adus` as an example:

$$EQ(\tau_{lexicon} \circ \tau_{habrep}, <, >) \circ \tau_{vowelmelody} \tag{4.18}$$

1. `adus +HabRep`	*output side of* $\tau_{lexicon}$	
2. `<X><adus> +HabRep`	*after* τ_{habrep}	(4.19)
3. `<adus><adus> +HabRep`	*after* EQ	
4. `<odas><adus> +HabRep`	*after* $\tau_{vowelmelody}$	

4.6.3.3. Copying and overapplication of phonological generalizations

Another, perhaps linguistically more interesting example is that modeling the reduplication in Malay in combination with a certain type of nasal harmony. There is a general phonological regularity at work in Malay whereby oral and nasal semivowels or vowels harmonize rightward in words. That is, vowels that follow a nasal consonant are all nasalized, as can be seen in the words hamə̃, waŋĩ, aŋãn, and aŋẽn. However, when reduplicated, all vowels become nasal, i.e. the above stems become hãmə̃-hãmə̃, wãŋĩ-w̃ãŋĩ, ãŋãn-ãŋãn, and ãŋẽn-ãŋẽn respectively. This is sometimes called 'overapplication' in the phonological literature (Wilbur, 1973; Marantz, 1982). The reason is fairly obvious: the nasal spreading—an independent phenomenon that usually only works in left-to-right fashion—seems to apply in an environment where it is not warranted, i.e. the left copy of reduplicated words.

There are two obvious strategies available to handle this in a finite-state grammar using EQ. The first is to simply model one rewrite rule that changes vowels that follow nasal consonants into their nasal counterparts, and to have a separate rule that applies to reduplicated forms only, that changes all vowels into nasal vowels in the presence of a single nasal consonant in the entire word. Clearly, this would capture the data and the generalization at work. However, a more intriguing possibility, and one that relies on a single generalization of the left-to-right harmony is to have EQ apply, not directly after the reduplication segments have been marked, but only after the rightward harmony has been applied. Such an approach requires no special phonology that only applies to reduplicated forms and captures the 'overapplication' of the harmony rule easily.

Assume we have a transducer $\tau_{nasalharmony}$ that changes vowels into nasal vowel whenever a nasal consonant occurs earlier in the word. And similarly to the previous examples, assume also we have another transducer τ_{red} which handles the optional prefixing of a $< X >$ sequence and surrounding the base with $<$ and $>$ markers that handles the reduplication in conjunction with EQ. Now, the following sequence of applying EQ and composition of the nasal harmony transducers will yield the desired result:

$$EQ(\tau_{lexicon} \circ \tau_{red} \circ \tau_{nasalharmony}, <, >) \tag{4.20}$$

The logic of having EQ apply after nasal harmony is seen in the following sketch of one of the possible intermediate stages of the reduplication of `hamə̃`:

1. `hamə`	*output side of* $\tau_{lexicon}$	
2. `<hãmə̃><hamə>`	*after* τ_{red}	(4.21)
3. `<hãmə̃><hãmə̃>`	*after* $\tau_{nasalharmony}$	
4. `<hãmə̃><hãmə̃>`	*after* EQ	

Note that if we break the chain of composition down like this into imaginary intermediate forms, that after τ_{red} (step 2), the lower-side language will contain an infinite number of possible words as the left reduplicant. In the above, we have illustrated one of the two possibilities that will yield a valid form in combination with `hamə` as the base. The other one is `hãmə`: both of these will produce the same form in step 4 which is not filtered out by EQ. An output such as `<hamə><hamə>` from τ_{red} will not produce any forms at all after EQ applies, because after $\tau_{nasalharmony}$ has applied the form changes into `<hamə̃><hãmə̃>`, which will not 'survive' EQ.[4]

[4]The fact that we can through string rewriting and the EQ operation model this phenomenon may be of more theoretical interest as well. The example from Malay is often used as a linchpin in arguments for or against some theory of phonology. For instance, Kager (1999) and others have concluded that modeling the Malay phenomenon with linearly ordered rewrite rules empowered with an additional 'copying' mechanism would require the copy mechanism to apply twice (which is deemed inelegant): once to produce the copy, and once again after the nasal harmony rule has applied. As seen in the above, if we forgo a 'copying' rule and instead rely on a combination of an equivalence constraint such as EQ and some epenthetic segments (the markers), no copying needs to occur twice. Similar arguments are presented for Paamese in Russell (1997): "The classical approach to phonology and morphology does not have any elegant and convincing way of doing this. The difficulty here is caused by the theory's requirement that the copying operation be performed at a particular point in the derivation, either before or after the phonological rules." (p. 110). The examples alluded to in the above are also solvable with a single application of EQ at one point in the 'derivation.'

Of course, from the point of view of merely having to construct a transducer which models the phenomenon, it does not really matter which method one chooses. The latter may be more elegant from a phonological perspective as we retain nasal harmony as an independent process and the quirks of nasalization in reduplicated forms fall out of the interaction between EQ and the nasal rightward spreading—without having to postulate that nasal harmony 'overapplies' in a certain environment.

4.6.3.4. Disjoint copies and dependencies

There is naturally no restriction that the parts of a string to be asserted equal in content with EQ be contiguous. In the Madurese reduplication examples (4.4) the final $CV(C)$ of a stem is prefixed to itself (Stevens, 1985). Using similar rules as in the above, we can model this by composing a set of rules that insert any $CV(C)$ combination to the stem, surrounded by delimiters, and mark the final $CV(C)$ of the stem with delimiters also (call this transducer τ_{red}) after which EQ applies, yielding sequences such as:

1. `garadus`	*output side of* $\tau_{lexicon}$	
2. `<X>gara<dus>`	*after* τ_{red}, X = any CV(C)	(4.22)
3. `<dus>gara<dus>`	*after* EQ	

It should be noted that Madurese exhibits a very similar interaction between nasal and oral vowels as the Malay (Stevens, 1985). The above example does not illustrate this, but cases where this 'phonological overapplication' comes into play can be handled exactly as in the Malay example.

4.6.3.5. Multiple copies and cross-linked dependencies

Even more complicated patterns can be captured with the EQ operation. The crossing partial reduplications of Coeur d'Alene as reported by Reichard (1938) may require that we employ several different delimiters, resorting to EQ on multiple occasions. Consider the form where the base `caq` and the morpheme `ip` are reduplicated multiple times to yield caqcaqaqɛlipəp. Such exotic examples are clearly far more complex than the previous ones presented. Here, the morphemes caq and ip occur both as reduplicated, with the entire syllable of the first one copied once, and the second time lacking an onset. Without going into a detailed analysis (as the exact grammatical functions remain unclear in the Coeur d'Alene documentation), it can be gleaned that the separated dependencies can be divided into:

$$\texttt{<c[aq]><c[aq]>[aq]ɛl(ip)(əp)} \tag{4.23}$$

That is, if the Coeur d'Alene examples present productive processes, in modeling them we may need to employ several instances of EQ to yield the correct form: once for the bases, such as caq and once for its onsetless copy, and once for the morpheme of the type ip. Note the actual realization of ip is əp, i.e. a separate phonological rule will apply subsequent to the instance where EQ pertains to the (and) delimiters.

4.7. Additional applications

The potential applications of the $EQ()$ operation are not restricted to modeling reduplication. There exist a number of uses for such an addition to the existing regular operations, of which we shall briefly discuss two cases: string rewriting rules that copy individual segments and a straightforward generalization of this that allows us to include backreferencing in string rewriting rules.

4.7.1. Segment copying

With the formalisms for string rewriting through finite-state transducers (see chapter 8), it is difficult to express the notion of single segment copying—a circumstance often found in models of phonology and morphology. Suppose, for instance, that we wanted to double a consonant in some environment, it would be difficult to express in a simple way through direct application of the rewrite formalisms developed in chapter 8. A simple solution, through inelegant, is to employ a set of rules, such as:

$$b \to bb, c \to cc, d \to dd, \ldots \tag{4.24}$$

With the $EQ()$-operator, such a batch of rules can be expressed much more concisely by inserting any arbitrary segment in the desired location, and also inserting appropriate delimiters to mark which segments need to be identical, and subsequently applying EQ. For example:

$$EQ(C \to < \ldots >< C >, <, >) \circ (< \cup >) \to \epsilon \tag{4.25}$$

where C represents the set of consonants, will perform the task in (4.24), although with a much more compact statement.

4.7.2. Backreferencing in regular expressions

The same approach that is used in the above to perform segment copying can be generalized to produce a form of backreferencing in regular expressions. The regular expressions used in many practical applications and programming languages (Perl, Python, Emacs, Ruby, flex, etc.) are often augmented with a backreference operator, where in a regular expression, one can refer to something that has occurred earlier in a string, not entirely unlike the logic of $EQ()$. By including such subexpressions, one can easily a describe a class of languages that are no longer regular, and thus not compilable into automata. For example, the simple non-regular (and non-context-free) ‘copy’ language $L = \{ww \mid w \in \{a,b\}^*\}$ is captured by the backreferencing Perl regular expression $(a \mid b) * \backslash 1$.[5] Our intent here is of course not to venture beyond the domain of the regular languages since by doing so we would

[5]For an interesting analysis of the formal power of these extended regular expressions found in various programming languages, the reader is directed to Câmpeanu et al. (2003).

sacrifice the closure properties we enjoy when staying within the class of regular languages and relations.

But backreferencing, like string equality, is of course regular when applied to a finite set of strings, or strings of finite length. The strategy we used in (4.25) to achieve single-symbol copying can be generalized in a number of ways to yield limited backreferencing, or backreferencing-empowered string rewriting rules.

A simple example is the above copy language in $L = \{ww \mid w \in \{a,b\}^*\}$, which of course can only be approximated for strings of some length $\leq k$, as follows:

$$EQ(< (a \cup b)^{\leq k} >< (a \cup b)^* >, <, >) \tag{4.26}$$

Naturally, in the above $EQ()$-statement we have needed to use the usual set of delimiting brackets to provide the EQ algorithm with an anchor for the substrings whose equality we want to enforce. This means that the resulting language will also contain the delimiters. But these can be removed through a simple string homomorphism (or string rewriting) to provide the desired result:

$$h_{\{<,>\}\to\epsilon}\big(EQ(< (a \cup b)^{\leq k} >< (a \cup b)^* >, <, >)\big) \tag{4.27}$$

Generalizing this idea, it's possible to build this extension into a regular expression formalism itself. By first declaring a set of special symbols $\{<_1, >_1, \ldots, <_n, >_n\} \notin \Sigma$ we can obviously compile a standard regular expression R containing such symbols using the expression:

$$R' = h_{\{<_1,>_1,\ldots,<_n,>_n\}\to\epsilon}\big(EQ(\ldots EQ(R, <_1, >_n) \ldots, <_n, >_n)\big) \tag{4.28}$$

with the natural caveat that the results be regular, i.e. that for all subexpressions delimited by some $<_i, \ldots, >_i$, if there are more than two of them, their intersection be finite.[6]

[6]Conditions on when an $EQ()$-expressions is possible to represent as a finite-state automaton are discussed below in section 4.8.2.3.

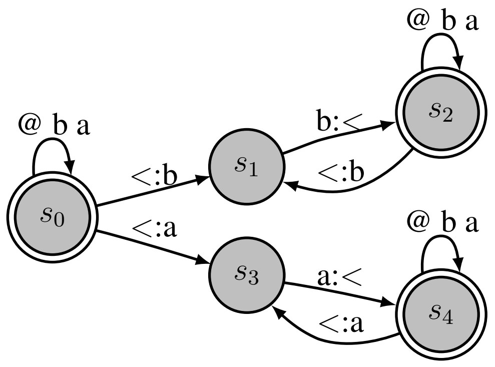


FIGURE 4.2. *Example transducer τ_{move} created by Algorithm EQ at lines 10–12 over the alphabet* $\{a, b\}$.

Assuming such a set of special symbols were made available, one could then model the copy language over $\{a, b\}^*$ for finite-length strings directly by the expression:

$$<_1 (a \cup b)^{\leq k} >_1 <_1 \Sigma^* >_1 \tag{4.29}$$

4.8. The EQ algorithm

Let us now turn to the actual algorithm by which transducers that encode a constraint $EQ()$ are built. The approach taken here is to build a set of intermediate transducers from the input arguments τ, $\mathcal{L}$, and $\mathcal{R}$ which through a series of compositions yield the desired transducer τ'. It should be noted that the input transducer τ may be a regular language T and not necessarily a relation, in which case it is considered the identity relation $Id(T)$ and the output τ' will also be a language instead of a relation.

4.8.1. Overview

The fundamental idea behind the algorithm is to iteratively refine the original transducer τ in such a way that words occurring in the output of τ between instances of $\mathcal{L}$ and $\mathcal{R}$

Algorithm 4.1: EQ(τ,$\mathcal{L}$,$\mathcal{R}$)

```
   Input: $\tau$, $\mathcal{L}$, $\mathcal{R}$
   Output: $\tau'$
1  begin
2      $\tau_{bracketed} = \tau \circ (\neg(\Sigma^*(\mathcal{L} \cup \mathcal{R})\Sigma^*)(\mathcal{L}\ [\epsilon{:}{<}]\ \cup\ [\epsilon{:}{>}]\ \mathcal{R}))^* \neg(\Sigma^*(\mathcal{L} \cup \mathcal{R})\Sigma^*)$
3      $L_{nobr} = \neg(\Sigma^*(<\ \cup\ >)\Sigma^*)$
4      $L_{brf} = L_{nobr}\ <\ L_{nobr}\ >\ L_{nobr}\ (<\ L_{nobr}\ >\ L_{nobr})^+$
5      $\tau_{removebr} = (L_{nobr} \cup [< :\epsilon] \cup [> :\epsilon])^*$
6      $\tau_{eq} = \tau_{bracketed} \circ L_{brf}$
7      $\tau_{bypass} = \tau_{bracketed} \circ \neg(L_{brf}) \circ \tau_{removebr}$
8      $\tau_{nonb} = \tau_{eq} \circ ([(\Sigma- <){:}\epsilon]^*[< :\epsilon](\Sigma- >)^*[> :\epsilon])^*[(\Sigma- <) : \epsilon]^*$
9      $\tau_{move} = \emptyset$
10     foreach symbol $\alpha$ occurring in $(\tau_{nonb})_2$ do
11         $\tau_{move} = \tau_{move} \cup ((\Sigma- <)^*[< :\alpha][\alpha{:}{<}])^*(\Sigma- <)^*$
12     end
13     $\tau_{check} = (\Sigma- <)^*([< :\epsilon][> :\epsilon](\Sigma- <)^*)^* \cup \neg(\Sigma^*\ <\ >\ \Sigma^*)$
14     while TRUE do
15         $\tau_{eq} = \tau_{eq} \circ \tau_{check}$
16         if the symbol $<$ does not occur in $(\tau_{eq})_2$ then
17             break
18         end
19         $\tau_{eq} = \tau_{eq} \circ \tau_{move}$
20     end
21     $\tau' = \tau \circ \tau_{eq} \cup \tau_{bypass}$
22 end
```

are identical. To accomplish this, we first split the original transducer τ into two disjoint parts: τ_{eq} and τ_{bypass}. The transducer τ_{bypass} contains all such words we can disregard in the algorithm; words that have improperly nested instances of $\mathcal{L}$ and $\mathcal{R}$, or words that have less than two instances of $\mathcal{L}$ and $\mathcal{R}$. Obviously, if there is only one instance of the delimiting languages, we need not assure that any content between instances be equal. Now, having divided the original transducer τ into τ_{eq} and τ_{bypass} we proceed to refine τ_{eq} until all possible words occurring between $\mathcal{L}$ and $\mathcal{R}$ are identical in content. The resulting transducer τ' is obtained by composing τ with the union of the ranges of τ_{bypass} and τ_{eq}.

In the preparatory operations of the algorithm, lines 2–7 capture the method of splitting τ into τ_{eq} and τ_{bypass}. This is done by first inserting two auxiliary bracket symbols, $<$ and $>$ between every occurrence of $\mathcal{L}$ and $\mathcal{R}$ in τ on line 2. Line 6 separates from τ the properly bracketed sequences defined on lines 3 and 4. Line 7 creates the τ_{bypass} transducer in a similar way, and also removes the auxiliary symbols from it.

Lines 9–12 construct an auxiliary transducer τ_{move}, which operates as follows: for every symbol α occurring between bracketed sections in the output of τ_{eq}, the two-symbol sequence $< \alpha$ is replaced with $\alpha <$. The transducer τ_{move} also rejects all strings where there exists a sequence $< \alpha_1 \ldots < \alpha_2$, if $\alpha_1 \neq \alpha_2$.

Lines 14–20 iteratively compose τ_{eq} with τ_{move} until no brackets remain in the output of τ_{eq}. Between every composition of τ_{eq} with τ_{move}, any sequences consisting of a single pair of brackets $< >$ are removed. Also, any strings containing both a single pair of brackets $< >$ and any non-empty bracketed sequence $< X >$ are removed. The latter operation is accomplished by a composition of τ_{eq} with the τ_{check} transducer defined on line 13.

To illustrate a run through the algorithm, consider a transducer τ that consists of two strings mapped to themselves, $laarlabr$ and $labrlabr$, and let $\mathcal{L} = l$ and $\mathcal{R} = r$. Clearly, now, $EQ(\tau, \mathcal{L}, \mathcal{R})$ should yield the single string $labrlabr$, since the other string contains non-identical material between occurrences of l and r.

Now, τ_{eq} as constructed between lines 2–7 consists of the two bracketed strings

$$l < aa > rl < ab > r$$

and

$$l < ab > rl < ab > r$$

The transducer τ_{move} needed for the iterative refinement of τ_{eq} is constructed for the alphabet $\{a, b\}$ on lines 9–12 and is seen in figure 4.2. Note that we need not include symbols l and r in the construction of τ_{move} since these are never found inside properly bracketed strings.

The first composition of τ_{eq} with τ_{move} on line 19 changes the output of τ_{eq} to the two strings *la<a>rla<b>r* and *la<b>rla<b>r*, i.e. moves our auxiliary bracket one symbol to the right. Now, the second time the composition is done, only the string *lab<>rlab<>r* remains, since the transducer τ_{move} does not accept as input the string *la<a>rla<b>r*. After the second application, the composition with τ_{check} on line 15 removes the brackets from the string *lab<>rlab<>r*, and the loop terminates at the check on line 16 since no brackets remain in the output of τ_{eq}, which consists of the single string *labrlabr*.

4.8.2. Analysis

4.8.2.1. Termination

Obviously, since the algorithm performs one iteration (movement to the right) for every symbol inside bracketed sequences in lines 14–20, it terminates if τ only contains finitely long strings between all occurrences of $\mathcal{L}$ and $\mathcal{R}$. It does not, however, terminate if τ contains infinitely long strings that agree in their prefixes between occurrences of $\mathcal{L}$ and $\mathcal{R}$. For example, a simple expression such as:

$$EQ(la^*rla^*r, l, r) \tag{4.30}$$

will fail to terminate. It should evaluate to the set $lrlr$, $larlar$, $laarlaar, \ldots$, i.e. , la^nrla^nr which is not a regular language. However, it is not the case that EQ terminates for all regular languages and relations, either. Indeed, if the algorithm terminates, the language or relation $EQ(\tau, \mathcal{L}, \mathcal{R})$ is regular, but the converse need not be true. The expression

$$EQ(la^*brla^*cr, l, r) \tag{4.31}$$

should evaluate to $\emptyset$, but the algorithm does not terminate because of the potentially infinite-length prefix a^* for which both delimited substrings are identical.

However, there are cases where even if we have a potentially infinite number of bracketed sequences consisting of infinitely long strings, so long as the length of potentially identical substrings is finite, the algorithm terminates. For example, EQ terminates for the expression:

$$EQ((la^*brlab^*r)^*, l, r) \tag{4.32}$$

and yields the language $(labrlabr)^*$.

Naturally, and most importantly, so long as one delimited sequence for all words is a finite language, the algorithm terminates. For natural language applications this is the expected case.

Finding a suitable termination condition that avoids an infinite loop is an interesting problem in itself, altough its practicality can certainly be questioned. It seems unlikely that a simple pre-analysis of τ which would yield a termination condition would be efficient enough to warrant its inclusion in practical implementations of the algorithm.

For cases where it can be ascertained that τ consists either of exactly n sequences between $\mathcal{L}$ and $\mathcal{R}$, or none at all, one can simply extract $L_1 \ldots L_n$—the languages occurring between the delimiting languages—and check whether their intersection yields a finite language, in which case EQ is guaranteed to terminate and otherwise not. In (4.30), for

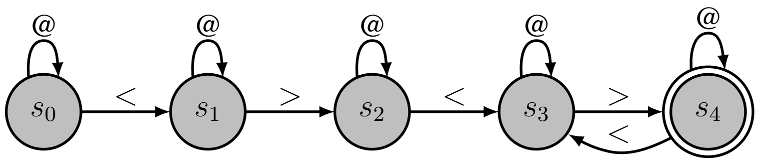


FIGURE 4.3. *The automaton L_{brf} used in Algorithm EQ.*

example, where $L_1 = L_2 = a^*$ and $L_1 \cap L_2 = a^*$, it can be seen that EQ is nonregular and will not terminate.

4.8.2.2. Efficiency

For cases where the algorithm does terminate, it is of interest to know its asymptotic behavior, particularly with respect to the size of the resulting automaton or transducer.

First, it can be observed that the sizes of $|\tau_{bypass}|$ and $|\tau_{eq}|$ are bounded by a small constant c with respect to the size of τ, as seen from the relatively small automata τ is composed with to produce the two languages τ_{bypass} and τ_{eq}. In fact, L_{brf} is represented by the 5-state automaton in figure 4.3.

Now, the size of τ_{move} is easily seen to be $2n+1$ states, if $n > 1$, where n is the number of different symbols occurring inside bracketed sequences. Naturally, this is $|\Sigma_\tau|$ in the worst case.

As we perform the iterative composition of τ_{eq} and τ_{move}, the size of each new τ_{eq} is bounded by $|\tau_{move}||\tau_{eq}|$.

Since we perform the composition k times in lines 14–20, we can see this yields an overall size bound of $O(|\Sigma_\tau|^k|\tau|)$, where k is the length of the longest instance of identical substrings.

It is unlikely, however, that the bound is reached in practical applications, where only a fraction of the possible words in $|\Sigma^n|$ where $0 < n < k$ are actually found between instances of $\mathcal{L}$ and $\mathcal{R}$.[7]

From this perspective, the algorithm is appealing in that the growth of τ_{eq} during each iteration is bounded by the number of actually occurring symbols α as substrings with the configuration $< \alpha$.

4.8.2.3. Decidability of the regularity of $EQ(\tau, \mathcal{L}, \mathcal{R})$

Let us return briefly to the question of knowing beforehand whether algorithm EQ terminates or not. It is often expected that a language which is a superset of the regular languages has the property that it is not in general decidable whether a particular instance of it can be expressed as a regular language. This is the case, for instance, with the context-free languages (Hopcroft and Ullman, 1979). With this in mind, it may of interest to note that the question of the regularity of the relation represented by $EQ(\tau, \mathcal{L}, \mathcal{R})$ is decidable as follows.

Consider τ_{eq} in algorithm 4.1. Obviously its range contains all the words of interest occurring between $\mathcal{L}$ and $\mathcal{R}$. From this, we can extract only the sequences enclosed in brackets and construct a reduced automaton X' containing these words. That is, X' consists exclusively of words of the form $< \Sigma^* > (< \Sigma^* >)^+$. From the minimal automaton representing X' construct X'' in such a way that each bracket symbol also contains an index that describes the target state for each transition marked $<$ or $>$. Clearly, this can be done by a simple inspection of X' and adding new symbols $<_i$ and $>_j$ for each transition $\delta(<, i, j)$ and $\delta(>, i, j)$, and replacing the original transitions with $\delta(<_j, i, j)$ and $\delta(>_j, i, j)$. Now,

[7]From a natural language processing perspective, the following figures may give some indication of the expected growth of an automaton that handles reduplication with EQ. We created from an English lexicon L_1 of $36,871$ words represented as an automaton of $30,668$ states the new language $L_2 = L_1(-L_1)$. This is the language where each word from L_1 could appear on its own, or as an entirely reduplicated word. Hence, if $word$ was in L_1, both $word$ and $word - word$ would be in L_2. The resulting lexicon then consisted of $73,472$ words encoded as an automaton of $299,793$ states, a roughly tenfold increase in the number of states. The size of the alphabet was 39 symbols and the length of the longest string 47 letters.

we can easily extract every possible combination of $<_p$ and $>_q$ for all p and q occurring in a single word, say by intersecting X'' with the language

$$\bigcap_{i=0}^{n}\bigcap_{j=0}^{n}\neg(\Sigma^* <_i \Sigma^* >_j \Sigma^* <_i \Sigma^* >_j \Sigma^*)$$

yielding all possible sequences of $<_i$ and $>_j$ without repetitions.

We can also extract from X'' the possible languages occurring between $<_p$ and $>_q$ for all p and q, by changing the initial state in X'' to p and making q the only final state. Call these languages L^p_q.

Now, all the possible nonrepeating sequences of L^p_q that can occur as words between brackets in X'' are of a finite number. For each such sequence, we can perform the intersection:

$$(L^{p_1}_{q_1} \cap \ldots \cap L^{p_n}_{q_n}) \tag{4.33}$$

And, if all intersections yield a finite language, $EQ(\tau, \mathcal{L}, \mathcal{R})$ is regular, otherwise not. Clearly, if there exists a sequence $L^{p_1}_{q_1} \cap \ldots \cap L^{p_n}_{q_n}$ such that the intersection of the individual languages is not finite, there exist two equal subwords in $EQ(\tau, \mathcal{L}, \mathcal{R})$ of arbitrary length within delimiters. Conversely, if the intersection is finite, no such word exists, and $EQ(\tau, \mathcal{L}, \mathcal{R})$ is regular. Whether each intersection in (4.33) yields a finite language can be checked by a standard depth-first-search on each resulting automaton.

We have thus established that it is decidable whether $EQ(\tau, \mathcal{L}, \mathcal{R})$ can be represented as a regular language. However, as was noted above in example (4.31) the operation of the algorithm is such that it also fails to terminate if any equal prefix is of arbitrary length. To ascertain whether the algorithm terminates it is therefore not sufficient to examine the potential regularity of the EQ-expression at hand by observing the set of co-occurring subwords L^i_j, but rather, the set of prefixes of the subwords L^i_j. Since the set of prefixes of any regular language L is also regular, we can obviously modify the above method to also decide whether a particular run of EQ will terminate.

4.9. Discussion

This chapter has presented a formalism by which closed-lexicon reduplication can be incorporated into the classical workflow of constructing lexical transducers. The EQ-algorithm is a simple operation that works on finite-state transducers and automata and can be seen as a filter that removes parts of marked strings that are unequal in content. The sections of a string which are to be marked, and thus subject to the equality constraint, can be flexibly defined using other regular relation operations.

Because EQ is defined to operate on transducers, where the equality constraint applies to the output side, the operation can naturally be incorporated into a chain of compositions. A number of examples from various languages illustrate the usefulness of being able to do so: in many cases of reduplication, the relationship between underlying and surface forms of a word can be conveniently modeled in such a way that the equality constraint applies at a very particular point in a sequence of phonological alternations. The EQ-operator can also be used to express other, simpler tasks: given a set of words in a lexicon automaton L, we can easily produce a new automaton L_r that contains reduplicated words. Even such a relatively simple operation is very difficult to define using other regular expression construction techniques.

One of the primary motivations for introducing the EQ operator in handling closed-lexicon reduplication was that the other simple alternative—to add non-finite-state preprocessing to parsing and generation—would make it difficult to capture phenomena where one of the reduplicants is different from the other one. The EQ operator handles such cases elegantly since it builds an equivalence constraint into the overall phonological system and this equivalence need not necessarily manifest itself in the surface forms as it may have applied to an intermediate form. Also, EQ is not strictly limited to closed lexicon reduplication: what is required for EQ to be incorporable into a finite-state grammar is that the set of the possible reduplicants be finite. In the Warlpiri examples, for instance, the reduplicants are always of the form $CV(C)(C)V$, and hence constitute a finite set. This makes it possible to parse and generate out-of-vocabulary items correctly within the general finite-state system without pre-specifying an upper bound on the length of words in the lexicon. If one on the other hand wants to model truly unbounded reduplication and incorporate this into a finite-state system, EQ will naturally not suffice, and alternatives such as preprocessing must be considered.

5. PROPERTIES AND DECISION ALGORITHMS FOR REGULAR LANGUAGES AND RELATIONS

5.1. Introduction

In this chapter we will look at closure properties and decision algorithms pertaining to finite-state devices. In particular, we will focus on questions that are of interest to natural language applications.

We shall almost exclusively focus on finite-state transducers (as opposed to automata) and certain subclasses of them. Since almost every imaginable property of finite-state automata is decidable—quite efficiently and easily—we shall say very little about automata here.

For finite-state transducers, however, the situation is different. As mentioned in chapter 3 our general model of regular relations represented by a finite-state transducer is not closed under the boolean operations of complement and intersection, and hence not subtraction. In addition to this, there are a number of properties of finite-state transducers that would be of interest for linguistic modeling that are not solvable by algorithms. But there are also many properties which do have effective algorithms—distinguishing between the two cases can be very difficult and often involves subtle analysis of the properties of finite-state transducers. It is easy, for example, to determine if two finite-state automata describe the same language: since there exists a unique minimal representation for each automaton, two automata can be minimized and compared for equality. No such representation exists for finite-state transducers, and many of the questions that have simple answers in the automata case turn out to be undecidable, or very difficult to answer in the transducer case.

As before, we shall assume that our finite-state transducers usually represent some linguistic model. Given two grammars G_1 and G_2, represented as finite-state transducers τ_1 and τ_2, some of the immediately interesting questions to ask are:

(i) is the relation τ_1 equivalent to τ_2?

(ii) is $\tau_1 \subseteq \tau_2$ or vice versa?

(iii) for which inputs W does τ_1 output a word different from τ_2?

(iv) for which inputs W does τ_1 output the same as τ_2 does?

Questions (i)–(iv) are very natural questions to ask both from a language engineering and from a linguistic point of view.

As we saw in chapter 1, many morphological or phonological theoretical mechanisms can fairly accurately be represented as finite-state transducers: this includes SPE-style rewriting grammars as modeled by Kaplan and Kay (1994), the Lexical Phonology and Morphology of Mohanan (1986), Optimality Theory (as shown in Karttunen (1998) and Gerdmann and van Noord (2000)), and realizational morphology as presented in Stump (2001) and modeled by Karttunen (2003), among others. In this context, the previous questions become of interest for reasons of research in phonology and morphology: it is often quite difficult to compare predictions of two different theories given in two different formalisms. The possibility of representing a theory in a fairly neutral way—a finite-state transducer—offers a method to answer a number of important questions.

Supposing one has a description of some linguistic phenomenon in formalism A and another description modeling the same phenomenon in formalism B, it would be of interest to know if the two proposals make the same predictions and are equivalent. If we could convert both descriptions to two finite-state transducers, the question would correspond to question (i). Additionally, given the same scenario, and supposing A is indeed known to differ from B, one would perhaps like to know exhaustively all the cases in which the grammars yield different predictions—the cases in question may of course not be a finite set. This question could then be answered by solving (iii). Question (iv) is of interest for similar reasons. Question (ii) is perhaps relevant from the point of view that some grammars tend to 'overgenerate' but could otherwise be correct in their predictions. This would be the case if one grammar is a subset of the other.

Apart from these decidability questions regarding the relationship of two finite-state transducers to each other, there are a number of further questions that are relevant in the context of a single arbitrary finite-state transducer τ.

Given a grammar G represented as a finite-state transducer τ,

(v) does τ ever yield two or more distinct outputs for one input w?

(vi) in τ, do we find two distinct paths through the transducer for one input w?

(vii) does τ always yield the same output as the input? That is, is it an identity relation only?

These questions also naturally prompt the subsequent follow-up questions:

(viii) Given that (v) is true for some τ, for which inputs does τ yield more than one output?

(ix) Given that (vi) is true for some τ, for which inputs does τ have multiple paths (is ambiguous)?

(x) Given some non-functional τ, can we produce a transducer τ' which allows only the functional inputs?

(xi) Given some ambiguous τ, can we produce a transducer τ' which allows only the unambiguous inputs?

Questions (v) and (viii) would effectively give answers about the type of ambiguity expressed by a grammar, in the direction of either generation or parsing.[1]

[1]For all of the properties, we assume that if one is interested in, say, ambiguity in parsing instead of generation, it is possible to invert the transducer in question and attempt to solve the problem at hand for τ^{-1} instead of τ.

Question (vii) is non-trivial despite appearances: there are an infinite number of ways in which an arbitrary identity relation can be encoded by distributing one-sided ϵ-transitions in a finite-state transducer.

The chapter will treat the above questions as follows: first we will address questions (i)—(iv). These are classical decidability results and we will not treat them in detail except when our proof or construction method is different and elucidates the subject matter in particular, or simplifies the demonstration of some subsequent, new result. Question (v) has been shown to be decidable by Schützenberger (1976) and Blattner and Head (1977), although no explicit algorithm was given. We shall present an efficient $O(n^2)$ algorithm for solving the question, which crucially also hinges on our algorithm for solving question (vii).

Question (viii) will turn out to be undecidable. Remarkably, the very closely related questions about ambiguous path—(vi) and (ix)—are decidable, as we will show by providing an algorithm for solving them. This is very useful for linguistic modeling, since the undecidable question (viii) often gives the exact same answers as the decidable question (ix) in linguistics applications. In effect, we can, by substituting question (ix) for question (viii) provide an analysis of the multiple outputs in a grammar.

We will also show that question (x) is undecidable by algorithm, while (xi) is solvable, and we will give an algorithm for it.

Given that some of the above questions are undecidable, and hence unsolvable by any algorithm in the general case, we will conclude the chapter by investigating subclasses of finite-state transducers where more decision and closure properties are available, with particular focus, again, on natural language applications. We will also deduce and present a hierarchy of the most common subtypes of finite-state transducer models proposed in the literature.

5.2. Fundamental decidability questions of FSTs

5.2.1. Reductions to the Post Correspondence Problem

Most cases of undecidability regarding finite-state transducers are shown to be undecidable through a reduction to the well-known Post Correspondence Problem (Post, 1946), which is unsolvable by algorithm.

What the PCP asks us is: given a sequence of paired strings $u_1, l_1, \ldots, u_n, l_n$, is there a sequence of indices $i_1, \ldots, i_n$, such that the strings $u_{i_1}, \ldots, u_{i_n} = l_{i_1}, \ldots, l_{i_n}$?

This is often informally visualized as the task of pairing up blocks of strings, each block having a string on the upper side and on the lower side. An instance of the PCP could then be represented as:

$$[\frac{b}{ca}][\frac{a}{ab}][\frac{ca}{a}][\frac{abc}{c}] \tag{5.1}$$

One solution to this particular instance of the PCP is

$$[\frac{a}{ab}][\frac{b}{ca}][\frac{ca}{a}][\frac{a}{ab}][\frac{abc}{c}]$$

since the string on the upper side of the blocks we have chosen from the instance is identical to the string on the lower side: $abcaaabc$.

Question (i) above, the undecidability of the equivalence of two arbitrary finite-state transducers, follows almost as a corollary to the PCP.

Theorem 5.1. *Transducer equivalence is undecidable*

Given an instance of the PCP, we can create the transducers:

$$\tau_{PCPu} = ((i_1 \times u_1) \cup \ldots \cup (i_n \times u_n))^+ \tag{5.2}$$

and

$$\tau_{PCPl} = ((i_1 \times l_1) \cup \ldots \cup (i_n \times l_n))^+ \tag{5.3}$$

that is, two transducers that map indices to either the upper or lower words according to the instance of the PCP in question. For example, the two transducers τ_{PCPu} and τ_{PCPl} constructed from the instance (5.1) would both map 21324 to $abcaaabc$—it being a solution to the instance of the problem.

Now, we can also modify τ_{PCPu} and τ_{PCPl} in such a way that the output is guaranteed to be different than in the original descriptions of the two. This can be achieved by composing τ_{PCPu} and τ_{PCPl} with an auxiliary transducer τ_{change}, seen in figure 5.1. That is,

$$\tau_{PCPu} \circ \tau_{change} \tag{5.4}$$

would map a sequence of indices to all words in the PCP alphabet as the word indices correspond to, except that particular word. Staying with the example instance (5.1), the transducer constructed by (5.4) will map 21324 to any word over $\{a, b, c\}$ except $abcaaabc$. Note that in this construction (5.4) the output is guaranteed to be different only under the assumption that τ_{PCPu} and τ_{PCPl} represent functions, i.e. have at most one output for each input. It is easy to see that this is the case for both τ_{PCPu} and τ_{PCPl}, our encodings of the PCP. If affairs were not such and we could find words u and v such that $\tau(w) = \{u, v\}$, $\tau \circ \tau_{change}$ would not work as intended and would still perform the same mapping since u could be mapped to v and v to u by τ_{change}, and we could no longer be sure that any change is actually produced.

We can also construct a transducer τ' and have it operate in such a way that it maps the indices to any string over the PCP word alphabet. It is immediate that the following holds

$$(\tau_{PCPu} \circ \tau_{change}) \cup (\tau_{PCPl} \circ \tau_{change}) = \tau' \tag{5.5}$$

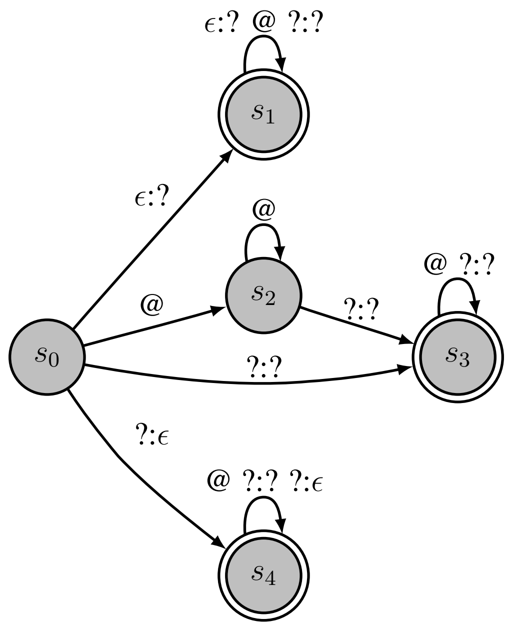


FIGURE 5.1. *Transducer τ_{change} that changes a word to anything except itself.*

if and only if the instance of the PCP lacks a solution. That is, if we could answer (5.5) we could solve the PCP—hence, transducer equality is unsolvable.[2]

We can also use the same reduction to show that question (iv)—for which inputs two transducers yield the same output—posed in the beginning of the chapter is undecidable.

Theorem 5.2. *Whether there exists a word w s.t. $\tau_1(w) = \tau_2(w)$ is undecidable*

Obviously, if τ_{PCPu} ever gives the same output as τ_{PCPl} for the same input, that input/output pair is a solution to the PCP.

In a similar way—question (iii)—for which inputs two transducers produce different outputs, can be addressed and shown to be undecidable. All the words on the left hand side that are different from words on the right in (5.5) would be solutions to the PCP.

Question (ii), the inclusion problem, if solvable, could also be used to the solve the PCP as laid out in (5.5) since inclusion both ways implies equality.

5.3. Functionality and ambiguity

Let us now turn to the question of whether a transducer τ represents a function, i.e. if it is single-valued for all inputs in its domain. Note that question (vi) about transducer path ambiguity is not the same. For example, a transducer that contains only the two paths

$$a{:}\epsilon \ \epsilon{:}b \ , \ a{:}b$$

both of which map an a to a b, is functional, but not unambiguous. Naturally, unambiguity implies functionality.

Schützenberger (1976) and Blattner and Head (1977) have shown that functionality is a decidable property of finite-state transducers. Subsequently, several algorithms have been proposed to effectively decide whether a given a τ, represents a functional relation (Roche

[2]This same result was first shown by Griffiths (1968) through a rather more involved reduction to the PCP.

and Schabes, 1997; Béal et al., 2000). Unfortunately, the existing algorithms are all very complex, and do not seem to have subexponential time requirements.

In what follows we shall first present a simple algorithm for deciding functionality.

5.3.1. Deciding functionality

We assume that the transducer in question is trim, i.e. the graph representing it is both accessible and coaccessible.

To decide the functionality of a transducer, we begin by observing that a transducer τ is functional if and only if the transducer

$$\tau' = \tau^{-1} \circ \tau \tag{5.6}$$

represents identity relations only, i.e. if for all words w in dom(τ'), $\tau'(w) = w$. That this holds is straightforward. If a relation τ is not a function, it must contain pairs (x, y) and (x, z) such that $y \neq z$. Then τ' contains (y, z) and (z, y), i.e. is not an identity relation. Conversely, suppose a) that τ is a function and that b) τ' contains a nonidentity relation. Then by b) we have that there exists a mapping $z \xrightarrow{\tau^{-1}} x \xrightarrow{\tau} y$ with $z \neq y$. However, then τ contains both (x, y) and (x, z) which contradicts the assumption a) that τ is a function. Hence, τ is a function iff τ' only contains identity relations.

Since transducer inversion and composition are straightforward to implement algorithmically, we can now focus on the latter question: does τ' represent identity relations only?[3]

5.3.2. An algorithm for deciding the identity property

Deciding the identity property of a transducer is non-trivial since it may contain one-sided ϵ-moves as well as symbol pairs $x_1{:}x_2$ where $x_1 \neq x_2$. Note that the latter circumstance is

[3]Though simple, I must attribute this crucial observation in (5.6) which the algorithm is based on to Culik II (1978), who, however, does not provide an effective procedure for the subsequent question about deciding the identity property.

not a sufficient condition for non-identity in a transducer (the transducer in figure 5.2 is indeed an identity transducer, yet contains unequal symbol pairs). This requires the algorithm to ascertain that every path in the transducer indeed represents an identity relation.

The algorithm for deciding the identity property is given in algorithm 5.1. The essence of the approach is as follows: we perform a depth-first search (DFS) on the graph representing the transducer in question. For every state v encountered during the DFS, we store a string d representing the discrepancy between the input and output sides of the path so far. For example, in figure 5.2, we have marked a discrepancy of the symbol **c** on the lower side of state 1, since we followed an $\epsilon{:}c$ edge to get to that state, and the previous discrepancy in state 0 was empty. The discrepancy is illustrated in the upper or lower square. In the example, the maximum length of the discrepancy is 1; naturally it can grow larger with several consecutive one-sided ϵ-moves.

In the algorithm, the function MATCH combines the new edge label with the discrepancy string stored in the state (if the leftmost symbol in the discrepancy string is compatible with the upper/lower symbol on the edge). It returns a new discrepancy string, which is stored in the target state. For instance, moving from state 1 to state 2 in figure 5.2, we have a discrepancy of **c** on the lower side, which matches the following upper-side **c**-symbol on the edge. However, the lower side of the edge contains **d**, so our new discrepancy string becomes a lower-side **d**, which is then stored in state 2.

The algorithm fails immediately if any one of three conditions are met during the DFS:

i) When attempting to follow an edge, the edge label is incompatible with the discrepancy stored in the current state.

ii) We discover a final state, and have a non-empty current discrepancy.

iii) We discover an already visited state, and the current discrepancy is not equal to the discrepancy stored in the visited state.

We assume that the special alphabet placeholder symbols @ and **?** may be present, an

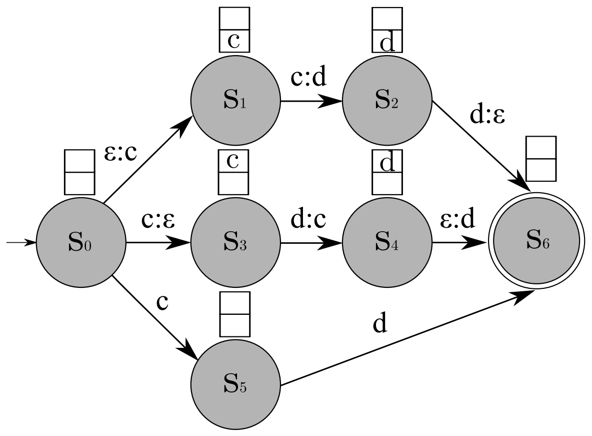


FIGURE 5.2. *Illustration of the* ISIDENTITY *algorithm on an identity transducer.*

@-edge representing any single-symbol identity of non-alphabet symbols, and **?** which can occur on both the input or output side, representing any symbol *not* in the alphabets, with **?**:**?** representing any non-identity translation of symbols not in the alphabets. To handle the presence of these two special symbols, we also fail if

iv) we encounter **?** on any side of any edge.

v) we encounter @ and have any non-empty discrepancy stored in the state the edge is in.

5.3.2.1. *Analysis*

To show the correctness of ISIDENTITY(τ) we note that if τ were to contain a non-identity path from s_0 to a final state s_f, it must be the case that there occurs a discrepancy in MATCH at some edge e. Conversely, if there exists a discrepancy at some edge e, and assuming the transducer is coaccessible, that discrepancy would propagate to some final state s_f, and there would exist a path for which τ is non-identity.

We can also show that the running time of the algorithm is bounded by a total $O(n^2)$, n being the number of states in τ. This follows from the fact that transducer inversion

Algorithm 5.1: ISIDENTITY(τ)

```
Input: $\tau$
Output: $\{TRUE, FALSE\}$
begin
    $\tau.id \leftarrow$ TRUE
    $s_0.d \leftarrow \emptyset$
    DFS_id($s_0$)
end

Procedure DFS_id(s)
begin
    $s.visited \leftarrow$ TRUE
    foreach edge $e$ ($s \rightarrow s'$) do
        if newd = MATCH(s.d,e) fails then
            $\tau.id \leftarrow$ FALSE; RETURN
        end
        if s' is final and newd $\neq \emptyset$ then
            $\tau.id \leftarrow$ FALSE; RETURN
        end
        if s' is visited and newd $\neq s'.d$ then
            $\tau.id \leftarrow$ FALSE; RETURN
        end
        if s' not visited then
            $s'.d$ = $newd$
            DFS_id($s'$)
        end
    end
end
```

is an $O(n)$ operation (we simply reverse the input and output labels), and composition is $O(|\tau_1||\tau_2|)$.[4] The DFS subsequently performed by ISIDENTITY takes time proportional to the number of edges in the new graph.

Now, for the string comparison between old and new discrepancies during the DFS, we note that we do not actually need to store redundant strings for each state—as the illustration in figure 5.2 perhaps suggests. Rather, the set of discrepancy strings form a tree (corresponding to two depth-first trees, one representing discrepancies on the lower side, and the other discrepancies on the upper side). The individual states can share parts of a tree of strings by maintaining two pointers marking the beginning and ending of its discrepancy string in the tree. To compare the equality of two discrepancies, we simply compare the two pointers. Since each edge adds at most one symbol to the discrepancy, we see that treating each edge takes $O(1)$-time. Thus, the total running time is $O(n^2)$, dominated by the composition operation.

5.4. Equivalence of functional transducers

Having settled the decidability of functionality, it is worth noting that as a consequence of the ISIDENTITY algorithm, we can also decide whether two transducers are equivalent, given that one of them is functional.

Theorem 5.3. *if τ_1 or τ_2 represent a function, whether $\tau_1 = \tau_2$ is decidable*

Of course, if one of the two is functional and the other not the question is trivial. If the two transducers τ_1 and τ_2 are both functional, we can test their identity by observing first that $\tau_1 = \tau_2$ iff dom(τ_1) = dom (τ_2) and

[4]Note that transducer composition may produce a nondeterministic transducer—however, we need not determinize the transducer in order to apply ISIDENTITY. If we did that, the algorithm would no longer be guaranteed to finish in polynomial time. To avoid determinization, the algorithm needs to be aware of ϵ:ϵ-transitions and include their treatments in the propagation of discrepancies. This means that for an ϵ:ϵ-transition, MATCH returns $newd = s.d$ for $e = \epsilon$ on line 11.

$$\textsc{IsIdentity}(\tau_2^{-1} \circ \tau_1) \tag{5.7}$$

Now, the testing of the equivalence of the domains of the two transducers simply involves testing the equivalence of their finite-state automata representations after extracting the input side only. That is, we can answer the question about transducer equivalence in the case of functional transducers, but not in the general case.

It should perhaps be noted that this is one of the few qualities that sets functional transducers apart from non-functional ones—the availability of the equivalence test. Most of the other properties under discussion, such as the emptiness of intersection, remain undecidable even for functional transducers as can be easily seen by the reduction to the PCP by τ_{PCPu} and τ_{PCPl} in (5.2) and (5.3), both of which are functional. Naturally, the intersection of two functional transducers cannot necessarily be represented as a regular relation either. For this last case, we need not even reduce the problem to the PCP, but may simply observe that for the two functional transducers

$$\tau_1 = (a{:}b)^*(b{:}\epsilon)^*, \tau_2 = (a{:}\epsilon)^*(b{:}b)^* \tag{5.8}$$

$$dom(\tau_1 \cap \tau_2) = a^n b^n \tag{5.9}$$

which is nonregular.[5]

[5]Curiously, one finds in the literature many examples where it is assumed that certain problems concerning functional transducers are decidable when they in fact are not. For example: "most of the problems such as equivalence or intersection [of FSTs] are easily shown to be equivalent to the Post Correspondence Problem and thus undecidable. The situation is drastically different for transducers that are functional, that is, transducers that realize functions, and the above problems become then easily decidable." (Béal et al., 2000). However, dealing with functional transducers as opposed to non-functional ones provides no further closure properties or decidable questions.

5.5. Extracting the non-functional domain

Now, turning to the follow-up question: supposing we knew that a transducer τ was non-functional. Perhaps τ was constructed with the intent that it be functional, after which the test revealed that it was not, and we would now like to know (perhaps for debugging purposes), which input words w cause $\tau(w)$ to produce multiple outputs. Unfortunately, the following is easily seen to hold:

Theorem 5.4. *Whether the set of words $\{w \mid \tau(w)$ is not single-valued$\} \subset \Sigma^*$ is undecidable*

Consider the transducer $\tau_{PCPu} \cup \tau_{PCPl}$ as constructed in (5.2) and (5.3). Obviously, if we could decide whether the set of words for which the output is not single-valued is a proper subset of Σ^*, we could solve the PCP.

5.6. Unambiguous vs. functional

A question closely related to the previous one about single-valuedness is the question about ambiguity. Recall that a transducer is ambiguous if there exists an input word x such that there is more than one path in the transducer leading to an accepting state with x as the input. Obviously a non-functional transducer is also ambiguous, but the converse is not always true. Consider, for instance, a transducer with the two paths

$$a{:}\epsilon \;\, \epsilon{:}b, a{:}b$$

Obviously τ is single-valued (as it only maps an a to a b), but ambiguous, as the transducer contains two paths with the same input.

There is a subtle difference between ambiguity and non-functionality which turns out to be very important. Namely, the set of words that produce an ambiguous path through a transducer τ is a regular set, and there exists an effective algorithm for extracting this set as we shall show below.

The ability to extract this set is important because most construction methods that produce complex transducers representing grammars in a natural-language processing setting will not produce ambiguous paths unless that transducer truly is non-functional.[6] What this means is that in practical cases we can forgo the (futile) attempt to extract the words w that produce multiple outputs in $\tau(w)$ and instead extract the words that produce multiple paths through $\tau(w)$, which is very likely the same set.

5.6.1. Testing transducer ambiguity

Before we proceed to the algorithm that extracts from a transducer τ the regular language U where each word in U has two accepting paths as input to τ, let us consider how to test for the presence of ambiguous paths in general.

The natural question is whether one can test for ambiguity in a transducer in a similar fashion as one tests for single-valuedness. As the algorithm for testing single-valuedness was developed from the observation that the composition of the inverse of a transducer with itself will exhibit non-identity relations if and only if the transducer is non-functional, we will develop a similar approach here.

The key to adapting the earlier algorithm to test ambiguity is to note that we are now interested in the actual path that an input word w induces in a transducer τ, and in whether an input word w yields two different paths. In other words, the range of the transducer, or the image of w, is irrelevant. Now, call the path induced by a word w in $\tau(w)$, $p(w)$. Naturally, $p(w)$ is a regular set and we can indeed represent each member of $p(w)$ as a string (for instance, as a sequence of state numbers). Hence, to test for transducer ambiguity, we want to test whether:

$$\textsc{IsIdentity}(\tau_p^{-1} \circ \tau_p) \tag{5.10}$$

[6]This assumes we have made sure that neither the cross-product nor the composition algorithms introduce multiple paths as discussed in chapter 3.

where τ_p is a transducer that maps a word in w to some string uniquely describing the path $p(w)$ induced by w in τ.

How do we modify the original transducer τ in such a way that it maps input words to some unique string for every path? Let us begin by observing that if τ were deterministic on its output side, every word emitted would also be a kind of description of the path taken in the transducer. Since we can disregard the image of w under τ, we can also directly disregard the original output labels of τ and modify them in such a way that τ becomes deterministic on the output side and produce a new transducer that maps input words to a string representation of the paths they induce.

One way to do this is to replace all the output labels on the transitions in τ with a symbol that corresponds to the target state of the transducer. This would directly yield a new transducer τ_p that would have the same domain as τ and would map input words w to a sequence of symbols describing the path that the input word induces in the transducer. In other words, τ_p maps words to descriptions of the path of states they induce in τ.

However, doing such a transformation is unnecessary and could result in some inefficiency as the number of new symbols that would need to be introduced into the alphabet would equal the number of states in τ. To obtain a unique description we need only to make sure that for each state there are no two outgoing transitions with the same output labels. The labels for the outputs can be chosen arbitrarily, so long as the requirement for determinism holds for each state: if the lower side is deterministic, every path p through τ will produce a unique word in its output labels. Hence, it may be the case, depending on the transducer at hand, that no new symbols need to be introduced at all, if the maximum number of transitions in some state does not exceed the size of the symbol alphabet.

Obviously, if we let m be the maximum number of transitions outgoing from a single state, we need to introduce $m - |\Sigma|$ new symbols, in case $m > |\Sigma|$. Asymptotically this may not improve the worst case much over having the lower side labels represent the target states, since if some state s has a transition to every state in the transducer, we will need $|\tau| - |\Sigma|$ new symbols. In practice, however, we may save much time by following the

Algorithm 5.2: TRANSDUCERTOPATH(τ)

```
   Input: $\tau$
   Output: $\tau_p$
 1 begin
 2     foreach state $s$ do
 3         $i \leftarrow 1$
 4         foreach transition label $p$:$q$ do
 5             if $i \geq |\Sigma|$ then
 6                 add a new symbol to $\Sigma$
 7             end
 8             replace $q$ with $\Sigma_i$
 9             $i \leftarrow i + 1$
10         end
11     end
12 end
```

latter strategy to only introduce new symbols where they are absolutely needed to maintain determinism in the lower side of the transducer.

5.6.2. An algorithm for extracting ambiguous words

In the above we showed how to decide transducer ambiguity by algorithm 5.2 and the observation in (5.10). If, instead of simply deciding if a transducer represents identity relations only, we could actually extract the set of words in τ where the output differs from the input, we could also extract the set of input words for which there are multiple paths through a transducer τ. Call the set of input words w where $\tau(w) \neq w$ NOTID(τ), and we have that

$$\mathcal{A}_{ambwords} = \text{dom}(\tau_p \circ \text{NOTID}(\tau_p^{-1} \circ \tau_p)) \tag{5.11}$$

is precisely the set of words

$$\{w \mid \tau(w) \text{ is ambiguous }\} \tag{5.12}$$

The notation τ_p is exactly as above, i.e. the result of TRANSDUCERTOPATH(τ).

5.6.2.1. Extracting the set of words not in an identity mapping

The algorithm 5.1 can be modified to yield the set of input words in a transducer which are not in an identity relationship only. This variant is given in algorithm 5.3 on page 155. The key to the modification is that, instead of terminating the algorithm when a discrepancy is noticed during the DFS, we simply change the output label in the offending edge to a new symbol **@notid@** that is not in the original alphabet Σ, and in such a way produce a transducer τ'. Now, any path in τ' containing this auxiliary symbol as output can participate in a mapping where the output word is different from the input word. Hence, as a subsequent step, we can perform the composition:

$$\tau' \circ (\Sigma^* \textbf{ @notid@ } \Sigma^*) \tag{5.13}$$

to yield a transducer whose input side or domain consists of all words that can produce a nonidentity output. The domain in this transducer then represents the regular set of words for which a nonidentity mapping is possible (line 5).

It is worth noting that what we are doing is extracting the set of input words

$$\{w \mid \text{there exists a word } \tau(w) \text{ s.t. } \tau(w) \neq w\} \tag{5.14}$$

which *may* also include paths that describe an identity relation. The similar task of extracting the set of words (or finding a single word) that *may*, through some path, produce an identity mapping, is easily seen to be undecidable.[7]

Combining algorithms 5.3 and 5.2 with observation (5.11) yields the following:

Theorem 5.5. *The set of words* $\{w \mid \tau(w)$ *is ambiguous* $\}$ *is a regular set*

[7]If such an algorithm were available, we could run it on the transducer $\tau_{PCPu} \circ \tau_{PCPl}{}^{-}1$ as defined in (5.2) and (5.3) and solve the PCP.

Algorithm 5.3: NOTID(τ)

```
   Input: $\tau$
   Output: $\mathcal{A}_{notid}$
1  begin
2      add @notid@ to $\Sigma$ in $\tau$
3      $s_0.d \leftarrow \emptyset$
4      DFS_notid($s_0$)
5      RETURN(dom($\tau \circ (\Sigma^*$ @notid@ $\Sigma^*)$))
6  end
8  Procedure DFS_notid(s)
9  begin
10     $s.visited \leftarrow$ TRUE
11     foreach edge $e$ ($s \rightarrow s'$) do
12         if newd = MATCH(s.d,e) fails then
13             change output side of edge label to @notid@
14         end
15         if s' is final and newd $\neq \emptyset$ then
16             change output side of edge label to @notid@
17         end
18         if s' is visited and newd $\neq s'.d$ then
19             change output side of edge label to @notid@
20         end
21         if s' not visited then
22             $s'.d = newd$
23             DFS_notid($s'$)
24         end
25     end
26 end
```

5.6.3. Splicing a transducer based on ambiguity

Having at our disposal the mechanism to extract the regular language $\mathcal{A}_{ambwords}$ from a transducer τ, we can subsequently also divide a transducer into two disjoint transducers such that one represents the unambiguous paths and the other the ambiguous paths. Again, we take advantage of the composition operation and can easily produce:

$$\tau_{ambiguous} = \mathcal{A}_{ambwords} \circ \tau \tag{5.15}$$

$$\tau_{unambiguous} = \neg\mathcal{A}_{ambwords} \circ \tau \tag{5.16}$$

Here, naturally

$$(\tau_{ambiguous} \cup \tau_{unambiguous}) = \tau \tag{5.17}$$

Figure 5.3 illustrates an ambiguous transducer (a) representing two parallel replacement rules

$$a \rightarrow (b \cup c)\ ,\ b \rightarrow a$$

and the resulting decomposition into its ambiguous part (b) and an unambiguous part (c).

5.7. A hierarchy of transducers

Having now answered a few fundamental decidability questions about general finite-state transducers, we can conclude that, while they are powerful models for expressing relations between strings, there are a number of shortcomings from the point of view of being used as tools for linguistic modeling. The foremost of these shortcomings is most likely the lack of effective algorithms for solving equivalence and subset problems between two arbitrary finite-state transducers. Along the same lines, the lack of the ability to enumerate as a set

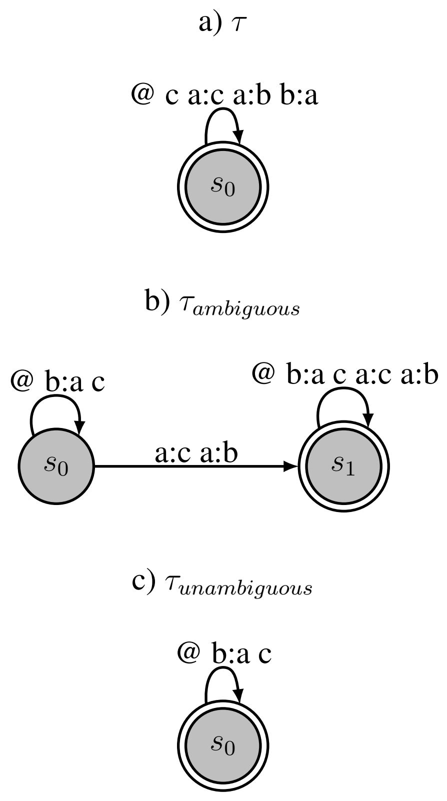


FIGURE 5.3. *Ambiguous transducer spliced into two disjoint ambiguous and unambiguous ones.*

(regular or otherwise) the words where the output of one transducer differs from another is a weakness. The lack of boolean operations—with the exception of the union of a regular relation—is less of a problem, and is made up for in part by the flexibility of boolean operations on either the domain or range of a relation together with the availability of composition and domain and range extractions.

It is therefore of some interest to consider the properties of other, more restricted models that perform regular translations—i.e. devices that map one regular language to another, and delineate the different strengths and weaknesses of each, especially with regard to the deficiencies just outlined that are associated with the general finite-state transducer model. Of the vast literature regarding regular translations, six models stand out as either direct candidates for performing such natural tasks as have been discussed, or have actually been employed in practice for such tasks.

These are:

1. Unrestricted finite-state transducers
2. Rabin-Scott two-tape automata/transducers
3. subsequential transducers
4. sequential transducers
5. even-length transducers
6. k-length-difference bounded transducers

So far we have primarily been dealing with the unrestricted transducer; a transducer where we pose no constraints on the structure of the underlying automaton or the occurrence of ϵ-moves. We will now consider the various restricted types more closely.

5.7.1. Sequential transducers and deterministic transducers

The sequential transducer is a transducer with an additional specific restriction on the output function—namely, that for each state there exist maximally one transition for each input symbol in Σ. That is, for all $s \in Q$:

(1) $\delta(s, a)$ contains at most one element for each $a \in \Sigma$.

While the currently established terminology is 'sequential' for this type of transducer (e.g. Mohri (1997b,a)), these are also sometimes called 'deterministic' transducers in the literature (see e.g. Aho and Ullman (1972)). Without transgressing the limits of input determinism, 'deterministic' transducers also allow ϵ-inputs in the special case where no other inputs are available in a state, i.e. for all $s \in Q$

(2) $\delta(s, \epsilon)$ contains maximally one element, and in such a case $\delta(s, a)$ is empty for all $a \in \Sigma$.

Including condition (2) allows us the transducer to yield multiple outputs for one input. We shall consider those transducers that fulfill condition (1) only to be sequential, and those that allow ϵ-inputs according to (2) deterministic.

5.7.2. Subsequential transducers

A variant of the sequential transducer are subsequential (Schützenberger, 1977) and p-subsequential transducers (Mohri, 1997b). These, in contrast to sequential transducers which allow no ambiguity, allow for restricted ambiguities in the output string, provided they occur at the end of the translation. This is achieved by declaring optionally the emission of p additional output strings at final states of the transducer.

The overarching purpose of the previous three classes of transducers is to guarantee that translations can be computed in linear time in proportion to the length of the input string, possibly with the addition of some constant of ambiguity in the case of p-subsequential transducers. Given an arbitrary finite-state transducer, it is decidable whether the same relation can be characterized by a sequential transducer (Choffrut, 1978).

5.7.3. Rabin-Scott transducers

What we shall here denote Rabin-Scott transducers were originally introduced in the landmark paper by Rabin and Scott (1959) which investigated many properties of different kinds of deterministic and nondeterministic finite-state automata. The term used in this publication for the concept was 'two-tape one-way automata.' This is a kind of deterministic finite automaton, with the additional feature that the set of states Q is divided into two disjoint classes Q_{in} and Q_{out} such that transitions from states belonging to Q_{in} operate on the input, and the ones in Q_{out} on the output. One can, depending on the application, interpret this as either an automaton that reads two input tapes where the state classes control which tape is to be read, or a transducer that reads a input string and outputs another one where the type of state controls which operation is currently to be performed. Additionally, the Rabin-Scott model is by fiat always aware of the end of the string and must input and output a special marker $\#$ not in Σ before halting.

The ability to foresee the end of tape allows the Rabin-Scott model to perform a strictly larger set of mappings than it would without the capability. For instance, the regular relation:

$$(a{:}b) \cup (a{:}\epsilon\ b{:}\epsilon) \tag{5.18}$$

is not expressible with the same model without the $\#$ symbol being available.

5.7.4. Even-length transducers

The concept of an even-length transducer is modeled by a deterministic finite automaton where each transition is marked by a pair of symbols from Σ. That is, ϵ labeled transitions are completely disallowed, and as such, input and output strings are always of the same length. Alternatively, we may consider also transducers they may contain ϵ-symbols, but where each translation is even-length, i.e. the input is always the same length as the output. Kaplan and Kay (1994, p. 344) contains a simple algorithm for normalizing ϵ-containing equal-length transducers into ϵ-free ones.

5.7.5. k-length-difference-bounded transducers

The k-length difference-bounded transducer is a simple generalization of the even-length transducer (Roark and Sproat, 2007). This is the case where for all w in dom(τ),

$$\big||w| - |\tau(w)|\big| \leq k$$

for some k. In other words the length of the input and output may be imbalanced, but there is a strict bound k on this imbalance.

5.7.6. Properties of restricted transducer models

Let us now consider the relationship between the classes of transducers outlined so far, both in terms of the types of relations that can be expressed by them, and in terms of some of their closure properties.

First, we observe that the class of even-length transducers τ_{el} enjoy all of the closure and decision properties that apply to finite automata in general since they can be modeled directly as special kinds of finite automata where the alphabet consists of symbol pairs only.

As regards the internal relationship between these classes, we can easily see that any even-length transducer can be encoded as a Rabin-Scott transducer with strict alternation of Q_{in} and Q_{out}, but that the reverse is not necessarily true for the simple reason that a Rabin-Scott transducer may output strings of different length that the input is. That is, even-length transducers are strictly included in the class of Rabin-Scott transducers. However, there exist even-length transducers not encodable as any subsequential transducer as illustrated by the FST in figure 5.4: this is a kind of transducer where the output depends upon whether

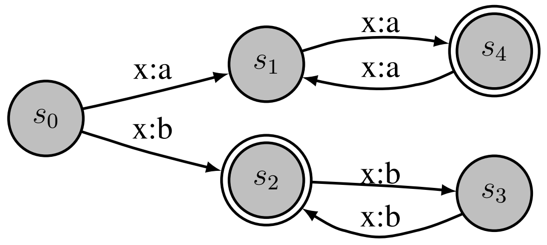


FIGURE 5.4. *Unsequentiable even-length transducer (Mohri, 1997b).*

the input is of even or odd length—something that cannot be ascertained without first consuming the entire input. Likewise, since sequential and subsequential transducers allow for the output of strings of different length than the input, there is a partial overlap between even-length and (sub)sequential transducers. Also, the class of sequential transducers is strictly included in the class of ($p-$)subsequential ones (by definition).

The class of subsequential transducers is also properly contained in the class of translations definable by Rabin-Scott transducers. This is easily seen as follows. Firstly, we can convert any p-subsequential transducer to a Rabin-Scott transducer by simply designating the start state as belonging to Q_{in} and from there interleaving Q_{in} and Q_{out} states according to the input/output labels of the p-subsequential transducer. As regards the additional emission of strings from final states in the p-subsequential transducer, we can model this by designating final states as belonging to Q_{in} from which we may have a $\#$-transition to a sequence of states all belonging to Q_{out}, ending in a $\#$-transition. Like this, we can optionally emit any regular language whenever the end of the input is encountered. Due to the lack of being able to express infinite ambiguity in the output, we cannot, however, convert an arbitrary Rabin-Scott transducer to a p-subsequential one. Figure 5.5 shows such a transducer. Also, the unsequentiable transducer in figure 5.4 which is easily representable as a Rabin-Scott transducer (and a k-length-difference bounded or even-length transducer) is not representable as any (sub)sequential one. We may also observe that there exists

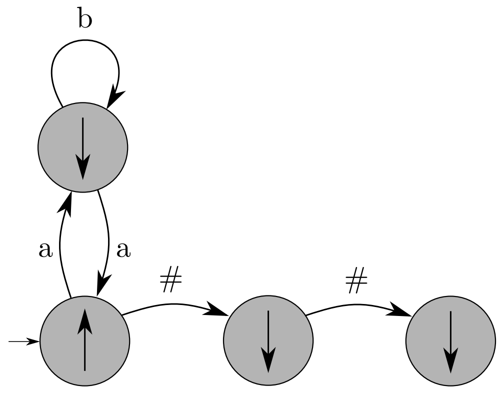


FIGURE 5.5. *A Rabin-Scott transducer which is not representable as a (sub)sequential transducer or k-length-difference-bounded transducer.*

a (sub)sequential transducer $(a{:}\epsilon)^*$ which is neither representable as k-length-difference-bounded one, nor as an even-length one.

These observations and the internal relationships between the classes are summarized in figure 5.6.

5.7.6.1. Boolean operations

The even-length transducers τ_{el} and τ_{k-lb} are closed under union and intersection—the former by virtue of it being representable as a finite-automaton consisting of label pairs. The proof for the latter is a straightforward extension of the conversion of a nonsynchronized even-length transducer to a synchronized one.[8] Neither class, however, is closed under complementation: to realize the complement of a relation within the two classes, both would have to include non-length-bounded translations.

That the union of two sequential or p-subsequential transducers is not necessarily sequential or p-subsequential is easily seen by a counterexample based on the transducer in figure 5.4: we may construct two transducers τ_1 and τ_2, such that τ_1 includes only the states and transitions pertaining to states $0, 1, 4$ and τ_2 only the ones in $0, 2, 3$. Clearly, both are

[8]The proof is omitted as e.g. Roark and Sproat (2007) contains a detailed exposition of it.

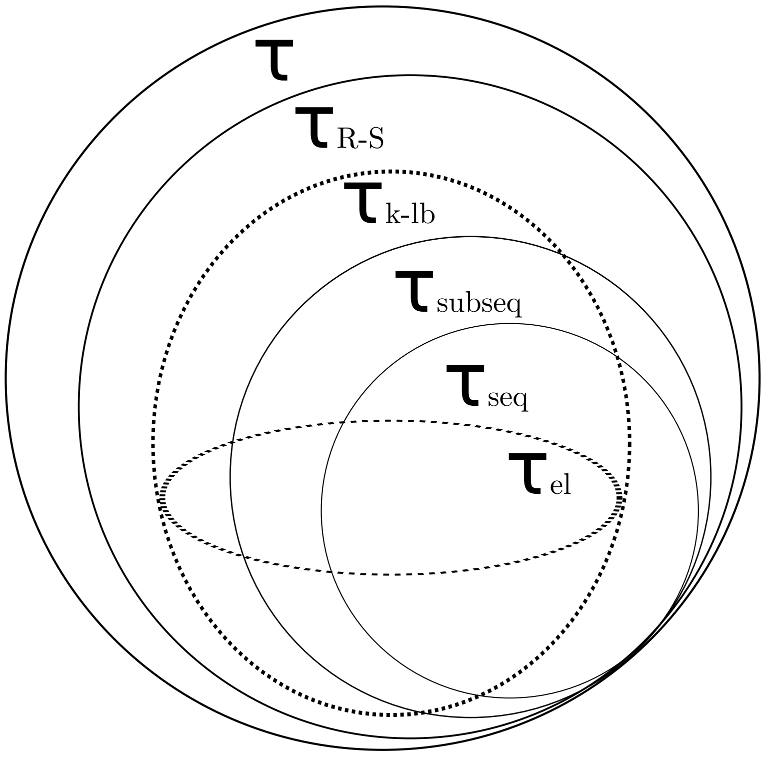


FIGURE 5.6. *A hierarchy of finite-state transducers.*

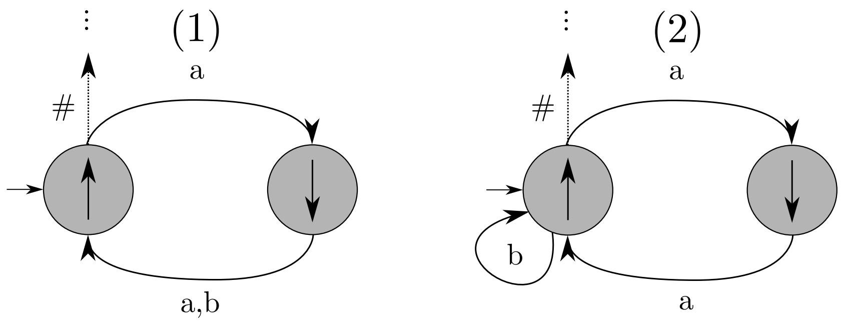


FIGURE 5.7. *Two Rabin-Scott transducers the composite of which cannot be represented by an R-S transducer.*

sequential (and hence p-subsequential). However, $\tau_1 \cup \tau_2$ is exactly the unsequentiable transducer in figure 5.4.

The (sub)sequential and length-difference bounded transducers are closed under composition (Choffrut, 1978; Mohri, 1997b), but the Rabin-Scott transducer is not. The latter circumstance can be illustrated by the two transducers in figure 5.7. Their composition yields the relation:

$$(a{:}a \cup a{:}b)^* \circ (a{:}a \cup b{:}\epsilon)^* = (a{:}a \cup a{:}\epsilon)^* \tag{5.19}$$

which cannot be expressed as a Rabin-Scott transducer.

5.7.6.2. Equivalence and inclusion

The equivalence problem for the Rabin-Scott transducer is decidable but the inclusion problem is not.[9] For the inclusion problem, undecidability follows because of closure under complement: we can construct two transducers τ_{PCPu} and $\neg\tau_{PCPl}$ as in (5.2) and (5.3), and now, an instance of the PCP has a solution iff $\tau_{PCPu} \nsubseteq \neg\tau_{PCPl}$.

[9]The former result was first shown in Bird (1973) and subsequently for the general case of deterministic multitape automata in Harju and Karhumäki (1991).

	τ_{el}	τ_{seq}	τ_{subseq}	τ_{k-lb}	τ_{R-S}	τ
$\tau_1 = \tau_2$?	+	+	+	+	+	-
$\tau_1 \subseteq \tau_2$?	+	+	+	+	-	-
$\tau_1 \cup \tau_2$	+	-	-	+	-	+
$\tau_1 \cap \tau_2$	+	-	-	+	-	-
$\tau_1 \circ \tau_2$	+	+	+	+	-	+
$\neg\tau_1$	-	-	-	-	+	-

TABLE 5.1. *Closure and decision properties for some classes of transducers.*

5.8. Discussion

In this chapter, an overview of a number of important fundamental decision properties of finite-state machines has been presented—with particular emphasis on such results that may be of significance when natural language grammars are encoded as finite-state devices.

Also, we have developed algorithms for deciding functionality and ambiguity of transducers; algorithms that may be useful both as a theoretical tool in linguistics research and as a practical grammar debugging tool.

Further, we have seen that equivalence is decidable for both unambiguous and functional transducers, and that we can splice a transducer into its ambiguous and unambiguous components. This offers a limited way of addressing the fundamental questions outlined in the beginning of the chapter. If we have two grammars G_1 and G_2 represented as transducers τ_1 and τ_2, we can at least decide their equivalence for those input words which produce an unambiguous path through the respective transducers. Naturally, from a linguistic and language engineering perspective we would like to answer the question of equivalence in its most general form, i.e. even for those parts of the grammars that are ambiguous. Further, if we know two grammars to be different, it would be helpful if we would also know for which input words their behavior differs. Unfortunately, the last two questions are undecidable by algorithm for transducers in general. This prompts the hope that one could deal with a restricted type of finite-state transducer for which the most relevant questions are solvable. Good candidates for such models are the class of p-subsequential transducers, or some variant of the Rabin-Scott-type transducer. For these two, the equivalence problem is decidable. Unfortunately, their closure properties are not ideal from the point of view of natural language processing applications, and it remains a challenge to produce a computational model of translation that fulfills all the desiderata of NLP: closure under the standard regular operators, composition, inversion, domain and range extraction, and the decidability of the most important tests such as equivalence and inclusion.

An additional complication is that none of the classes, with the exception of unrestricted transducers, are natural in the sense that they would arise through a combination of clearly specified abstract operations and primitives. The unrestricted class corresponds exactly to the type of transducers that can be obtained through a finite number of union, concatenation, Kleene closure, and cross-product operations. The unrestricted class is also closed under composition and inversion. Unfortunately, for the other classes no such characterization exists, and closure under composition and inversion is often not the case. Also, there is no algebraically elegant description of the more restricted classes of translations. Given an arbitrary transducer, it is often a matter of a posteriori showing that it in fact belongs to a more restricted class. But no restricted selection of operations seems to exist that would well capture the needs of natural language modeling and that would characterize a subclass of transducers with the desirable decidability properties outlined.

In many practical applications the lack of algorithms for equivalence tests will not be a problem. Due to the interaction of a number of constraints included in actual grammars, the resulting linguistic model will often be suitably restricted to one of the subclasses discussed above. When this is the case, equivalence tests and the like become available. This is the case whenever one is working with linguistic generalizations encoded as transducers (which are perhaps composed together) but that are also constrained against some finite lexicon. In short, for many purposes the reliance on a model that carries the burden of a plethora of undecidable properties may not be as severe a restriction.

On the other hand, if one wants to use the techniques and tools available in finite-state transducer technology for research in linguistic theory, the state of affairs may be different. In such a setting, one often finds that it would be desirable to ascertain whether some postulated hypothesis of a linguistic principle or rule is equivalent to, or substantially different from some other rule. If such hypotheses are encoded as finite-state transducers, it would be of great benefit to this type of research if the fundamental questions outlined in this chapter could be easily answered. However, it is precisely for this type of applications that the mappings one is likely to encode in a set of finite-state transducers are open-ended enough, not being constrained against lexica or fully complete grammars, to not fall into the subcategories where interesting properties can easily be solved by established algorithms.

Part III ALTERNATIVE FINITE-STATE MODELS

6. EXTENDING REGULAR EXPRESSIONS WITH PREDICATE LOGIC

6.1. Introduction

A three-operator regular-expression formalism $\{\cdot, \cup, ^*\}$ allows us (by Kleene's Theorem) to express any regular language and construct any finite automaton. In a similar way, adding a cross-product operator, $\times$, as the fourth operator to create a relation from two regular languages, allows us to express any regular relation, or any mapping that a finite-state transducer can perform. Despite the expressive power of such systems, there remain types of regular languages that are still very difficult to express through such a limited selection of operations. These types of languages usually express some form of existential constraint on the well-formedness of strings in some language L. For example, it is surprisingly difficult in a three-operator system to express the regular language that contains all strings over an alphabet, except those that contain some forbidden string w as a substring. It is easy to show that such a language must be regular, but much harder to actually construct such a language from arbitrary instances of w. This is especially true if w is a set, and not a single string. Such problems are alleviated substantially by the introduction of additional Boolean operators $\{\neg, \cap, -\}$. The language that does not contain a forbidden subword w, for example, is then easily described as $\neg(\Sigma^* w \Sigma^*)$.

However, even the power of Boolean operators is insufficient in many circumstances. There remains a class of regular languages that is still very cumbersome to capture through extended regular expressions. This class is characterized by language problems that concern *overlapping substrings*. This group of language construction problems comes up very frequently in natural language applications: in converting rewrite rules to finite state transducers, in compiling two-level rules, in expressing existential constraints over string sets, and the like.

We need not venture that far into natural language processing to find an example of a difficult-to-define regular language. The simple language L' where there exists one and only one instance of a substring drawn from a set L, if constructed through basic regular expressions and Boolean operators, needs to be expressed by something as unwieldy as:[1]

$$(\Sigma^* L \Sigma^*) - (\Sigma^*((\Sigma^+ L \Sigma^* \cap L\Sigma^*) \cup (L\Sigma^+ \cap L))\Sigma^*) \tag{6.1}$$

Now, even if the construction here appears to be correct, it is quite difficult to show that it actually works for every instance of the problem. As we move further into the realm of modeling natural language problems, small subproblems such as the one above become much more frequent. Even if there is a solution to every subproblem based on extended regular expression manipulation, this construction method becomes uncomfortable due to its complexity, the length of the regular expressions, and the sheer difficulty of showing correctness.

In this chapter we shall try to alleviate the problems relating to the description of such language problems. We will do so by developing a formalism of first-order logic over substrings with the purpose of simplifying the construction of complex languages. The fundamental idea behind the approach is that it allows us to use the basic building blocks of first-order logic—quantifiers, propositions—and combine these with power of regular expressions to define very complex languages in a simple way.

First, in section 6.2 we will discuss other solutions to the problem of constructing complex automata and transducers. In section 6.3, the notation and the semantics of the formalism will be briefly introduced, after which section 6.4 will present in detail the actual method for compiling logical sentences into automata. Section 6.5 will discuss a range of immediate applications drawn mainly from the needs of computational morphology and phonology where the construction method can be profitably employed. We conclude with

[1]It might appear that the much simpler $(\Sigma^* L \Sigma^*) - (\Sigma^* L \Sigma^* L \Sigma^*)$ does the job. However, the expression crucially cannot capture the cases where instances of L overlap with each other. Consider $L = (ab \cup ba)$. Then L' should not include the string aba since it contains both a substring ab and a substring ba. But aba is included in the language defined by the simpler expression just considered.

a discussion in section 6.6 on the relationships between the logical formalism and other construction methods found in the literature.

6.2. Previous work

This chapter will indirectly touch on two different topics of research. The first is the relationship between regular languages and logic over strings. The second is the question of defining the kinds of regular languages necessary to model components of natural language, primarily phonology and morphology.

The strong relationship between logic over strings and regular languages has been established in works such as Büchi (1960); Elgot (1961); McNaughton and Papert (1971). Each of these take a slightly different approach, but the overall methods are very similar. In particular, the closest resemblance to the current work is found in what is now called first-order logic of one successor over strings (FO[<]). This is a logic that combines first-order quantification and two simple predicates which allows one to assert properties of languages: either that a symbol in position x is a, or that position x is succeeded by position y. Through this formalism, one can characterize exactly the set of star-free languages, i.e. languages definable through extended regular expressions without Kleene closure (McNaughton and Papert, 1971).

An extension of this, called monadic second-order logic over strings (MSOL[S]), that augments the previous approach with a single monadic predicate, can be shown to equal the regular languages in its expressive power.[2]

The main differences between the work here and previous notations for first-order logic over strings is that in the current notation, quantifiers operate over substrings, and the model discards completely the connection between 'integer'-marked positions in a string as is done in FO[<]. Also, in the current work, predicates are designed to be augmented liberally: it is not the intention here to construct a maximally impoverished system to investigate

[2]Instead of the original sources on these topics, the reader is primarily directed to Thomas (1997) for a fairly thorough and accessible description of all of the above.

its formal properties—in fact, we strive to do the opposite, to construct a rich system that includes as many different propositions and notations as possible. To this end, we retain the ability to use regular expressions whenever they are a more convenient formalism, and at the same time use the predicate logic formalism for those parts that it is particularly suitable for. We shall assume the design philosophy that our system should to be as flexible as possible so that our logical formulas can express clearly and concisely a large amount of information in little space.

An earlier effort to combine logical formalisms and finite-state language processing is found in Vaillette (2003), who develops quite successfully the MSOL[S]-logic from the ground up. The main line of reasoning behind that work appears to be the verifiability of correctness of complex regular languages and relations—the idea that replacing ad hoc regular expressions with a logical formalism allows one to ascertain that a complex series of operations really does what it is supposed to do. In this respect, the motivation in this chapter is similar. Using the logical primitives, Vaillette shows that many of the same problems we tackle here (compiling string rewriting formulas in particular) can be characterized by MSOL[S]-logic. Our approach is different in three respects. Firstly, Vaillette relies on a separate compiler for converting MSOL[S]-formulas into automata; second, the logical notation is built by combinations of very primitive predicates, and third, there is no direct interface between the logical formalism and the well-established regular expression formalism. In contrast, in this chapter we shall convert logical formulas systematically into regular expressions, and in such a way make it feasible to directly compile the logic into automata. Indeed, we will incorporate the logic as alternative notation when dealing with regular language descriptions. We shall also allow arbitrary predicates (as long as they are built from regular languages) and so provide predicates that operate directly on an abstract level, and third, we shall allow arbitrary intermixing of regular expression formalisms and the logical formalism.

The second thread by which this chapter ties in with previous research is the compilation problems of phonological and morphological transducers. Most of the problematic cases relate to compiling either string rewriting rules (of which there exist several flavors) or two-level rules.

The standard technique for constructing complex automata and transducers in finite-state natural language processing (such as those modeling rewrite rules) has generally been to augment the symbol alphabet Σ with supporting auxiliary symbols and then define, usually through a multi-step cascade of compositions of more primitive regular relation, a more complex regular relation that encodes the result, with the exception that some new symbols are present. These auxiliary symbols are then removed from the language or relation, after which the desired characterization has been encoded. In fact, this method of compiling complex transducers is so widespread that it is difficult to find any instance in the literature where the technique is not used. For instance,

- Karttunen et al. (1987) use "auxiliary brackets" to develop a rule compiler for two-level rules, where a significant portion of the description of the method is devoted to complications in compiling "overlapping" restriction rules.
- Kaplan and Kay (1994) make extensive use of "auxiliary brackets" which are inserted, whose presence is constrained, and which are then appropriately ignored in some contexts in defining rewrite rules and two-level rules as regular relations.
- Karttunen (1997), Kempe and Karttunen (1996), and Karttunen (1996) use various bracketing systems to define replacement, directed replacement, and parallel replacement rules.
- Yli-Jyrä and Koskenniemi (2004) define a context restriction operator ($\Rightarrow$), as well as a more generalized context restriction operator, through the use of a $\diamond$-symbol, whose occurrence is constrained and which is then removed.

This method, though very expressive, contains possible drawbacks such as the difficulty of post-design analysis of complex constructions, as well as verification of their correctness (as aptly noted by Vaillette (2003)).

Nevertheless, the ‘auxiliary symbol’ construction method is very useful and there is no reason to abandon it other than that it is often employed ad hoc and in a completely unsystematized fashion. In this chapter, we shall define a logical notation through a decomposition of the auxiliary symbol method into its most primitive components. That is, we define the logical notation through very simple primitives which involve manipulation of auxiliary symbols. It may even be argued that the formalism presented here is really a generalization of the technique of auxiliary symbol usage which is so popular in solving complex language problems. In fact, as will be discussed later, one can, after the logical notation in this chapter has been presented, go back and reanalyze some of the compilation formulas given in the literature, which at first appear ad hoc, in terms of this first-order logic. In a way then, what follows can be interpreted as taking a variety of seemingly different approaches to compiling complex regular languages and relations and combining them under a more abstract heading.

6.3. Notation

Before going into the actual method of converting our logical notation into finite automata, let us look at a brief overview of what we are trying to accomplish with the notation, as well as the intended semantics of our formalism.

In what follows we shall assume the standard notational devices of first-order quantification $(\exists x)$, $(\forall x)$, the five connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, as well as a number of predicates which we shall separately define. At this point in the exposition we shall only be concerned with three types of predicates:

- $(x \in L)$
- $x = y$
- $S(t_1, \ldots, t_n)$

The predicate $(x \in L)$ is true whenever x takes as its value a substring which is a member of the regular language L, $x = y$ is true if the position of a substring x in a string coincides with the position of y. Here 'position' refers to the span of a substring that includes both a beginning position and ending position (which may be the same). Therefore $x = y$ is not necessarily true if the substring denoted x begins where the substring y begins, but the two must also end in the same position. The k-ary predicate $S(t_1, \ldots, t_n)$ is true for all strings where substring t_1 is immediately (with no intervening symbols) followed by t_2, and t_2 immediately by t_3, etc.

What we want to achieve is a notation where we can construct sentences using quantifiers and propositions in such a way that the quantification and bound propositions apply to substrings. The end result is that a sentence in our notation shall characterize some subset of Σ^* for which that sentence is true. That is, our domain of discourse is Σ^* and the range of a quantification is the set of substrings of strings in Σ^*.

Let us examine some example sentences that characterize regular languages, and look at the languages they indirectly define:

(i) $(\exists x)(x \in ab)$

(ii) $(\exists x)(x \in ab) \wedge (\exists y)(y \in cd)$

(iii) $(\forall x)(x \in ab)$

(iv) $(\forall x)((x \in ab) \rightarrow S(x, d))$

(v) $(\forall x)((x \in ab) \rightarrow (S(c, x) \vee S(x, d)))$

(vi) $(\forall x)((x \in ab) \rightarrow S(x, \Sigma^* d))$

(vii) $(\exists x)(x \in L \wedge (\forall y)(y \in L \rightarrow x = y))$

Sentence (i) is true for the language $(\Sigma^* ab \Sigma^*)$. That is, the sentence characterizes all the strings that contain ab as a substring—or more precisely, all the strings that contain a substring which we call x and where the substring x is a member of the language ab. Sentence (ii) is true for all strings that contain both a substring ab and a substring cd.

Sentence (iii) is not true for any string in Σ^*. That is, there is no language that characterizes sentence (iii). Hence, it is true only for the empty language $\emptyset$. It is easy to see that there can not exist a string in Σ^* such that every substring in it would be ab.

Sentence (iv) is true for all strings where, whenever ab is found as a substring, it is followed by the string d. Sentence (v) is true for all strings where, whenever ab is found as a substring, it is preceded by c or followed by d. Sentence (vi) is the language where, whenever ab occurs as a substring, it is followed some time later (not necessarily contiguously) by d.

Sentence (vii) is the language already characterized by the regular expression in (6.1). It is true for all strings that contain one and exactly one substring drawn from the language L. Note the use of the equality of position predicate: there exists a substring x such that x is a member of L, and for all y such that y is also in L it must be the case that the position of x and y are the same.

6.4. Compiling logical formulas

6.4.1. Notational preliminaries

As mentioned above, in compiling logical formulas to automata, we will take advantage of auxiliary symbol manipulation in the construction process. To this end, we shall refer to the alphabets Σ (which is the actual alphabet that we're interested in), an auxiliary alphabet Γ, and a joint alphabet $\Delta = \Sigma \cup \Gamma$.

6.4.2. Introductory notions

Consider the effect of defining a language over an alphabet $\Delta = \Sigma \cup \Gamma$, where the alphabet is divided into two parts $\Sigma = \{a, b\}$ and $\Gamma = \{\bigcirc\}$ (our auxiliary alphabet), such that it contains exactly one instance of $\bigcirc$, i.e. $(\Sigma^* \bigcirc \Sigma^*)$, and then intersecting this language with a language that contains the $\bigcirc$-symbol followed by a, and finally deleting the $\bigcirc$ symbol from the language. Let us call this auxiliary symbol removal operation $\Pi(\mathcal{L})$, i.e. a homomorphism $\Gamma \rightarrow \epsilon$. In other words:

$$\Pi((\Sigma^* \bigcirc \Sigma^*) \cap (\Delta^* \bigcirc a\Delta^*)) \tag{6.2}$$

The end result in this example is the language over Σ^* that contains at least one a. From one perspective it is simply another way of saying $(\Sigma^* a \Sigma^*)$. However, laid out in this fashion, we can see that separating the regular expression into two parts has brought about two independent statements with different semantics: the first one, $(\Sigma^* \bigcirc \Sigma^*)$, asserts the existence of exactly one symbol $\bigcirc$, while the second, $(\Delta^* \bigcirc a\Delta^*)$ asserts that there is a $\bigcirc$-symbol which is followed by an a. Informally, the first part says "there exists exactly one position called $\bigcirc$," and the second part: "some position called $\bigcirc$ is followed by an a."

In effect what we have achieved in intersecting these two statements and deleting the $\bigcirc$-symbol is a form of variable binding—the first regular expression being equivalent to existential quantification of a position in a string, or the "existence of a substring," and the latter a proposition bound by the variable $\bigcirc$. This specific example illustrates the fundamental connection between first-order logic and regular languages.

We can now expand the same idea, and replace the first part of the regular expression with $(\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*)$ (for the sake of clarity in notation, let us replace the $\bigcirc$ with $\textcircled{x}$ in the auxiliary alphabet Δ, to make clear that our auxiliary symbol says something about a letter variable which we shall call x). We are now defining the language over Δ^* that contains exactly two symbols $\textcircled{x}$. The purpose of the two $\textcircled{x}$-symbols is to delineate two positions in a string x, the starting and the ending position. Let us call the combined effect of this regular expression and of removing the auxiliary symbols $\Pi(\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*)$ the *regular expression equivalent* of $(\exists x)$. In intersecting this language before removal of the auxiliary symbols with any regular language φ (that may or may not contain $\textcircled{x}$) we can achieve a propositional sentence

$$(\exists x)(\varphi) \tag{6.3}$$

6.4.3. Propositions

To continue with this idea: what about the possible propositions in φ? In modelling of the open statements φ, the simplest desirable proposition would be one with the semantics that a substring is a member of some language $\mathcal{L}$, i.e. $x \in \mathcal{L}$. Over Δ^* the regular expression $(\Delta^* \textcircled{x} \mathcal{L} \textcircled{x} \Delta^*)$ describes precisely this circumstance: "there exists a substring $\textcircled{x}\mathcal{L}\textcircled{x}$." The successor-of-relationship $S(t_1, \ldots t_n)$ alluded to earlier—where t_1 is immediately succeeded by t_2 etc., and where the terms could either be arbitrary languages or variables—translates naturally to $(\Delta^* t_1 t_2 \ldots t_n \Delta^*)$, for example, $S(x, \mathcal{A})$ would be rendered as $(\Delta^* \textcircled{x} \Delta^* \textcircled{x} \mathcal{A} \Delta^*)$.

Since we have seen that $(\exists x)$ in our still tentative logic over strings can be modelled by $\Pi(\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*)$, and since a universally quantified proposition $(\forall x)(\varphi)$ is equivalent to $\neg(\exists x)\neg(\varphi)$, the regular expression equivalent of a universally quantified statement is:

$$(\forall x)(\varphi) \equiv \neg\Pi((\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*) \cap \neg(\varphi)) \tag{6.4}$$

We are now in a position to put together a complete logical sentence. For example, the sentence:

$$(\forall x)(x \in \mathcal{A} \rightarrow S(x, \mathcal{B})) \tag{6.5}$$

would describe the language where every instance of a member of language $\mathcal{A}$ is immediately followed by a string from language $\mathcal{B}$. In translating the open statements to regular expressions, we make use of the conditional laws of statement logic, where $(P \rightarrow Q) \Leftrightarrow (\neg P \vee Q)$, and we find the equivalent regular expression following the above scheme:

$$\overbrace{\neg\Pi\Big((\Sigma^*\textcircled{x}\Sigma^*\textcircled{x}\Sigma^*)}^{(\forall x)} \quad\cap\quad \neg(\neg\,\overbrace{(\Delta^*\textcircled{x}\mathcal{A}\textcircled{x}\Delta^*)}^{x\in\mathcal{A}} \quad\cup\quad \overbrace{(\Delta^*\textcircled{x}\Delta^*\textcircled{x}\mathcal{B}\Delta^*)}^{S(x,\mathcal{B})})\Big)$$

6.4.4. Variables

So far we have only assumed propositions quantified by a single variable x. Naturally, we will want to extend this to an arbitrary number of variables. This requires some bookkeeping on the part of the alphabets. Suppose we have a sentence

$$(\forall x)(\exists y)(\varphi) \tag{6.6}$$

Now, it will not do to define $(\exists y)$ as $(\Sigma^*\textcircled{y}\Sigma^*\textcircled{y}\Sigma^*)$ for the simple reason that this precludes the existence of $\textcircled{x}$ symbols (as $\textcircled{x}$ is not a symbol of Σ). So, with several symbols, we need the ability to describe "any symbol in Δ except $\textcircled{y}$," to ensure that we allow other auxiliary symbols in the regular expression equivalent of $(\exists y)$. This is of course easy to describe as a regular language $(\Delta - \textcircled{y})$, and as a shorthand and to keep the notation clean we shall say Δ_y signifies precisely this: any symbol in the alphabet Δ except $\textcircled{y}$. Hence, a construction such as

$$(\forall x)(\exists y)(\varphi) = \neg(\exists x)\neg\big((\exists y)(\varphi)\big) \tag{6.7}$$

becomes

$$\neg\Pi\Big((\Delta_x^*\textcircled{x}\Delta_x^*\textcircled{x}\Delta_x^*) \cap \neg\Pi\big((\Delta_y^*\textcircled{y}\Delta_y^*\textcircled{y}\Delta_y^*) \cap (\varphi)\big)\Big) \tag{6.8}$$

Until now, we have said little about the operation $\Pi(\mathcal{L})$, except that it deletes the symbols in our "variable alphabet" Γ from the language $\mathcal{L}$. From a formal language perspective, this is simply a substitution $\Gamma \rightarrow \epsilon$, or, from an automaton perspective, a replacement of transitions containing symbols from Γ with ϵ-transitions. Again, in order to keep the notation uncluttered, we shall define $\Pi(\mathcal{L})$ as a *dynamic* operation, that also changes the alphabet Γ, shrinking it by one symbol, which is the symbol that is currently being removed.

This operation is crucial for the possible language complements that need to be taken in the process of eliminating several quantifiers. In the above example (6.8), for instance, the innermost Π-operation deletes the symbol $\textcircled{y}$ from the language and removes the symbol $\textcircled{y}$ from Γ, leading to that the following complement is taken with respect to an alphabet Δ (recall that $\Delta = \Gamma \cup \Sigma$) that only contains one auxiliary $\{\textcircled{x}\}$. Likewise, after the outermost Π operation, Δ = Σ, since all auxiliaries have now been purged from the auxiliary alphabet. This operation could be described non-dynamically, but at the cost of much lengthier expressions and without contributing to the clarity of the operation.

6.4.5. Propositions

We are naturally not restricted to the propositions developed so far—in fact a generous interpretation in this approach is that any subset of the language Δ^* is a proposition.

Since a proposition, such as $x \in \mathcal{L}$, i.e. $(\Delta^* \textcircled{x} \mathcal{L} \textcircled{x} \Delta^*)$ may contain sublanguages where no variable symbols occur—in this example $\mathcal{L}$ may be such a language—care must be taken to ensure that other variables can freely occur within the regular expression equivalent of the proposition. Hence, propositions should in general be augmented with freely interspersed symbols from Γ, our marker alphabet. The proposition $x \in \mathcal{L}$ then becomes $(\Delta^* \textcircled{x} \mathcal{L} \textcircled{x} \Delta^*) \parallel \Gamma^*$.[3]

For example, combining this with the successor-of predicate yields for $S(\mathcal{L}, x, \mathcal{R})$ the regular expression

$$(\Delta^* \mathcal{L} \textcircled{x} \Delta^* \textcircled{x} \mathcal{R} \Delta^*) \parallel \Gamma^* \tag{6.9}$$

[3]This would of course be equivalent to $(\Delta^* \textcircled{x} (\mathcal{L} \parallel \Gamma^*) \textcircled{x} \Delta^*)$, which may be more efficient to compile because of less non-determinism in the intermediate results: if $\mathcal{L}$ contains no symbols from Γ, which should be the case, then allowing symbols from Γ to freely occur within strings from $\mathcal{L}$ will not introduce non-determinism in the automaton construction. However, for the sake of generality, we will simply say that a proposition P shall be implemented as above, with symbols from Γ occurring anywhere, i.e. $P \parallel \Gamma^*$.

Logic		**R.E. equivalent**	**Notes**
$(\exists x)(\varphi)$	$\equiv$	$\Pi\big((\Delta_x^*\textcircled{x}\Delta_x^*\textcircled{x}\Delta_x^*)\cap(\varphi)\big)$	$\Delta_x = (\Delta - \textcircled{x})$
$(\forall x)(\varphi)$	$\equiv$	$\neg\Pi\big((\Delta_x^*\textcircled{x}\Delta_x^*\textcircled{x}\Delta_x^*)\cap\neg(\varphi)\big)$	
$x \in \mathcal{L}$	$\equiv$	$(\Delta^*\textcircled{x}\mathcal{L}\textcircled{x}\Delta^*) \parallel \Gamma^*$	
$S(t_1,\dots,t_n)$	$\equiv$	$(\Delta^* t_1 \dots t_n \Delta^*) \parallel \Gamma^*$	$t_i = \textcircled{x_i}\Delta^*\textcircled{x_i}$ for a variable x_i
$x = y$	$\equiv$	$(\Delta^*(\textcircled{x} \parallel \textcircled{y})\Delta^*(\textcircled{x} \parallel \textcircled{y})\Delta^*)$	

TABLE 6.1. Table summarizing the logical notation and their the regular expression equivalents.

6.4.6. Interim summary

We now have a construction method by which our proposed logical notation can be systematically converted to regular expressions, and hence to finite-state automata. In particular, new propositions can be introduced in a fairly straightforward way, and we shall do so whenever needed in the upcoming examples. The basic construction together with basic propositions is summarized in Table 6.1, where we assume the alphabets Γ and Σ, where Γ is the marker alphabet that contains the variable symbols under quantification, such as $\textcircled{x}$, $\textcircled{y}$, etc. Collectively, the two alphabets together are denoted Δ, i.e. $\Delta = \Gamma \cup \Sigma$. The operation $\Pi(\mathcal{L})$ deletes the currently quantified variable symbol from $\mathcal{L}$, and removes it from Γ.

We can now proceed to tackle a selection of difficult regular language problems and illustrate their solution through the notation developed here.

6.4.7. An example construction

Let us return to the example construction of a language that contains only one factor from some arbitrary regular language $\mathcal{L}$, for which a regular expression was given in (6.1). As we saw, this language, in our logical notation translates to:

$$(\exists x)(x \in \mathcal{L} \land (\forall y)(y \in \mathcal{L} \rightarrow x = y)) \tag{6.10}$$

Here we need a way to model the proposition $x = y$ for some variables x and y. This circumstance is captured by the language where both $\textcircled{x}$ and $\textcircled{y}$ markers share the same positions, i.e. occur in either order without intervening symbols from Σ (although other symbols from Γ may intervene between the two); see table 6.1.

Again, using the fact that $(P \rightarrow Q) \Longleftrightarrow (\neg P \vee Q)$, and following the translation method given, we get the following regular expression:

$$\Pi\big((\Delta_x^*\textcircled{x}\Delta_x^*\textcircled{x}\Delta_x^*) \cap (\alpha \cap \neg\Pi((\Delta_y^*\textcircled{y}\Delta_y^*\textcircled{y}\Delta_y^*) \cap \neg(\neg\beta \cup \gamma)))\big) \tag{6.11}$$

where:

$$\begin{array}{rcl}
\alpha & = & (\Delta^*\textcircled{x}\mathcal{L}\textcircled{x}\Delta^*) \parallel \textcircled{x}^* \\
\beta & = & (\Delta^*\textcircled{y}\mathcal{L}\textcircled{y}\Delta^*) \parallel (\textcircled{x} \cup \textcircled{y})^* \\
\gamma & = & (\Delta^*(\textcircled{x} \parallel \textcircled{y})\Delta^*(\textcircled{x} \parallel \textcircled{y})\Delta^*) \parallel \Gamma^*
\end{array}$$

It should be noted that there is much room for optimization in this particular construction: for instance, it is obvious that the shuffle product is unnecessary in α and partly so in γ, etc.; however, we represent them explicitly here to follow the construction method mechanically. In general, depending on the nature of the propositions and the formula, some steps can be optimized or omitted to avoid unwanted nondeterminism in the intermediate stages of automaton construction.

6.5. Applications

Let us now consider some larger example applications of the logical formalism. From now on, we shall omit the actual translation of a logical sentence into a regular expressions, with the exception that new propositions will be defined as needed. We assume that the construction is carried out mechanically following the method given above and the equivalences in table 6.1.

6.5.1. Context restriction

The context restriction operator (Koskenniemi, 1983; Yli-Jyrä, 2003) is a notational device to build regular languages from existential constraints over how and where certain strings can occur. The notation is as follows:

$$\mathcal{A} \Rightarrow \mathcal{B}_1 \,\_\, \mathcal{C}_1, \ldots, \mathcal{B}_n \,\_\, \mathcal{C}_n \tag{6.12}$$

The intended semantics is that a context restriction statement of the format above defines the language where every instance of a substring from the language $\mathcal{A}$ is surrounded by some pair $\mathcal{B}_i$ and $\mathcal{C}_i$. The languages $\mathcal{A}$, $\mathcal{B}_i$ and $\mathcal{C}_i$ are all assumed to be arbitrary regular languages. For example, the statement

$$x \Rightarrow a \,\_\, b \;,\; c \,\_\, d \tag{6.13}$$

characterizes the language where each instance of x must either be preceded by a and followed by b, or preceded by c and followed by d. This language includes strings such as $axbb$ and cxd, but not strings like x or axd.

This type of a constraint is quite challenging in the general case to capture through standard regular expression operators. Yli-Jyrä (2003) presents a conversion method from such a formula to a regular expression, where this regular expression grows exponentially as the length of the formula grows—in effect showing that more advanced methods are necessary to compile such statements into automata with any efficiency. Subsequently, Yli-Jyrä and Koskenniemi (2004) have provided an efficient formula that—not unsurprisingly—uses auxiliary symbol manipulation and removal to achieve the goal of compiling the formula.

The language of context restriction translates very naturally into a logical notation: if x is a substring that is a member of language $\mathcal{A}$, then x is the successor of a string from $\mathcal{B}$ and a string from $\mathcal{C}$ is the successor of x. Employing the n-ary successor-of predicate introduced earlier, this becomes:

$$(\forall x)\Big(x \in \mathcal{A} \rightarrow \big(S(\mathcal{B}_1, x, \mathcal{C}_1) \vee \ldots \vee S(\mathcal{B}_n, x, \mathcal{C}_n)\big)\Big) \tag{6.14}$$

and can be translated into a regular expression and a finite automaton exactly as described above.

6.5.2. Two-level rules

We will now turn our attention to the possibility of compiling the two-level formalism of Koskenniemi (1983) into finite-state automata. This is another nontrivial task which has been treated fairly extensively in the literature (see e.g. Karttunen et al. (1987) for a comprehensive description of a compilation method that uses the auxiliary symbols technique).

A two level grammar defines a subset of the language Σ_f^*, where Σ_f is the set of *feasible pairs*, defined in advance. The set Σ_f^* is constrained by the use of statements involving four operators: $a{:}b \Rightarrow l \_ r$ (saying the symbol $a{:}b$ is only permitted between l and r), $a{:}b \Leftarrow l \_ r$ (which says a symbol a occurring between the languages l and r must be realized as b), and $a{:}b/\Leftarrow l \_ r$ (saying $a{:}b$ is never allowed between l and r). The notation $a{:}b \Leftrightarrow l \_ r$ is a shorthand for the conjunction between the first two types of rules.

The feasible pairs Σ_f are also assumed to include every symbol pair occurring in some statement in the collection of grammar rules on the left-hand side.

Compiling a collection of such rules into a finite-state automaton of symbol pairs (or a transducer) is non-trivial, again precisely for the reason that the problem contains overlapping substrings. This is true in particular for right-arrow rules with multiple contexts.

However, each of the rule types are quite comfortably expressible in the logical notation proposed:

$$\begin{array}{lcl}
a{:}b \Rightarrow l \_ r & \equiv & (\forall x)(x \in a{:}b \rightarrow S(l,x,r)) \\
a{:}b \Leftarrow l \_ r & \equiv & \neg(\exists x)(x \in a{:}\overline{b} \wedge S(l,x,r)) \\
a{:}b \;/\!\Leftarrow l \_ r & \equiv & \neg(\exists x)(x \in a{:}b \wedge S(l,x,r))
\end{array}$$

Here, the negation in statements such as $a{:}\overline{b}$ refers to any symbol in the alphabet, except

b. As we consider symbol pairs to be single symbols, $a{:}\overline{b}$ is the set of single symbols such that input side is a and the output side is not b.

There is the additional practice (Karttunen et al., 1987; Beesley and Karttunen, 2003) that right-arrow rules with multiple contexts are allowed, are separated by commas, and are interpreted disjunctively; i.e. one of the contexts must hold for the symbol pair $a{:}b$ to be legal. For right-arrow rules, this prompts exactly the same solution as for context restriction above:

$$(\forall x)(x \in a{:}b \rightarrow S(l_1, x, r_1) \vee \ldots \vee S(l_n, x, r_n)) \tag{6.15}$$

For left-arrow rules and disjunctive multiple contexts, the logical specification is:

$$\neg(\exists x)\big(x \in a{:}\overline{b} \wedge (S(l_1, x, r_1) \vee \ldots \vee S(l_n, x, r_n))\big) \tag{6.16}$$

that is, in every context l_i _ r_i, a must be realized as b.

In essence, the above is a complete compilation algorithm and logical specification for two-level rules, if translated to regular expressions through the method presented above. The collection of individual rules are assumed to be intersected with each other and the set of feasible pairs. Hence, we get that a two-level grammar $\mathcal{G}$ can be compiled as:

$$\mathcal{G} = \Sigma_f^* \cap \mathfrak{R}$$

where $\mathfrak{R}$ is the intersection of the individual rules compiled through the notation presented here.

6.6. Relationship to auxiliary symbol manipulation

Let us briefly pick up the thread that was left unfinished earlier in the chapter: the notion that the logical formalism is an abstraction of a collection of techniques widely used in finite-state language processing: that of auxiliary symbol manipulation. We can go back and analyze some of the results and compilation methods advanced in the literature in terms of the logical notation and, first of all, verify that previous approaches are correct, and secondly, see that they essentially perform the same operation as compiling the equivalent logical formula would do.

6.6.1. Context restriction compilation with auxiliary symbols

A compilation method for context restriction rules of the type in (6.12) is given in Yli-Jyrä and Koskenniemi (2004) as follows. For a set of context-restriction rules

$$\mathcal{A} \Rightarrow \mathcal{B}_1 \,\_\, \mathcal{C}_1, \ldots, \mathcal{B}_n \,\_\, \mathcal{C}_n \tag{6.17}$$

we construct

$$\Sigma^* - h_{\{\diamond\}}(\Sigma^* \diamond \mathcal{A} \diamond \Sigma^* - \bigcup_{i=1}^{n} \Sigma^* \mathcal{B}_i \diamond \Sigma^* \diamond \mathcal{C}_i \Sigma^*) \tag{6.18}$$

where $\diamond$ is a symbol not included in Σ and h a homomorphism that deletes the symbol $\diamond$.[4] The authors do not discuss how the compilation formula was discovered, although their arguments for its correctness are definitely of a procedural nature. The authors also illustrate the difficulty of compiling such statements by providing an extensive appendix that analyzes the errors in many previous attempts found in the literature to reach a general-purpose formula.

To slightly ease the analysis of formula (6.18) and to help with its comparison to the logical formalism, we can restrict ourselves to the single-context case which becomes, in Yli-Jyrä and Koskenniemi's notation:

$$\Sigma^* - h_{\{\diamond\}}(\Sigma^* \diamond \mathcal{A} \diamond \Sigma^* - \Sigma^* \mathcal{B} \diamond \Sigma^* \diamond \mathcal{C} \Sigma^*) \tag{6.19}$$

[4]The original formula did not consider the contexts to be automatically extended on the left and right with Σ^*, something usually done in natural language processing applications. That addition to the formula is mine.

In our logical notation, as seen in (6.14), we would compile such an expression as:

$$(\forall x)(x \in \mathcal{A} \rightarrow S(\mathcal{B}, x, \mathcal{C})) = \tag{6.20}$$

$$\neg(\exists x)\neg(x \in \mathcal{A} \rightarrow S(\mathcal{B}, x, \mathcal{C})) \tag{6.21}$$

which by the regular expression conversion becomes

$$\neg\Pi\big((\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*) \wedge \neg(\neg(\Delta^* \textcircled{x} \mathcal{A} \textcircled{x} \Delta^*) \vee (\Delta^* \mathcal{B} \textcircled{x} \Delta^* \textcircled{x} \mathcal{C} \Delta^*))\big) \tag{6.22}$$

where $\Delta = \Sigma \cup \{\textcircled{x}\}$ and by DeMorgan's rule

$$\neg\Pi\big((\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*) \wedge (\Delta^* \textcircled{x} \mathcal{A} \textcircled{x} \Delta^*) \wedge \neg(\Delta^* \mathcal{B} \textcircled{x} \Delta^* \textcircled{x} \mathcal{C} \Delta^*)\big) \tag{6.23}$$

Inspecting the distribution of $\textcircled{x}$, and noting the fact that none of the languages $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$ contain the symbol $\textcircled{x}$, we see that the different occurrences of Δ can be reduced to Σ, yielding

$$\neg\Pi\big((\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*) \wedge (\Sigma^* \textcircled{x} \mathcal{A} \textcircled{x} \Sigma^*) \wedge \neg(\Sigma^* \mathcal{B} \textcircled{x} \Sigma^* \textcircled{x} \mathcal{C} \Sigma^*)\big) \tag{6.24}$$

and of course $(\Sigma^* \textcircled{x} \Sigma^* \textcircled{x} \Sigma^*) \wedge (\Sigma^* \textcircled{x} \mathcal{A} \textcircled{x} \Sigma^*) = (\Sigma \textcircled{x} \mathcal{A} \textcircled{x} \Sigma)$, so we have that the above equals

$$\neg\Pi\big((\Sigma^* \textcircled{x} \mathcal{A} \textcircled{x} \Sigma^*) - (\Sigma^* \mathcal{B} \textcircled{x} \Sigma^* \textcircled{x} \mathcal{C} \Sigma^*)\big) \tag{6.25}$$

that is, exactly the same expression as given in (6.19), except the symbol $\textcircled{x}$ corresponds to $\diamond$ and the operation Π to the homomorphism that deletes the $\diamond$-symbols.

6.7. Discussion

We have presented an extension to the formalism of regular expressions, and kind of regular predicate logic. It systematizes the prevalent use of auxiliary symbols in defining complicated languages in a way that is notationally clear and can be intermixed with standard regular expressions. In particular, the propositions of our regular predicate logic are freely extendable and it is assumed that one can take advantage of other finite-state calculus operators in defining new predicates.

We have also demonstrated how the notation can be used to systematically define other formalisms used in natural language processing applications using two-level rules as an example. We believe the new notation brings a level of transparency to the definition of other complex regular expression operations. We will take advantage of this notation in subsequent chapters, in particular when dealing with multitape automata where the notation and compilation method becomes extremely useful.

It is interesting to note that Kaplan and Kay (1994), in defining what they call "if-P-then-S$(\mathcal{L}_1, \mathcal{L}_2)$"—the language where every string from $\mathcal{L}_1$ is followed by some string from $\mathcal{L}_2$—as $\neg(\Sigma^*\mathcal{L}_1\neg(\mathcal{L}_2\Sigma^*))$, point out an intuition that the "double complementation in the definitions ... and also in several other expressions ... *constitutes an idiom for expressing universal quantification*" [italics mine].

In the logical formalism presented in this chapter it is the *combination* of a specific type of use of auxiliary symbols together with a double negation that constitutes an idiom for universal quantification. The double negation (taken with respect to different alphabets) then becomes an artifact of the definition of universal quantification in terms of existential quantification and the construct where variable binding is achieved through intersection in expressions such as

$$(\forall x)(\varphi) \equiv \neg\Pi\big((\Delta_x^* ⓧ \Delta_x^* ⓧ \Delta_x^*) \cap \neg(\varphi)\big) \tag{6.26}$$

The 'if-P-then-S'-type functionality of Kaplan and Kay (1994) in a way marks the boundary of what is definable easily through standard regular expressions. As we move beyond such statements in terms of complexity, other methods become necessary. The logic presented in this chapter is one such method, and we find that many previous approaches that use auxiliary symbol manipulation really reduce to this.

7. MULTITAPE AUTOMATA

7.1. Introduction

In this chapter we will introduce a method for encoding multitape automata (or k-ary transductions) with the standard representation of finite-state automata and present a set of operations with which the content of such multitape automata can be manipulated and constructed.

The motivation for introducing and exploring the properties of multitape automata is twofold. First, as we will explore more in-depth in chapter 8, the ability to manipulate multitape automata provides a useful construction aid for ordinary automata and transducers, in particular for such automata and transducers that would be difficult to construct using standard methods.

Second, multitape automata—as has been noted in the literature (Kay, 1987; Kiraz, 1994, 2000)—contain features that make them attractive as such for the treatment of phonology, morphology, and other linguistic tasks. The ability to model nonlinear features and hierarchies by placing different types of material on different tapes in a multitape automaton corresponds intuitively to many popular theoretical models of phonology and morphology. Modeling natural language phenomena with multitape automata is a suggestion that has occasionally surfaced in the literature, usually in the context of how to model nonconcatenative morphologies. However, a coherent perspective on multitape automata and tools to handle them has been lacking to the extent that they are now usually perceived as arcane devices which are theoretically interesting, but difficult to construct and maintain in practice. We will address this by developing a unified model of multitape automata and the tools to manipulate them. This model is a combination of the logical formalism developed in chapter 6 and the multitape model developed in the current chapter. After presenting the model, the separate chapter 9 will outline the development of actual grammar applications based on it.

In this chapter, we will first, in section 7.2, look at previous work and consider some different possibilities for working with multitape automata, after which the proposed model is discussed in detail in section 7.3. In section 7.4 we will consider some basic operations of manipulating multitape automata that our construction allows. The possibilities of Boolean operations are discussed in section 7.5. Different tape-manipulation operations such as extraction and insertion of tapes are presented in section 7.6. We will also want to express constraints on the content of tapes vs. other tapes and their alignment. The basic tools for performing this are given in section 7.8. Some additional operations related to parsing mechanisms and conversion to and from finite-state transducer representations are given in sections 7.9 and 7.10.

7.2. Previous work and perspectives

Before we settle on an actual method for handling multitape automata, let us now briefly consider some of the options available for a treatment of multitape automata or k-ary transductions. One obvious strategy would probably be to extend the definition of a finite-state transducer directly to perform a k-ary transduction. That is, we would simply modify the transition function in the transducer model to perform a mapping from $Q \times \Sigma_1 \times \ldots \times \Sigma_k$ to Q. From a graph-based point of view, this model would entail no change to the state-transition model we have been considering up until now, with the exception that transition labels would be k-ary, i.e. of the form $s_1{:}s_2{:}\ldots{:}s_k$.

There are two obvious drawbacks to such an approach. The first is that in extending the transducer model in this fashion, we would not be able to take advantage of the operations on automata and transducers developed so far, and would have to modify each algorithm to generalize to k-ary transitions. The second shortcoming arises from complexity concerns. There is a fundamental space complexity problem with k-ary transduction models, which is that when the number of tapes grows, the required joint symbol alphabet—exhibited on the transitions—grows with exponential rapidity unless special mechanisms are devised to curtail this. This explosion in the number of transitions in a k-tape automaton can in many cases be more severe than the growth in the number of states of a complex grammar. To take a simple, concrete example of an operation that would seem to arise naturally in a language processing setting: suppose we have a 5-ary transduction model, each 'tape' consisting of the same alphabet of, say, 22 symbols $\{s_1, \ldots, s_{22}\}$. Such an alphabet is not improbable in natural language setting. Now, assume we want to restrict the co-occurrence of s_1 on any combination of tapes, meaning s_1 can only occur once on one tape in the same position, i.e. we would be accepting any strings containing a symbol such as $s_1{:}s_2{:}s_2{:}s_2{:}s_2$ or $s_2{:}s_2{:}s_2{:}s_2{:}s_3$ but not, $s_1{:}s_2{:}s_3{:}s_4{:}s_1$. Without further treatment of the alphabet behavior, this yields a k-ary transducer which has a single state, but 5,056,506 transitions—each transition representing a legal combination of symbols on the five tapes. This kind of transition blow-up is not completely inevitable: of course one can devise many tricks to avoid it, such as adding certain semantics to the transition notation—in our example by perhaps having a special type of 'failure' transition which leads to non-acceptance. For the above example this would cut down the number of transitions from 5,056,506 to 97,126. The drawback with such advanced methods is that any changes will tend to affect the entire finite-state system one is working with, requiring adaptations in almost every underlying algorithm to construct automata. One, again, is then unable to leverage the power of existing algorithms designed for finite-state machine construction, but needs to build special-purpose algorithms for whatever transduction model one has in mind.

Another possibility is to directly consider established models for multitape automata such as the Rabin-Scott model discussed in section 5.7.3 on page 160. The weak point in such models is that they are deliberately deterministic in ways that inhibit one from representing such ambiguities that would be desirable in natural language.

In contrast to the two models above, the one we shall pursue here is based on a *simulation* of multitape automata by a standard deterministic finite automaton. We do so by defining an isomorphism between single-tape automata and multitape automata. Simulations of multitape machines by single-tape ones most often come up in proofs and gedankenexperiments regarding the equivalence of different models of computation, but are more rarely employed in practice (Aho and Ullman, 1972; Kozen, 1997; Sipser, 2006). The immediate advantage is that by doing a simulation, rather than dealing with an entirely new model, we may fall back on the algorithms already established and use those to develop new ones that fulfill our needs in the treatment of multitape automata.

7.3. Encoding

As mentioned above, our encoding of multitape automata on a single tape is similar to simulations often used in proofs of the equivalence of multitape and multihead Turing machines to single-tape and single-head ones. In particular, we shall abstract slightly away from the details of transitions and states, and concern ourselves with simulating the set of accepted tape-combinations of a k-tape automaton with a single-tape one.

The fundamental notion in the simulation is this: we consider a multitape automaton MT of k tapes, where each tape contains strings over the alphabet Σ, and that only accepts tape combinations where the content all tapes are of equal length n. If this multitape automaton MT accepts a given set of strings over Σ^* in the configuration

$$\{s_1^1 s_1^2 \dots s_k^n\} \dots \{s_k^1 s_k^2 \dots s_k^n\}$$

we represent this configuration as a single string $\{s_1^1 s_2^1 \dots s_k^{n-1} s_k^n\}$. That is, for each accepted combination of tapes in the k-tape automaton, we can convert this combination into a string over a 1-tape automaton by 'collecting' the symbols by traversing the tapes in a zig-zag fashion across columns and rows as seen in figure 7.1.

Since we are interested in being able to define configurations where the contents of the different tapes are of different length, we reserve a special semantics for the symbol $\{\Box\} \in \Sigma$, which can be thought of as marking a 'blank' on a tape.

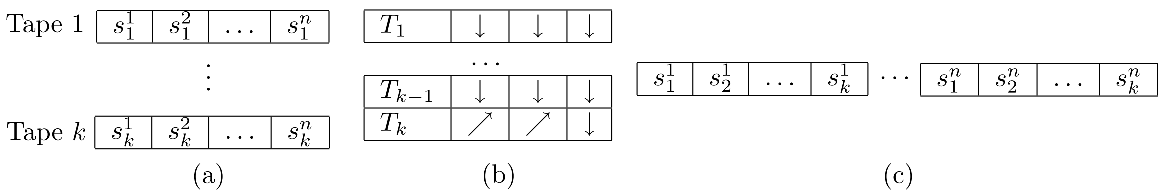

Tape 1
Tape k
s_1^1 s_1^2 … s_1^n
s_k^1 s_k^2 … s_k^n
T_1
T_{k-1}
T_k
s_1^1 s_2^1 … s_k^1 … s_1^n s_2^n … s_k^n
(a)
(b)
(c)

FIGURE 7.1. *Encoding a k-tape multitape automaton configuration in (a) using traversal pattern (b) with the single tape in (c).*

To illustrate the representation in the single-tape encoding, the two-tape different-length configuration:

T_1	a	
T_2	b	c

would be represented as the string $ab\Box c$.[1]

Requiring blanks to be marked allows us to keep the tapes synchronized and thus also permits the encoding we suggest. On the other hand, this also requires special precautions in treatment of these automata, as we shall see later.

7.4. Basic operations

Let us first consider the possibility of concatenating, taking the union, or the Kleene closure of two acceptable configurations of a multitape automaton.

It follows fairly straightforwardly from the encoding we have chosen that all the standard operations of $\{^*, \cdot, \cup\}$ apply directly to k-tape configurations as well. That is, if we have two sets of k-tape configurations $\mathcal{C}_1^k$ and $\mathcal{C}_2^k$, their concatenation, union, and Kleene closure can be directly applied to the one-tape representations $\mathcal{C}_1^1$ and $\mathcal{C}_2^1$ using standard algorithms as well. This can be shown by a simple induction of the operations.

[1]This use of tape blanks is very similar to the use in two-level morphology (Koskenniemi, 1983).

7.5. Boolean operations

Similarly to the above, the Boolean operations which can be applied to a single-tape automaton simulating a multi-tape one, yield the correct results from the perspective of tape configurations. Since we are forcing the length of the content of the different tapes to be equal (by including 'blank' symbols), the complement and intersection operations, however, are valid with respect to a particular configuration and alignment of the blanks, and no other.

7.6. Operations on tapes

From a practical perspective, the most important operations we can perform have to do with the manipulation of individual tape contents in a k-tape machine. This includes the possibility of extracting from a k-tape automaton the entire contents of a tape, producing a $k - 1$ tape automaton, or inserting a stand-alone tape, producing a $k + 1$-tape automaton, or swapping the contents of individual tapes.

To perform these operations, we shall take advantage of the methods already introduced in chapter 3, in particular the combined operation of composition and domain extraction, which allows us to perform a variety of tasks directly through regular expression calculus without low-level manipulation of states and transitions.

7.6.1. Tape insertion and removal

Tape insertion, as we shall define it, is the task of, given a set of strings, represented as a regular language L, and an automaton M^k, which is a single-tape representation of a k-tape automaton, arbitrarily inserting the contents of L as the ith tape, resulting in a $k + 1$-tape automaton. Tape removal is the inverse of tape insertion: from a k-tape automaton, extract tape i, producing a $k - 1$ tape automaton.

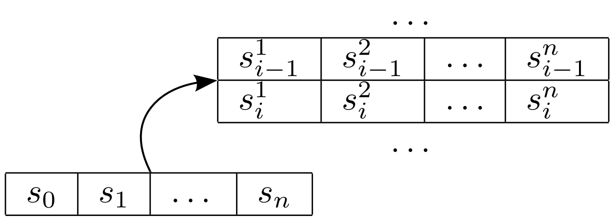


FIGURE 7.2. *Inserting a single tape as the ith tape in a k-tape automaton.*

Before we look at the details of tape insertion and extraction, let us first consider how to talk about the contents of a single tape, the ith tape, disregarding the possible contents of the other tapes in a k-tape automaton. Since two symbols in a k-tape automaton are on the same tape if their positions are congruent modulo k, we can express the content of tape i as L with respect to a k-tape machine as:

$$\mathfrak{T}_k^i(L) \equiv \text{range}(L \circ ((\epsilon{:}\Sigma)^{i-1}\ \Sigma\ (\epsilon{:}\Sigma)^{k-i-1})^*) \tag{7.1}$$

Note that this allows us directly to construct a k-tape automaton from individual single-tape automata by declaring the contents of each tape independently and intersecting the different tapes. For example, supposing we have sets of strings $L_1 \dots L_4$ and want to construct a 4-tape automaton where each of the tapes i contains only strings from L_i, this can be done by:

$$\mathfrak{T}_4^1(L_1) \cap \mathfrak{T}_4^2(L_2) \cap \mathfrak{T}_4^3(L_3) \cap \mathfrak{T}_4^4(L_4) \tag{7.2}$$

Let us now consider a k-tape machine M and a set of strings L which we want to insert as the ith tape of M as in figure 7.2. We can achieve this in a fairly intuitive way by first 'expanding' M to include an ith tape, the contents of which are arbitrary, displacing the old tapes, and then intersecting this with $\mathfrak{T}_{k+1}^i(L)$, i.e.

$$\text{insert}(L, M^k, i) \equiv \text{range}(M \circ (\Sigma^{i-1}\ (\epsilon{:}\Sigma)\ \Sigma^{k-i+1})^*) \cap \mathfrak{T}_{k+1}^i(L) \tag{7.3}$$

In a similar way, we can perform the tape removal: from a k-tape machine, remove the ith tape, producing a $k-1$ tape machine:

$$\text{remove}(M^k, i) \equiv \text{range}(M \circ (\Sigma^{i-1}\ (\Sigma{:}\epsilon)\ \Sigma^{k-i})^*) \tag{7.4}$$

Tape extraction—or projection—is the operation where we simply want to, from a k tape automaton, extract the contents of the ith tape, producing a single-tape automaton (language). It can be defined as follows:

$$\text{extract}(M^k, i) \equiv \text{range}(M \circ ((\Sigma{:}\epsilon)^{i-1}\ \Sigma\ (\Sigma{:}\epsilon)^{k-i})^*) \tag{7.5}$$

We can also define an operation of tape swapping. Given a k-tape machine, we swap the contents of tape i and tape j, not otherwise altering their content. This is defined as:

$$\text{swap}(M^k, i, j) \equiv \text{range}\big(M \circ (\bigcup_{s \in \Sigma \times \Sigma} (\Sigma^{i-1}\ s\ \Sigma^{j-i-1}\ s^{-1}\ \Sigma^{k-j}))^*\big) \tag{7.6}$$

assuming $i < j$. The strategy of the swap is to use as an intermediary a transducer that performs a mapping $a{:}b$ on each tape i and, later in the same 'column', when reaching the symbol on tape j, maps $b{:}a$.

7.7. Alignment of tapes

In the above operations we have implicitly assumed that in constructing and manipulating a k-tape automaton, the individual tapes are of equal length. Or, alternatively, the languages L which we combine through tape insertion, extraction, etc., already contain the appropriate number of blanks, such that combining several different-length tapes by intersection and the $\mathfrak{T}$ operation as in (7.2) yields something other than the empty language.

There are several possibilities at hand if we want to allow arbitrary alignments in the above operations. If we have a language L which we want to insert in a k-tape machine, and supposing L contains many different-length strings without $\Box$-blanks, we can modify

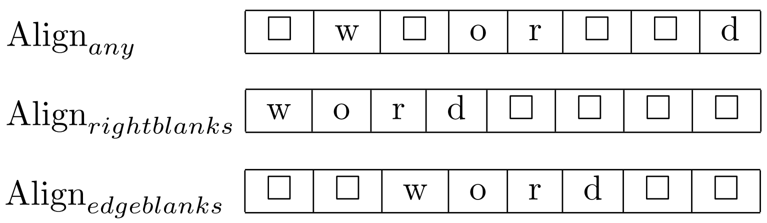


FIGURE 7.3. *Three example alignment strategies for words on multitape automata.*

L so that it either allows blanks to occur at any location or aligns the blanks in a particular way. Let us first consider the 'any'-type of alignment. Given a language L, we can produce L such that blanks occur at any location by simply declaring:

$$\text{Align}_{any}(L) \equiv \text{range}(L \circ (\Sigma \cup (\epsilon{:}\Box))^*) \tag{7.7}$$

Alternatively, we may want all the blanks to align after the 'real' symbols in L.

$$\text{Align}_{rightblanks}(L) \equiv \text{range}(L \circ \Sigma^*(\epsilon{:}\Box)^*) \tag{7.8}$$

Or, we may want to allow for blanks either in the beginning or in the end of the tape's contents, such that the actual symbols occur consecutively, without intervening blanks:

$$\text{Align}_{edgeblanks}(L) \equiv \text{range}(L \circ (\epsilon{:}\Box)^* \ \Sigma^* \ (\epsilon{:}\Box)^*) \tag{7.9}$$

Naturally, there are a number of strategies we could pursue in aligning blank symbols with non-blank symbols. The most useful ones are probably the three discussed above, illustrated in figure 7.3.

7.8. Constraint operations

All the operations to construct multitape automata from single-tape ones discussed above are such that the contents of the tapes remain independent of one another regardless of construction. In effect, we have not yet developed any machinery with which, given our encoding, we can express co-occurrence restrictions across tapes. This is something we will want to do because the ability to construct multitape automata from different elements where the tapes interact with each other is fundamental for future applications we have in mind.

Some natural constraints that we would obviously like to express over different tapes include concepts such as:

(i) If a string x occurs on tape i, it must/must not be aligned with a string y on tape j

(ii) If a string x occurs on tape i, it must/must not be preceded/followed by a string y on tape j

(iii) If a string x occurs on tape i, there must/must not exist a string y anywhere on tape j

Also, we would perhaps like to express boolean combinations of all of the above existential statements.

Developing a separate formalism and ways to express such constraints through combinations of the basic operations is a difficult task. The situation is analogous to the problem statements in chapter 6, where we were faced with having to define increasingly complex regular languages—but this time with the additional complication of keeping track of multiple tapes and the possible alignment of blanks as well.

7.8.1. Predicate logic

Fortunately, the adaptation of the predicate logic formalism in chapter 6 to the current problem is relatively uncomplicated. Indeed, the only modification required in the conversion of logical statements in that chapter (cf. page 182) regards the definition of the regular expression equivalent of the existential quantification over some variable, i.e. $(\exists x)$.

Recall that the function of the existential quantifier regular expression equivalent was to isolate a particular substring in a language L and label it $\textcircled{x}$ so that subsequent propositions, statements, and boolean combinations thereof could be identified as referring to the same

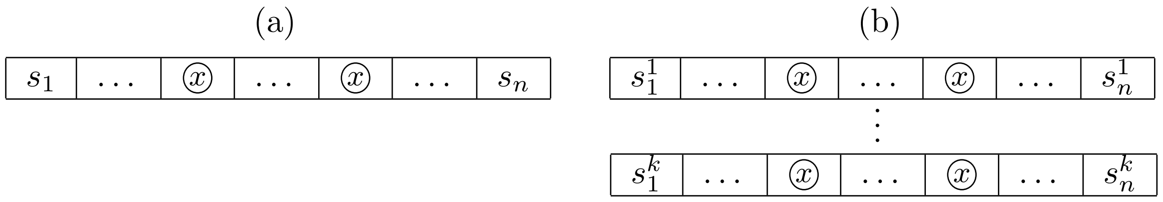


FIGURE 7.4. *Existential quantification with a single tape (a) and multiple tapes (b).*

substring. The situation we are faced with now requires that, as we will want to quantify over substrings which may have different content in different tapes, the symbol $ⓧ$ must be distributed across all tapes (see figure 7.4). This requires that $ⓧ$ mark an entire column, and so the regular expression equivalent must consist of k consecutive $ⓧ$-symbols. To this end, we can generalize the regular expression equivalent of $(\exists_k x)$, k being the number of tapes we are dealing with, and where the case $k = 1$ remains exactly as defined in chapter 6, i.e. $(\exists x)$. Hence,

$$(\exists_k x) \equiv ((\Delta_x^k)^* ⓧ^k (\Delta_x^k)^* ⓧ^k (\Delta_x^k)^*) \tag{7.10}$$

And, as we have defined $(\forall x)$ in terms of existential quantification, this construction requires no change for capturing $(\forall_k x)$.

Leveraging the possibilities offered by the predicate logic formalism we can now construct statements that dictate co-occurrence restrictions of arbitrary complexity, such as those given in the above examples (i)—(iii). Let us look at some actual examples of how to express these:

- Every occurrence of L on tape i should coincide with L' on tape j

  $(\forall_k x)(\mathfrak{T}_k^i(x \in L) \rightarrow \mathfrak{T}_k^j(x \in L'))$

- Every occurrence of L on tape i should be preceded by L' on tape j

  $(\forall_k x)(\mathfrak{T}_k^i(x \in L) \rightarrow \mathfrak{T}_k^j(S(L', x)))$

- If L occurs on tape i, L' does not occur anywhere on tape j

  $(\exists_k x)(\mathfrak{T}_k^i(x \in L) \rightarrow \neg(\exists_k y)(\mathfrak{T}_k^j(y \in L')))$

- Every occurrence of L on tape i is either aligned with L' on tape j, or preceded by L'' on tape l

  $(\forall_k x)(\mathfrak{T}_k^i(x \in L) \rightarrow (\mathfrak{T}_k^j(x \in L') \vee (\mathfrak{T}_k^l(S(L'', x)))))$

The same observations as before apply to the distribution of blanks, which may or may not be enforced whenever making logical statements over multiple tapes. If we want to restrict the occurrences of some set of strings L using the predicate logic, we may specify precisely under which interpretation of the distribution of blanks the propositions shall hold. Suppose L consists of the single string ab. Now, if in conjunction with propositions we wish to say something about L, we can use the alignment functions above to specify whether constraints shall apply only to the string ab on a tape, or also to the various possibilities $\square ab$, $a\square b$, $ab\square$, $\square\square ab\square$, etc., etc. Naturally, the same holds for any language we discuss over the various tapes, such as any of the L'-languages discussed in the above examples.

7.9. Conversion to and from transducers

In the event that we have a complex k-tape automaton, the ability to extract a transducer from 2 tapes in that automaton is useful. As we have decided, because of complexity concerns, to deviate from the transduction model in going from 2-ary transduction to k-ary ones, converting a 2-tape automaton to and from a finite-state transducer requires some special treatment. A k-ary transducer where the model is an automaton where transitions are symbol vectors $< s_1, \ldots, s_n >$ can of course trivially be converted into a transducer. But our model, where the contents of different tapes are modeled in transitions in different states, requires a slight manipulation of the machines with methods that lie outside the regular expression calculus.

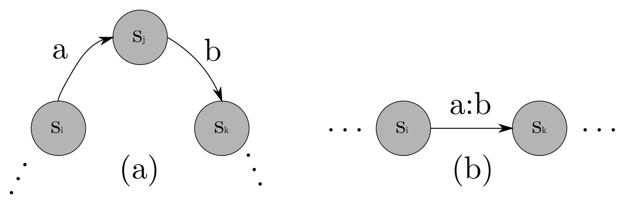


FIGURE 7.5. *Converting a 2-tape automaton in (a) to a finite-state transducer.*

In going from a 2-tape representation to a transducer, we know that every other symbol in the language corresponds to one of two tapes, odd-numbered ones being tape 1, and even-numbered ones tape 2. Hence, we can traverse the 2-tape automaton (breadth-first or depth-first) and construct from state sequences s_i, s_j, s_k (where s_i is an even position) the equivalent transducer states s_i, s_k, and replace the labels in the transducer with the labels going from $s_i \rightarrow s_j : s_j \rightarrow s_k$ (see figure 7.5). In the inverse construction we do the opposite, namely for any label $a{:}b$ going from a state s_i to s_k, we introduce a state s_j, and a transition on a from s_i to s_j and one with b from s_j to s_k.

Naturally, we need to take the blanks into account as well and convert those into ϵ or vice versa before the conversion procedures.

7.10. Parsing

The final question we will want to address is that of 'parsing' with a k-tape automaton. This task could be defined as first calling one of the k tapes an input tape and the remaining $k-1$ tapes output tapes, then asking the question, what are the possible configurations of the output tapes in M^k, given that the input is w? This can naturally be answered by first calculating

$$\mathfrak{T}_k^i(w) \cap M \tag{7.11}$$

for tape i declared as the input tape, and subsequently extracting all the words from the resulting automaton. Of course we may want to consider alternate alignments, or perhaps all the possible ones, of the word w in which case we again would modify the automaton representing w before performing the parse. Again, using one of the above *Align*-functions will make this possible.

The other, more low-level, possibility is to consider an algorithm for doing this task. In this case we may want to declare the input tape to be the first one, or by swapping tape contents make it so if it is not already. Subsequently, since every kth symbol corresponds to the input, we can by standard graph search techniques match the input against every kth symbol and in such fashion extract the output.[2]

This second possibility may in practice be more efficient, but not necessarily asymptotically so, as a simple analysis reveals. It is easy to see that an 'input' word with any alignment of blanks can be represented by an automaton with $|w| + 1$ states. Also, the language

$$\mathfrak{T}_k^i(\text{Align}_{any}(w))$$

requires an automaton of $k(|w| + 1)$ states. Hence, the complexity of the entire operation is of the order $|M|k(|w| + 1)$. Since $|M|$ and k are constant, the entire operation grows linearly in proportion to $|w|$. Extracting the set of legal words from the result automaton can of course be performed in linear time with respect to its size as well using standard techniques.

7.11. Discussion

In this chapter we have discussed a method for representing k-tape automata with the single-tape model, the basis of which has been laid out in the previous chapters. The

[2]Ron Kaplan (p.c.) reports that in unpublished work on a similar interleaving encoding for 2-tape automata (transducers), it was observed that a low-level search algorithm for parsing was efficient in the forward direction, but very inefficient in the inverse direction (where the contents of tape 2 is supplied). This suggests that one may need to store multiple representations of a grammar for efficient parsing and generation. Of course these different representations can be constructed by the tape-swapping operation after a grammar has been constructed.

foremost advantage with simulating a k-tape automaton is that all of the basic operations regarding construction transfer painlessly into the k-tape model. An example of this was seen with implementing the most difficult operations with regard to multitape automata—that of dictating constraints that should hold across different tapes. These could with very little modification be accommodated by the single-tape predicate logic introduced in chapter 6.

The purpose with the chapter has not been to consider the construction and manipulation of multitape automata as an end to itself, but rather, lay the formal foundations for two types of applications we shall see in later chapters: that of defining complex transducers and automata with multitape automata as an additional intermediate tool, and to define natural language grammars directly in terms of multitape automata.

8. STRING REWRITING AND MULTITAPE AUTOMATA

8.1. Introduction

In this chapter we will present a method for compiling different types of string rewriting rules to finite-state transducers. The ability to create finite-state transducers from abstract descriptions of string rewrite rules is the cornerstone of many tasks in finite-state language processing. It means that we can, for instance, construct morphological and phonological grammars where finite-state transducers are composed in a cascade that map an abstract underlying form to actual surface strings as well as construct shallow syntactic parsers, part-of-speech disambiguators, tokenizers, phrase identifiers, and spelling correctors. String rewriting is also a prerequisite for encoding other linguistic formalisms, such as Optimality Theory grammars (Karttunen, 1998; Gerdmann and van Noord, 2000), or Realizational Morphology (Karttunen, 2003), as finite-state transducers. In addition, the ability to encode a basic string rewriting formalism into a transducer allows one to build a layer of increasingly complex new operators, which is useful for a variety of tasks.

The task of encoding string rewriting rules goes back to the work of Johnson (1972), who showed that the alternation rules in the *Sound Pattern of English* (Chomsky and Halle, 1968) could in principle be encoded as finite-state transducers. Although this discovery went unnoticed at the time, the idea was independently rediscovered in the early 1980s by Kaplan and Kay, whose work was finally published in Kaplan and Kay (1994). Although many finite-state systems have been built around the two-level morphology of Koskenniemi (1983), an increasing amount of work since the 1980s has been accomplished with various string rewriting formalisms.

Not all aspects of string rewriting, as it was defined in early work of generative phonology, can be exactly modeled using finite-state transducers. For instance, cyclic rules, where rules keep repeatedly applying to a string until the string has changed to something where the rule can no longer apply, are outside the realm of finite-state transducers. The actual effect of most cyclic rules, however, can be achieved by a suitable choice of noncyclic rules (Sproat, 1992). On the other hand, there are a number of surprisingly powerful variants of string rewriting that can be encoded as a transducer—variants that have been discovered by analysis of finite-state transducers themselves and do not stem from an effort to imitate a formalism of theoretical linguistics. Many such modifications to the basic idea of string rewriting have found a place in the toolkit of the working computational linguist and contribute to the relative conciseness by which a finite-state notation can express complex linguistic phenomena.[1]

For all but the simplest types of string rewriting rules, compilation into transducers is a very complex task and needs to be broken down into smaller pieces. This chapter is laid out as follows. In section 8.2 we look at previous work addressing the topic. We then give a general outline of the method presented in this chapter, without going into too much detail, in section 8.3. Next we turn to the overall compilation algorithm in detail in sections 8.4 through 8.6, after which we consider various types of different rewriting logics in sections 8.7 through 8.11. We summarize the compilation method in section 8.12 and conclude with a comparison to previous work in section 8.13 as well as general discussion in section 8.14.

8.2. Previous work

The first work that systematically laid out how to compile transducers that encode SPE-style rewrite rules using a small number of regular language operations was Kaplan and Kay (1994). This publication, based on the authors' work from the early 1980s, also presented a number of different interpretations of string rewriting rules and reviewed thoroughly the tacit assumptions that were present in the phonological literature whenever rewrite rules were discussed. Subsequently, work stemming from Xerox (published in Karttunen (1996);

[1]For example Beesley and Karttunen (2003) show how various phrase-chunking operations can be performed with longest-match type replacement rules, and Hulden (2006) shows that many syllabification processes in phonological descriptions are naturally expressed with shortest-match rules.

Kempe and Karttunen (1996); Karttunen (1997) among others) documented a number of additions and new techniques of string rewriting. Most importantly, the contents of these papers were also implemented in actual software designed at Xerox for the development of phonological and morphological grammars. Mohri and Sproat (1996) described another compilation algorithm for basic rewrite rules, although no implementation was made available.

Our approach differs from Kaplan and Kay and the work of Kempe and Karttunen in that the construction method here is not based on manipulation of finite-state transducers. These earlier approaches have all defined string rewriting through a series of compositions of regular relations—relations that insert predefined marker symbols, constrain their occurrence, replace strings with other strings, and remove marker symbols—in effect changing a relation in small steps until reaching the desired result. Here, by contrast, we define a multitape automaton of three tapes that models the rewrite rules we want to construct a transducer from in a more abstract way. At first, the relation defined by the multitape automaton is underdefined: it defines rewrites where none should occur, and allows for rewrites where they are not warranted. Subsequently, a set of filters are applied to this overgenerating rewrite model, which removes the unwanted relations. During this construction, we never remove individual symbols or insert symbols as previous approaches have done, but simply filter out illicit relations. In a way this approach is more static: once a preliminary relation is declared that models rewriting, we simply remove illegal relations without intermediate steps of inserting and constraining auxiliary symbols to identify which relations are to be removed. This multitape automaton is constructed in such a way that it is easy, after compilation, to convert it to a finite-state transducer.

8.3. General method

Let us first look at the general method we shall pursue for converting a replacement rule into a transducer. At this stage we shall not be concerned with actually constructing the intermediate regular languages that are necessary for the approach, but will merely look at an outline of the strategy and logic behind the construction method. After the general approach is clear, we shall delve into the specifics of each step and examine the various possibilities of expressing a variety of specific types of string-rewriting rules.

We are here concerned with rules of the general format

$$\phi \to \psi \;/\; \lambda \_ \rho \tag{8.1}$$

where the arguments ϕ and ψ may be arbitrary regular languages. At this stage, we ignore details such as the particular 'mode' of rule application (left-to-right, right-to-left, choose-longest-match, optional vs. obligatory rewriting, etc.) or the possibility of having several rules apply at the same time, and only give an outline the general procedure by which a transducer can be constructed. We shall assume, though, that what is to be constructed is a transducer where strings from ϕ, if they occur in the proper conditioning environment, will be mapped to strings from ψ.

In order to achieve this, we will use a multitape intermediary during construction. The overall method will proceed in three general steps:

(1) We convert a set of rewrite rules to a generic multitape representation that encodes string replacements from ϕ to ψ, but where these replacements occur in arbitrary positions.

(2) We constrain the multitape automaton created from (1) in such a way that it represents only those rewritings that are licensed by a rule.

(3) We convert the constrained multitape automaton into a finite-state transducer.

See figure 8.1 for all illustration of the overall approach.

The multitape encoding we use is the one presented in chapter 7, where we represent the multitape automaton as a single-tape one with the symbols from each tape interleaved evenly.

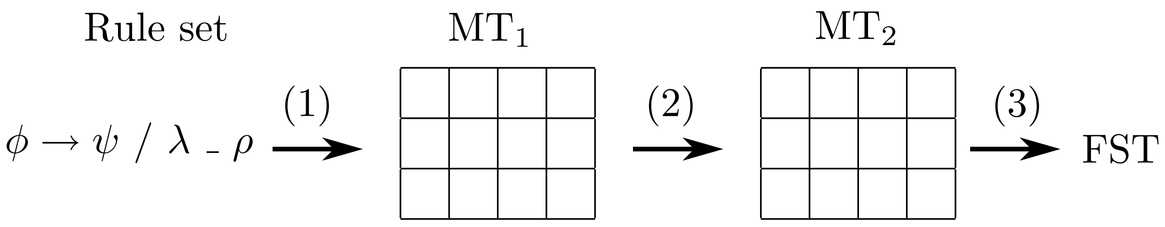


FIGURE 8.1. *General procedure for converting string rewriting rules to finite-state transducers.*

The representation in the scheme will be a very specific one. In fact, we use three tapes to encode string replacement. The idea is that in step (1) we construct a multitape automaton where tapes 2 and 3 represent the input and output respectively, and where tape 1 always carries extra symbols that signify whether the current symbol on tape 2 and 3 are participating in a rewrite rule or not. In short:

- Tape 1 contains auxiliary semantic symbols to aid us in filtering out illegal rewrites.
- Tape 2 contains the possible input strings together with a smaller set of auxiliary symbols different from those on tape 1.
- Tape 3 contains the possible output strings, aligned with tape 2, as well as auxiliary symbols.

The alignment of strings on tape 2 and 3 is exactly the alignment we will use when converting this multitape automaton to a transducer in the final step.

The auxiliary symbols we use are the following, together with their semantics:

@**0**@: represents a blank (and may occur on tapes 2 and 3)

@**#**@: represents end of string/beginning of string and occurs only on tape 2

@**ID**@: represents an identity symbol (only occurs on tape 3)

@**O**@: marks sequences outside the action of a rule on tape 1

@I@: marks sequences inside the action of a rule on tape 1

@I[@: marks the first symbol in the action of a rule on tape 1

@I]@: marks the last symbol in the action of a rule on tape 1

@I[]@: marks the first and last symbol in the action of a rule on tape 1

8.3.1. The candidate automaton MT$_1$

In constructing the 'candidate' 3-tape automaton in step (1), we shall follow a very specific procedure in aligning the tapes and their contents. In particular, MT$_1$ only allows 3-tape configurations where:

- Every 3-tape string begins and ends with $\langle$**@O@**, **@#@**, **@ID@**$\rangle$
- The intervening 'columns' alternate arbitrarily between single-symbol identity relations and rewriting sequences

We encode the single symbol identity relations as the triplet $\langle$**@O@**$, a,$ **@ID@**$\rangle$ for some symbol a where the symbol **@O@** occurs on tape 1 aligned with a on tape 2 and the symbol **@ID@** on tape 3. There is a special reason for not using the triplet $\langle$**@O@**$, a, a\rangle$ for this purpose, as will become clear when we move to the details of the construction. In fact, a three-tape triplet $\langle$**@O@**$, a, a\rangle$ is never allowed.

Rewriting sequences are encoded by placing the input/output pair of symbols on tapes 2 and 3, where the shorter one of the two strings is padded with the zero symbols **@0@** at the end.

Tape 1 contains the symbol **@I[@** for every first symbol in a rewriting sequence, the symbol **@I]@** for every last symbol of a rewriting sequence, the symbol **@I[]@** if the symbols on tape 2 and 3 are both the first and last symbols in a rewriting sequence, and the symbol **@I@** to mark all characters in between.

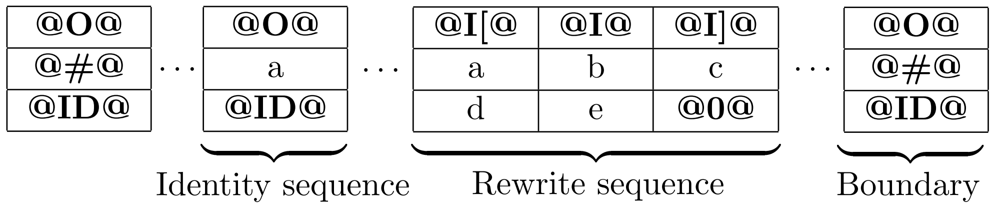


FIGURE 8.2. *Example configuration of 3-tape rewrite representation MT_1.*

In effect, tape 1 annotates, for every symbol, whether that symbol is in the process of being rewritten or not, and if it is, what stage the rewriting is at. In this way, we know from looking at just tape 1 symbols if we are in a substring position which is not being rewritten at all (by the presence of @**O**@-symbol), or if it is being rewritten, what stage the rewriting process is at (by the **I**-symbols).

For example, if we are encoding a rewrite rule:

$$abc \rightarrow de \tag{8.2}$$

ignoring for the time being the possible conditioning environment, the configuration sequence in a rewrite rule will be

@**I**[@	@**I**@	@**I**]@
a	b	c
d	e	@**0**@

(8.3)

Note, in particular, that we have padded the shorter of the two strings with the symbol @**0**@.

The 3-tape machine MT_1 will then define the set of 3-tape paired strings that begin and end with a boundary marker on each tape, and that otherwise alternates arbitrarily between identity sequences and rewriting sequences (see figure 8.2).

8.3.2. Filtering out incorrect configurations from MT_1

The output of step (1)—MT_1—in a somewhat abstracted way represents a set of string pairings such that ϕ is paired with ψ in arbitrary locations. Also, interspersed with these pairings are arbitrary sequences of single-symbol identity relations. Converting this to a transducer would simply produce the relation where ϕ is rewritten as ψ in arbitrary places, and where every instance of ϕ can also remain unrewritten. Of course, no other translations would be allowed by such a transducer. This is the point where we need to make further restrictions on the possible configurations of MT_1 as it wildly 'overgenerates.'

The next task is to filter out (a) all the occurrences of ϕ on tape 2 that are not rewritten as ψ when they should be and (b) to assure that rewrite sequences only occur where they are warranted.

Let us postpone the examination of condition (b) for the moment and focus on (a), that ϕ on tape 2 must be rewritten whenever the environment for its rewriting is fulfilled.

In order to advance the description, let us assume for the sake of exposition that in a rewrite rule:

$$\phi \rightarrow \psi \ / \ \lambda \ \_ \ \rho \tag{8.4}$$

the conditioning environment λ and ρ are regular languages, and that the logic of the rule is such that λ and ρ must be fulfilled on the left and right-hand sides of ϕ with respect to the original input string. In our 3-tape encoding, that means that for a string rewrite to be warranted, tape 2 must contain λ and ρ to the left and right of a string from ϕ. We shall see that there are many other modalities of rewriting that can be defined, but for the moment, let us settle for this one.

In order to hinder the possibility that a string from ϕ remains unrewritten when it should not, we need to filter out such strings from MT_1 where ϕ is unrewritten with λ occurring to the left of it and ρ occurring to the right. In constructing such a filter, the first question is: how do we know when ϕ is not rewritten? Since we have declared the strings of MT_1 in such a way that unrewritten sequences always align with the special symbol **@O@** on tape 1, this part of the task is quite easy. So, what we want to rule out are sequences such as:

Any	**@O@**	**@O@**	**@O@**	Any
λ	ϕ_1	$\ldots$	ϕ_n	ρ
Any	Any	Any	Any	Any

(8.5)

that is, any sequence where

1. An instance of ϕ occurs on tape 2, aligned with **@O@** on tape 1 throughout
2. That instance of ϕ is preceded by λ on tape 2, disregarding the contents of tape 1 at λ
3. That instance of ϕ is followed by ρ on tape 2, disregarding the contents of tape 1 at ρ

Clearly, if ϕ occurs in such an environment, it could also have been legitimately rewritten; therefore, ϕ should not occur in such an environment (unless the rewrite rule is designated as ‘optional’).

We can now turn to condition (b), that ϕ should be rewritten only if it occurs in the proper conditioning environment. Since we have marked the first symbol of a rewrite sequence on tape 1 by either **@I[@** or **@I[]@** we are interested in allowing a tape configuration

@I[@ or **@I[]@**	$\ldots$	**@I]@** or **@I[]@**
ϕ		
ψ		

(8.6)

only if that configuration is preceded by λ on tape 2, and followed by ρ on tape 2. Here, the schematic indicates that **@I[@** or **@I[]@** shall align with the first symbol of ϕ and ψ and **@I]@** or **@I[]@** with the last with symbols from **@I@** possibly intervening on tape 1.

Combining the two requirements (a) and (b), and removing illegal configurations from MT_1, then produces a 3-tape automaton encoding where strings on tape 2 (representing possible inputs) and strings on tape 3 (representing the outputs) are aligned according to legitimate rewriting sequences as designated by a rule.

8.3.3. Converting MT$_2$ to a transducer

At the end of the construction it is clear that tape 1 is completely unnecessary since it only contains special symbols which we used to filter out illegal sequences during the intermediate steps. We can therefore remove tape 1 from our set of 3-tape configurations. This can easily be done through:[2]

$$MT_2' = \text{range}\Big(MT_2 \circ \big((\Sigma{:}\epsilon)\Sigma\Sigma\big)^*\Big) \tag{8.7}$$

leaving us only with the ‘input’ and ‘output’ tapes. This two-tape representation can obviously be converted to a transducer by the method given in chapter 7 where we traverse the automaton and collapse states, naming odd-numbered transitions input symbols and even-numbered ones output symbols. Before this can be done, however, there are three things that need to be addressed. First, the automaton always begins and ends with the pair $\langle$**@#@**, **@ID@**$\rangle$. Removing the boundary symbols is of course easy: all we do is

$$MT_2'' = \text{range}\Big(MT_2' \circ \big((\textbf{@\#@}{:}\epsilon)(\textbf{@ID@}{:}\epsilon)\Sigma^*(\textbf{@\#@}{:}\epsilon)(\textbf{@ID@}{:}\epsilon)\big)\Big) \tag{8.8}$$

The next concern are the symbols **@ID@** which occurred on tape 3, signaling that the symbol in the same position on tape 2 need to be in an identity relationship. Also, we may have **@0@**-symbols on either tape, representing ϵ. Both of these special symbols can be taken into account in the conversion to a transducer as described in chapter 7—with the addition that **@0@** are converted into ϵ-symbols, and any even position **@ID@** shall be converted to the previous symbol.

We have now reached the point where the construction method has been outlined in its basics while partly disregarding the actual details of how the intermediate multitape configurations and filters are constructed. The illustration thus far has been concerned

[2]In general, many of the operations in this chapter can be defined much more elegantly by the abstract multitape operations in chapter 7. However, so that the contents here should serve as an independent reference for the compilation method on the level of regular expressions, we omit taking advantage of those abstractions.

with compiling a very simple type of rewrite rule. For such rules, the elaborate semantic tape we have constructed is really not necessary—there are a number of simpler ways to construct these transducers. However, the notation on tape 1 turns out to be extremely useful for subsequent variants that we have in mind: constructing parallel replacement rules, and rules that operate with different modalities: left-to-right, right-to-left, longest-match, shortest-match, as well as rules where the triggering context is specified both with respect to the input and output sides.

8.4. Details of basic construction

Let us begin by considering the placement of string pairs $\phi \times \psi$ on tapes 2 and 3, together with the associated semantic symbols on tape 1. The question is how to construct the three-tape configuration such that

- ϕ occurs together with ψ on tapes 2 and 3
- In case one of them in shorter than the other, the other one is padded with **@0@**-symbols to the right
- tape 1 contains the symbols **@I[@**, **@I[]@**, **@I]@**, or **@I@** in the proper sequence as outlined above

This problem can be taken apart as follows. Let us first consider the problem of pairing up ϕ and ψ on two tapes in such a way that the shorter of the two is padded with **@0@**-symbols at the right edge. Clearly, following the basic mechanism outline in chapter 7, we can do so by:

$$CP(\phi, \psi) =$$

$$\text{range}\Big(\phi\mathbf{@0@}^* \circ (\Sigma(\epsilon{:}\Sigma))^*\Big) \;\cap\; \text{range}\Big(\psi\mathbf{@0@}^* \circ ((\epsilon{:}\Sigma)\Sigma)^*\Big) \cap \neg\big(\Sigma^*\mathbf{@0@@0@}\big) \quad (8.9)$$

What we do above is extend ϕ and ψ with an arbitrary number of **@0@**-symbols at the end, interleave the two, and filter out those strings that have two consecutive **@0@** symbols at the end. The latter condition removes any redundant **@0@@0@** pairings at the end of the string.

The second thing that needs to be done is the alignment of the symbols on tape 1. For reasons that will become clear as we move to different modes of replacement, the exact distribution of the symbols on tape 1 is as follows:

1. The first symbol on tape 2 must be aligned with **@I[]@** or **@I[@**
2. The last non-**@0@** symbol on tape 2 must be aligned with **@I[]@** or **@I]@**
3. **@I[]@** can only occur as the very first symbol, and can only be followed by **@I]@**-symbols
4. A **@0@** on tape 2 must be aligned with **@I[]@** or **@I]@**
5. Other symbols are aligned with **@I@**

In other words, the idea of the different **I**-markers is that an opening marker always appears in the first position of a rewrite and a closing marker appears as soon as the last non-zero symbol appears on tape 2. This means several closing markers can appear in succession. However, the symbol **@I[]@**, which is both an opening and closing marker, can only appear once in a rewrite sequence and is always the first symbol when it does appear.

We can encode the above requirements in a three-tape encoding as:

$$\begin{aligned} \mathit{Semtape} = {} & \textbf{@I[]@}\ \textbf{@0@}\ \Sigma\ (\textbf{@I]@}\ \textbf{@0@}\ \Sigma)^*\ \cup \\ & \textbf{@I[]@}\ \backslash\textbf{@0@}\ \Sigma\ \cup \\ & (\textbf{@I[@}\ \backslash\textbf{@0@}\ \Sigma\ \cup\ \textbf{@I]@}\ \textbf{@0@}\ \Sigma)^*\ \textbf{@I]@}\ \Sigma\ \Sigma \end{aligned} \tag{8.10}$$

Now combining the two languages requires us to insert a tape 1 before the 2-tape $CP(\phi, \psi)$ with arbitrary content, and intersecting the two above languages. Hence,

$$RS(\phi, \psi) = \text{range}\Big(CP(\phi, \psi) \circ \big((\epsilon{:}\Sigma)\ \Sigma\ \Sigma\big)^*\Big) \cap Semtape \tag{8.11}$$

For example, *RS*$(abc, defg)$ will produce a single string, the contents of which can be displayed in 3-tape format as:

@**I[**@	@**I**@	@**I]**@	@**I]**@
a	b	c	@**0**@
d	e	f	g

(8.12)

The reason for the stringent requirements on the distribution of the symbols on the semantic tape is that, as we construct additional filters to remove illegal configurations, we can ascertain very quickly, for any symbol on tape 2 or 3, how that symbol participates in a rewrite sequence. For most applications, we need not look ahead, or behind, to know whether a symbol is outside the action of a rewrite rule, is the first symbol in a rewrite rule, or the last, or anywhere in between.

8.4.1. Constructing MT$_1$

As discussed above, MT$_1$ shall contain arbitrary sequences of identity symbols, interspersed with *RS*(ϕ, ψ), and begin and end with the boundary markers.

The Boundary language is quite simply

$$\text{Boundary} = \textbf{@O@ @\#@ @ID@} \tag{8.13}$$

while a single identity symbol in a 3-tape encoding is

$$\text{Identity} = \textbf{@O@}\ \Sigma\ \textbf{@ID@} \tag{8.14}$$

In order to construct the desired MT$_1$ where the above requirements are fulfilled, we construct

$$MT_1 = \text{Boundary } (\text{Identity } \cup \text{ RS}(\phi, \psi))^* \text{ Boundary} \tag{8.15}$$

8.4.2. Filtering out unrewritten sequences

Let us now see how we can filter out from MT$_1$ unwarranted rewrites, and to force rewrites to happen in the correct conditioning environment.

As mentioned, in the above encoding, any occurrence of ϕ in the environment λ _ ρ must be rewritten, unless the rule is designated as optional.

The identifier for an unrewritten ϕ-sequence is of course that a word from ϕ occurs on tape 2 and is aligned with **@O@**-symbols on tape 1 throughout the word. That is, we can define the regular 3-tape language that characterizes all the unrewritten sequences of ϕ.

$$\text{Unrewritten}(\phi) = \text{range}\Big(\phi \circ \big((\epsilon{:}\mathbf{@O@})\ \Sigma\ (\epsilon{:}\Sigma)\big)\Big) \tag{8.16}$$

That is, ϕ on tape 2, aligned with **@O@**-symbols on tape 1, with arbitrary material on tape 3.

This is not exactly the set of sequences we want to rule out, though. What we want to rule out are those configurations where ϕ is preceded by the designated left context and succeeded by the right context. Let us postpone for a minute the possibilities of defining the left and right contexts and assume we can define their 3-tape configurations as well, and call these LeftContext(λ) and RightContext(ρ). Then, what we want to rule out from MT$_1$ are sequences

$$\text{LeftContext}(\lambda)\ \text{Unrewritten}(\phi)\ \text{RightContext}(\rho) \tag{8.17}$$

That is, we can define a language

$$\text{NoUnrewritten} = \neg(\Sigma^*\ \text{LeftContext}(\lambda)\ \text{Unrewritten}(\phi)\ \text{RightContext}(\rho)\ \Sigma^*) \tag{8.18}$$

which we can use as a filter to remove part of the overgeneration in MT_1.

8.4.3. Filtering out improper rewrites

The second part is then to remove those sequences where ϕ has been rewritten, although the context does not warrant as rewrite. We assume again we can define the languages LeftContext and RightContext to identify the conditioning environment. We want to rule out sequences where ϕ occurs on tape 2 rewritten in its entirety, and either the left context or right context are improper. In fact, it is advantageous to talk about the sequences where ϕ is allowed to occur as rewritten, instead of where it is disallowed. In effect we want to define the 3-tape filtering language where:

- The grouping where a sequence ϕ appears on tape 2, aligned with ψ on tape 3, aligned with the beginning and ending symbols on tape 1, only occurs when preceded by LeftContext(λ) and followed by RightContext(ρ)

This, the language whose occurrence we want to constrain, is of course precisely the language we defined earlier, namely

$$\text{RS}(\phi, \psi)$$

In other words, using the logical formalism developed in chapter 6 we want to say that

$$\text{LicensedRewrite} = (\forall x)(x \in \text{RS}(\phi, \psi) \rightarrow S(\text{LeftContext}(\lambda), x, \text{RightContext}(\rho))) \tag{8.19}$$

Another way of saying the same thing is by using the context restriction operation

$$\text{LicensedRewrite} = \text{RS} \;\Rightarrow\; \text{LeftContext}(\lambda)\ \_\ \text{RightContext}(\rho) \tag{8.20}$$

8.4.4. Defining MT_2

As we have now defined both the sufficient and necessary conditions for a rewrite rule to apply, a replacement rule can then be compiled in its entirety as

$$\text{Replace}(\phi, \psi, \lambda, \rho) = \text{MT}_1 \cap \text{LicensedRewrite} \cap \text{NoUnrewritten} \tag{8.21}$$

8.4.5. Optional rules

Now, we may also want to define an 'optional' rule semantics. That is, in an optional rewrite rule, which we shall designate by surrounding the arrow with parentheses, the result we want is that a sequence ϕ may only be rewritten as ψ in the correct environment, but rewriting is not required even though ϕ occurs in such an environment.

For example, a rule such as

$$a \; (\rightarrow) \; \epsilon \; / \; \_ \; b \tag{8.22}$$

would produce two distinct outputs for the input ab: ab and b.

To achieve this in compiling a rule, we simply omit constructing the filter NoUnrewritten and the corresponding intersection in formula (8.21).

8.5. More on conditioning environments

In the preceding discussion we left slightly unspecified exactly what kind of configurations the 3-tape languages LeftContext(λ) and RightContext(ρ) should represent. In the introductory discussion, we assumed that both of these shall refer to symbol sequences in the input which meant that rewrite rules would need to be conditioned with respect to the input string. But this is of course not an absolute requirement: we might as well have them refer to sequences in the output, or a combination of both. Karttunen (1997) contains an instructive example on the different semantics of where the left and right context apply. In particular he outlines four possibilities:

(1) LeftContext(λ) and RightContext(ρ) both refer to the input string

(2) LeftContext(λ) and RightContext(ρ) both refer to the output string

(3) LeftContext(λ) refers to the input while RightContext(ρ) refers to the output

(4) LeftContext(λ) refers to the output while RightContext(ρ) refers to the input

The example given to illustrate the different semantics in Karttunen (1997) is as follows.[3]

$$ab \;\rightarrow\; x \;/\; ab \,\_\, a \tag{8.23}$$

The output for this rewrite rule and the input string $abababa$ is then distinct for each of the four directionalities of the context, namely:

(1)	(2)	(3)	(4)
$abxxa$	$ababxa$ or $abxaba$	$ababxa$	$abxaba$

Defining 3-tape configurations of LeftContext(λ) and RightContext(ρ) in all of the above ways is straightforward. In essence, we want to encode λ and ρ only on tape 2 or tape 3, ignoring the contents of the other tapes. That is, if we place λ on tape 2 when defining the language LeftContext(λ), the left context will apply to the input side, and if placed on tape 3, the output side, and similarly for the right context. Then, if we apply this definition to formulas (8.18) and (8.19), we construct MT$_2$ in such a way that the left and right contexts apply to the input or the output.

[3]Karttunen (1997) uses special symbols instead of the context separator (/) to define these different types of contextual requirements. The corresponding cases and symbols are (1) `||`, (2) `\/`, (3) `\\`, and (4) `//`.

Now, let us declare two functions, Input() and Output(), which place regular languages on either tape 2 or tape 3 according to the above. Constructing both of these is fairly similar to how we have constructed the other 3-tape representations where we declare the contents of one tape and allow arbitrary material on the other tapes. Two things need special treatment in this case, however:

- λ or ρ in a context on tape 2 or 3 may contain $\mathbf{@0@}$-symbols arbitrarily interspersed, which need to be ignored. We need to be able to identify both even with interspersed $\mathbf{@0@}$-symbols.
- Substrings of λ or ρ may be represented on tape 3 as $\mathbf{@ID@}$, in which case we need to, for that symbol, look at tape 2 to match a string on tape 3

Taking this into account, we need to treat the placement of string sets on tape 2 slightly differently than their placement on tape 3, and define

$$\text{Input}(L) = \text{range}\Big(L \circ \big((\epsilon{:}\Sigma)\ \Sigma\ (\epsilon{:}\Sigma)\ \cup\ (\epsilon{:}\Sigma)\ (\epsilon{:}\mathbf{@0@})\ (\epsilon{:}\Sigma)\big)^{*}\Big) \tag{8.24}$$

and

$$\begin{aligned}\text{Output}(L) = \text{range}\Big(&\\ &L \circ \big((\epsilon{:}\Sigma)\ \Sigma\ (\epsilon{:}\mathbf{@ID@})\ \cup\ (\epsilon{:}\Sigma)\ (\epsilon{:}\Sigma)\Sigma \cup\ (\epsilon{:}\Sigma)\ (\epsilon{:}\mathbf{@0@})\ (\epsilon{:}\Sigma)\big)^{*}\\ &\Big)\end{aligned} \tag{8.25}$$

With these functions, we can define LeftContext(λ) and RightContext(ρ) in all of the above combinations of (1)–(4) in the compilation formulas (8.18) and (8.20) as:

(1) LeftContext(λ) = Input(λ), RightContext(ρ) = Input(ρ)

(2) LeftContext(λ) = Output(λ), RightContext(ρ) = Output(ρ)

(3) LeftContext(λ) = Input(λ), RightContext(ρ) = Output(ρ)

(4) LeftContext(λ) = Output(λ), RightContext(ρ) = Input(ρ)

8.6. Multi-level conditioning environments

In defining the conditioning environment for rewrite rules above we a priori decided, following Karttunen (1997), that λ and ρ apply either to the input side or the output side. But because of the way the 3-tape representation is set up in MT$_1$ there is really no need to settle for one or the other. There is nothing preventing us from defining an environment that constrains both the input side and output side separately. In effect, we are able to compile a rewrite rule of the format:

$$\phi \rightarrow \psi \;/\; \lambda_{in} \times \lambda_{out} \_\; \rho_{in} \times \rho_{out} \tag{8.26}$$

where λ_{in}, λ_{out}, ρ_{in}, and ρ_{out} are all arbitrary regular languages. In that case we are left with only one kind of conditioning environment specification. The above 'directional' rules (1)–(4) would all be special cases of this one type of rule—cases where either the input side or the output side is Σ^*. Namely,

(1) $\phi \rightarrow \psi \;/\; \lambda \times \Sigma^* \_\; \rho \times \Sigma^*$

(2) $\phi \rightarrow \psi \;/\; \Sigma^* \times \lambda \_\; \Sigma^* \times \rho$

(3) $\phi \rightarrow \psi \;/\; \lambda \times \Sigma^* \_\; \Sigma^* \times \rho$

(4) $\phi \rightarrow \psi \;/\; \Sigma^* \times \lambda \_\; \rho \times \Sigma^*$

The way to actually compile these formulas is fairly obvious. We declare

$$\text{LeftContext}(\lambda_{in}, \lambda_{out}) = \text{Input}(\lambda_{in}) \cap \text{Output}(\lambda_{out}) \tag{8.27}$$

$$\text{RightContext}(\rho_{in}, \rho_{out}) = \text{Input}(\rho_{in}) \cap \text{Output}(\rho_{out}) \tag{8.28}$$

and use these definitions whenever we refer to the context 3-tape languages in (8.18) and (8.20).

8.7. Additional conditioning environments

We need not restrict ourselves to talking about left and right contexts that legitimize a rewrite rule only in terms of strings or languages. We can additionally constrain the conditioning environment depending on semantic issues. We may, among other things, want to have a rewrite rule apply only in the context where the previous symbol immediately to the left or the following one to the right is not affected by a rule. For example, consider a rule:

$$a \rightarrow b \,/\, a \;\times\; \Sigma^* \,\_ \tag{8.29}$$

in effect stating that the symbol a must be rewritten as b if preceded by an a on the left input side, and anything on the output side.

With such a rule an input aaa would be rewritten as abb. However, if we added the above semantics, that the rule apply only in environments where symbols to the left and right are unrewritten, the rule would yield aba. This is easy to see because of the two possible alignments

a	a	a
a	b	b

and

a	a	a
a	b	a

the former would not be legitimate because the middle and rightmost as would be rewritten although they are not completely flanked by nonrewritten symbols.

8.7.1. Epenthesis rules

This type of an additional constraint is useful when dealing with epenthesis rules in phonology and morphology. A rule such as:

$$\epsilon \rightarrow a \;/\; c \;\times\; \Sigma^* \; \_ \; d \;\times\; \Sigma^* \tag{8.30}$$

intended to insert an a between c and d (on the input side) would certainly do so, and given a input string cd yield cad. It would, however, also yield $caad$, $caaad$, etc. etc. Indeed, the conditioning environment is such that an infinite number of legitimate input/output pairings exists for the input cd. If we now added the constraint that, for a rule to be legitimate, an unrewritten symbol must occur immediately to the left and right, we would get only cad as an output, as intended for the above rule.[4]

Including such an extension is not complicated as we have intentionally marked all symbols unrewritten by a rule with an @**O**@-symbol on tape 1. Hence, this semantics can be added into the contexts by intersecting the language defining the other contextual restrictions with the language where the first symbol is @**O**@ (to the right) and the last @**O**@ (to the left) of a rewrite. That is, what we denote

$$\phi \overset{ep}{\rightarrow} \psi \tag{8.31}$$

as an additional constraint on the contents of the left and right contexts, and compile it by defining

$$\text{LeftContext}_{ep} = \text{LeftContext} \;\cap\; (\Sigma\;\Sigma\;\Sigma)^*(@\mathbf{O}@\;\Sigma\;\Sigma) \tag{8.32}$$

$$\text{RightContext}_{ep} = \text{RightContext} \;\cap\; (@\mathbf{O}@\;\Sigma\;\Sigma)(\Sigma\;\Sigma\;\Sigma)^* \tag{8.33}$$

[4] In Beesley and Karttunen (2003) a similar type of logic is defined—precisely for treatment of epenthesis rules—called [..]-rules. The idea there is that an input string is first considered to contain exactly one ϵ symbol between each input symbol, i.e. cd is interpreted as $\epsilon c \epsilon d \epsilon$, after which an [..]-rule only rewrites the ϵ-symbols.

which we then use in lieu of the normal LeftContext and RightContext when compiling a rule.

This is of course not the only type of additional constraint we may add: because we have access to fairly detailed semantic information on tape 1, the conditioning environment can be augmented with many different types of semantic content.

8.7.2. Word boundaries

Naturally, we will want to be able to refer to the edge of a word in a conditioning environment. For example, we will want to compile rules like:

$$L \rightarrow \epsilon \, / \# \_ \tag{8.34}$$

in effect deleting any members of the regular language L at the left edge of an input string. This is the reason we added the word boundary symbols **@#@** to the beginning and end of every configuration accepted by MT_1. Doing so allows us to simply convert #-symbols (or whatever special symbol we designate boundaries with) to the symbol **@#@** when compiling replacement rules and proceeding as usual.

Attempting to capture rules that are conditioned by word boundaries by somehow modifying the compilation depending on the presence of #-symbols results in great difficulties, and the method presented here allows us to completely disregard this in the compilation process (this idea is based on the approach used in Kaplan and Kay (1994) for handling boundaries).

8.8. Multiple conditioning environments

A convenient additional notation would be the ability to specify multiple conditioning environments for a rule, such as:

$$\phi \rightarrow \psi \, / \, \lambda_1 \_ \rho_1 \, , \; \dots \, , \; \lambda_n \_ \rho_n \tag{8.35}$$

In such a case, we could like the interpretation to be disjunctive: the rule may apply in any of the contexts listed, and ϕ cannot be unrewritten in any of the contexts listed. For example, a rule:

$$a \;\rightarrow\; b \,/\, \lambda \,\_\; , \;\_\; \rho \tag{8.36}$$

would signify that we rewrite a as b whenever it occurs following λ or preceding ρ, and that a must be rewritten as b in those environments.

In such a case, we must modify formulas (8.18) and (8.20). To handle the first case, we declare:

$$\begin{aligned}\text{NoUnrewritten} \;&=\; \neg(\Sigma^* \,\text{LeftContext}(\lambda_1)\; \text{Unrewritten}(\phi)\; \text{RightContext}(\rho_1)\; \Sigma^*) \cap \ldots \cap \\ &\quad \neg(\Sigma^* \,\text{LeftContext}(\lambda_n)\; \text{Unrewritten}(\phi)\; \text{RightContext}(\rho_n)\; \Sigma^*)\end{aligned} \tag{8.37}$$

Similarly, (8.20) must be modified as

$$\begin{aligned}\text{RS} \;\Rightarrow\; &\text{LeftContext}(\lambda_1)\,\_\,\text{RightContext}(\rho_1)\,, \ldots\,, \\ &\quad \text{LeftContext}(\lambda_n)\,\_\,\text{RightContext}(\rho_n)\end{aligned} \tag{8.38}$$

8.9. Multiple rule application

Until now we have only examined the compilation of a single rule into a finite-state transducer. Multiple rules, where the idea is that the output of one rule serves as the input of the next rule, is of course easily handled by compiling individual rules into transducers and composing them.

However, there is nothing to prevent us from declaring multiple rules that apply in parallel. In fact, there is a natural semantics for parallel rule application if we declare, for a set of rules $\mathcal{R}$, that for each rule in $\mathcal{R}$, the following shall hold:

(a) for each obligatory rule $\mathcal{R}_i$ in $\mathcal{R}$, there shall be no unrewritten sequences ϕ_i in the proper conditioning environment

(b) Each ϕ_i may only apply in their respective conditioning environments

Note that the way we have specified item (a) means indirectly that two rewrite rules cannot be in conflict even if they apply to the same string. Suppose we have two rewrite rules that both need to write the same string s differently, and whose conditioning environments overlap, say:

$$x \;\rightarrow\; y \,/\, a \,\_\;, x \;\rightarrow\; z \,/\, a \,\_ \tag{8.39}$$

Now, an x occurring after an a—as in the string ax—obviously applies to both rules: the leftmost one needing to rewrite that x as y, and the rightmost one as z. However, both outputs become possible in this scenario, since in defining parallel rules, point (a) only says that a may not remain *unrewritten*. It crucially does not say a must be rewritten as one of the rules dictate, say y, or z, in which case the rules would conflict with each other. But the way we have defined the interaction is that as long as that a does not remain unrewritten, from the 'point of view' of either rule, the configuration is acceptable. The end result is that if two rules apply to a substring, several outputs become a possibility.

To handle this kind of interaction in the compilation, we must first modify the construction of MT$_1$ in formula (8.15) so that the rewrite sequences which occur interspersed with identities in MT$_1$ reflect all the possible rules.

$$MT_1 = \text{Boundary (Identity } \cup \text{ RS}(\phi_1, \psi_1) \cup \ldots \cup \text{RS}(\phi_n, \psi_n))^* \text{ Boundary} \tag{8.40}$$

As for the other parts—the constraints NoUnrewritten and LicensedRewrite—we compile these for each rule in $\mathcal{R}$ exactly as in formulas (8.37) and (8.38) and intersect these languages, i.e.

$$\text{NoUnrewritten} = \text{NoUnrewritten}_1 \cap \ldots \cap \text{NoUnrewritten}_n \tag{8.41}$$

and

$$\text{LicensedRewrite} = \text{LicensedRewrite}_1 \cap \ldots \cap \text{LicensedRewrite}_n \tag{8.42}$$

8.10. Modes of rule application

As has been motivated elsewhere, we would also like to be able to assert constraints over the mode of rewriting if a string in ϕ occurs in such a position that there are several ways in which one can legitimately rewrite such a sequence. The type of modality we have motivated from a linguistic point of view is a leftmost-longest and leftmost-shortest type of matching. Of course the symmetric versions of rightmost-longest and rightmost-shortest are also definable. In contrast to previous work, we shall split the two types of requirements into separate constraints of leftmost/rightmost and longest-match/shortest-match.

8.10.1. Leftmost

A leftmost mode rule application becomes a possibility when a string ϕ contains substrings that may be a prefix or suffix of ϕ itself. Consider a very simple type of rule

$$aa \rightarrow x \tag{8.43}$$

This rule would normally provide two possible outputs for the input string aaa: ax and xa. Obviously the string aaa has two overlapping occurrences of ϕ, $[aa]a$ and $a[aa]$, and so both rewrites are correct. However, a rule with the added leftmost semantics

$$aa \overset{leftmost}{\rightarrow} x \tag{8.44}$$

would only accept only the relation $aaa \rightarrow xa$. Again, relying on the information contained on tape 1, we can create an additional filtering mechanism on top of the standard one to rule out rewrites that do not follow this semantics. What we want to do is remove an otherwise legal configuration such as:

@O@	**@I[@**	**@I]@**
a	a	a
@ID@	x	**@0@**

(8.45)

since in such configurations, we find that there is an instance of ϕ that begins outside the action of a rewrite rule to the left. That is, an instance of ϕ is aligned with **@O@**, in the correct conditioning environment.

These configurations are characterized by being flanked to the left by the left context, to the right by the right context, and contain ϕ aligned with **@O@** as the first character. That is:

$$\text{NonLeftmost}(\phi) = \text{Input}(\phi) \ \cap \ (\textbf{@O@}\Sigma^*) \tag{8.46}$$

and hence, we want to rule out situations where we find

$$\text{LeftContext}(\lambda)\ \text{NonLeftmost}(\phi)\ \text{RightContext}(\rho) \tag{8.47}$$

This can be achieved by adding to the set of filters that apply to MT_1 the language

$$\neg(\Sigma^*\text{LeftContext}(\lambda)\ \text{NonLeftmost}(\phi)\ \text{RightContext}(\rho)\ \Sigma^*) \tag{8.48}$$

This is the same language as in (8.18) with the exception that the language Unrewritten has been substituted for NonLeftMost(ϕ). We can thus incorporate this into (8.18) directly, by adding to a leftmost-oriented rule the statement:

$$\text{NoUnrewritten} =$$
$$\neg(\Sigma^*\ \text{LeftContext}(\lambda)\ (\text{Unrewritten}(\phi) \cup \text{NonLeftmost}(\phi))\ \text{RightContext}(\rho)\ \Sigma^*) \tag{8.49}$$

Interestingly, adding a leftmost semantics to rewrite rules has ramifications as regards the interaction with optionality and other modalities. For example, we have tacitly assumed that an 'optional' rewrite rule $\phi(\rightarrow)\psi$ simply rewrites ϕ in the correct environment, but does not have to do so. Now, adding a leftmost constraint raises questions about how the two rule types should work together. For example, suppose we have a rule

$$aa \; (\overset{leftmost}{\rightarrow}) \; x \tag{8.50}$$

How should this rule behave with respect to the input aaa? Clearly, if the rewriting is 'optional' aaa is a legitimate output, as is xa. But what about the output ax, where we have *not* rewritten the leftmost instance of aa? Whether this is a valid interpretation depends on how we define 'optionality': is optionality something that only applies to rewriting vs. not rewriting, or does optionality also apply to the 'leftmost' criterion? In other words, is following a leftmost strategy also 'optional'? We shall in what follows remain slightly agnostic as to the semantics of optionality since cases they affect would arguably not surface very often in natural language applications. However, it can be easily seen that both types of optionality—optionality of rewriting, and optionality of leftmostness—could be encoded in filters on top of MT$_1$, if we wanted to choose a specific semantics for the interaction of these modalities.

8.10.2. Longest-match

Let us now consider how to add longest-match type semantics to a rewrite rule. For a rule where a rewriting of ϕ could apply in various ways depending on how much material we match in chunks, multiple outputs are possible, and we want to filter out the non-longest candidates for each possibility. For example, the normal rewrite rule:

$$a^{+} \; \rightarrow \; x \tag{8.51}$$

would apply to the string aa in two ways, either outputting xx or x. What we want to introduce is a semantics such that a rule

$$a^{+} \overset{longest}{\rightarrow} x \qquad (8.52)$$

only rewrites aa as x, since that is the maximal extent to which the rule can apply.

In other words, we want to rule out configurations in MT$_1$ such as

@I[]@	**@I[]@**
a	a
x	x

(8.53)

In the above, what we have is two independent rewrites of the symbol a (which is a member of ϕ). However, we *could* have chosen the longer member of ϕ, aa as our substring to be rewritten and performed just one rewrite operation.

The way to identify non-longest rewrites is then naturally to look at tape 2 and tape 1: a non-longest rewrite of ϕ on tape 2 begins with an opening bracket on tape 1, but extends at least one symbol across an instance of a closing bracket. In the above case, we find aa, a instance of ϕ which begins with **@I[]@** (which is simultaneously an opening and closing bracket), and extends one step to the right beyond a closing bracket.

We then define a non-longest rewrite in almost the same way as a nonleftmost one:

$$\text{NonLongest}(\phi) = \text{Input}(\phi) \cap ((\text{IOPEN}\ \Sigma^{2})\ (\Sigma^{3})^{*}\ (\textbf{@O@}\ \cup\ \text{IOPEN})\Sigma^{*}) \qquad (8.54)$$

where IOPEN = $($**@I[@** $\cup$ **@I[]@**$)$.

This can then be incorporated into NoUnrewritten exactly as in the leftmost case: of course we need to make sure that the filter only applies when the left and right contexts are correct, analogously to (8.49).

Naturally, we will often want to include both types of semantics in one rule, such that the directionality of a rule follows both the leftmost and longest-match semantics. For example, a rule

$$aba \cup ab \cup ba \xrightarrow{X} x \tag{8.55}$$

for an input string aba would produce the following outputs, depending on the semantics chosen.

- X = no semantics: ax,xa,x
- X = leftmost only: xa, x
- X = longest-match only: ax, x
- X = both leftmost and longest-match: x

For most applications we assume that both a leftmost and a longest-match semantics would be needed.

8.10.3. Shortest-match

Shortest match is defined symmetrically to longest match, although its definition is slightly more complicated. Consider again the rule $a^+ \to x$ and a possible (non-shortest-match) configuration:

@I[@	**@I]@**
a	a
x	**@0@**

(8.56)

Here, we see that to identify the offending non-shortest match sequences, we need to find a complete instance of ϕ on tape 2, the beginning of which is aligned with an opening bracket on tape 1, but that never contains a closing bracket subsequently. In the above example, the first a symbol occurs in such exactly such a configuration.

$$\text{NonShortest}(\phi) = \text{Input}(\phi) \cap (\textbf{@I[@}(\Sigma - (\textbf{@I]@} \ \cup \ \textbf{@I[]@}))^*) \tag{8.57}$$

Again, we can add such a constraint, possibly in conjunction with the leftmost match, to the set of filters that apply to MT$_1$.

8.11. Markup rules

An interesting and useful rule device introduced in Beesley and Karttunen (2003) is a so-called markup rule. The idea is that we allow rules of the format

$$\phi \rightarrow \psi^l \ \ldots \ \psi^r \ / \ \lambda \ \_ \ \rho \tag{8.58}$$

that denote that a string from ϕ in the proper conditioning environment shall remain untouched, but ψ^l shall be inserted to the left of ϕ and ψ^r to the right of ϕ. This type of rule has found use in phrase marking applications where we can imagine a rule such as

$$P \rightarrow [_P \ \ldots \ ]_P \tag{8.59}$$

where P is an abstract symbol—a regular language—denoting a sequence of some sort, a syllable or a noun phrase, for instance. With such a rule, possibly augmented with leftmost or longest-match semantics, one could then 'mark' such sequences with $[_P \ P \ ]_P$.

In order to accommodate this type of a rule we need to include the possibility of such a configuration in the definition of MT$_1$. The core of the modification has to do with the function $RS(\phi, \psi)$ defined in (8.11), to which we need to provide an alternative $RS(\phi, \psi^l, \psi^r)$ to insert the proper material in rewrite sequences of MT$_1$. We need, then, to create rewrite sequences in MT$_1$ that look like:

@I[@	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	**@I]@**
@0@	$\ldots$	**@0@**	ϕ_1	$\ldots$	ϕ_n	**@0@**	$\ldots$	**@0@**
ψ^l_1	$\ldots$	ψ^l_n	**@ID@**	$\ldots$	**@ID@**	ψ^r_1	$\ldots$	ψ^r_n

(8.60)

In other words, a rewrite sequence $RS(\phi, \psi^l, \psi^r)$ produces 3-tape configurations where the segments from ϕ appear in the middle of tape 2, aligned with **@ID@**-markers, as ϕ itself shall pass through untouched. Also, to the left and right of ϕ appear sequences with **@0@** throughout on tape 2 aligned with ψ^l (to the left) and ψ^r (to the right).

To this end, we define

$$RS(\phi, \psi^l, \psi^r) =$$
$$\text{range}((CP(\epsilon, \psi^l)\ CP(\phi, \textbf{@ID@}^*)\ CP(\epsilon, \psi^r)) \circ ((\epsilon{:}\Sigma)\ \Sigma\ \Sigma)^*) \cap Semtape \quad (8.61)$$

This of course needs to be incorporated in place of $RS(\phi, \psi)$ whenever a markup rule is included in MT_1. Since the contents of the tapes with such a modified RS is otherwise indistinguishable from that of any other rewrite rule, the contextual constraints or rule modalities can be included exactly as in the ordinary replacement rule cases.

A notational variant or generalization of this idea is found in Gerdmann and van Noord (1999) who calls the notation 'rewrite rules with backreferences.' In that work, the idea is that we describe a set of separate transduction functions for ϕ which modifies it in the way we desire. This function, $T(\phi)$, may be quite arbitrary and may function as a 'markup rule' or an ordinary transduction (a cross product), etc. In such a case we define rewrite rules in general as

$$\phi \rightarrow T(\phi)\ /\ \lambda\ \_\ \rho \quad (8.62)$$

where $T(\phi) = \psi$ is a special case of our ordinary rewrite rule. Now, this can be included in our compilation method by redefining the $RS()$-function, in the manner

$$RS(\phi, T(\phi)) = \text{range}(CP(\phi, T(\phi)) \circ ((\epsilon{:}\Sigma)\ \Sigma\ \Sigma)^*) \cap Semtape \quad (8.63)$$

and leaving it to be defined elsewhere what $T(\phi)$ does.

The reason we have defined the markup rule separately instead of relying on the above abstraction is that doing so allows us to force the alignment of the inserted material with zeroes. If the markup relied on an external function $T(\phi)$, then a markup rule such as

$$a \rightarrow\ [\ \ldots\ ] \quad (8.64)$$

could provide alignments such as

@I[]@	**@I]@**	**@I]@**
a	**@0@**	**@0@**
[	a	]

(8.65)

instead of the arguably more natural

@I[@	**@I]@**	**@I]@**
@0@	a	**@0@**
[	**@ID@**	]

(8.66)

In the latter case, we preserve the identity mapping of the elements of the string ϕ, and will more likely produce smaller, more efficient transducers with that alignment.

8.12. Summary of rewrite rule compilation

Let us now briefly summarize the compilation procedures described. In its most general form, we can compile rules of the type,

$$\phi^1 \overset{op^1}{\rightarrow} \psi^1 \ / \ \lambda^1_1 \text{-} \ \rho^1_1 \ , \ldots , \lambda^1_m \text{-} \ \rho^1_m$$
$$\vdots$$
$$\phi^n \overset{op^n}{\rightarrow} \psi^n \ / \ \lambda^n_1 \text{-} \ \rho^n_1 \ , \ldots , \lambda^n_m \text{-} \ \rho^n_m \qquad (8.67)$$

where each ϕ and ψ are arbitrary regular languages. Also, each of the contexts containing λ and ρ can be divided into an input side and output side: $\lambda^i_j = L_{in} \times L_{out}$, where L_{in} and L_{out} are arbitrary regular languages.

The different modalities available for op are:

- optional rewriting $(\rightarrow)$
- leftmost rewriting $\overset{leftmost}{\rightarrow}$
- longest-match rewriting $\overset{longest}{\rightarrow}$
- shortest-match rewriting $\overset{shortest}{\rightarrow}$

- epenthesis rewriting $\overset{ep}{\rightarrow}$

which can be combined like boolean operators. For instance, one can have a rewrite rule operating with a combined operation *leftmost/longest*.

In addition, we presented a compilation method for the markup rule, denoted for the rule part

$$\phi \overset{op}{\rightarrow} \psi^l \ \ldots \ \psi^r \tag{8.68}$$

which can of course be included in a set of multiple rules as in (8.67). For such a rule, ϕ is left untouched, but ψ^l and ψ^r are epenthetically inserted to the left and right of instances of ϕ.

8.13. Comparison to previous work

Large subsets of the types of rewrite rules that are included in this chapter have been inspired either by the work of Kaplan and Kay (1994) or Beesley and Karttunen (2003). In particular Beesley and Karttunen provide many linguistic arguments why different modalities and operator types are needed. This has been the starting point of the work in the chapter: to provide a detailed compilation method for linguistically motivated rewrite rules.

Most of the rule compilation semantics here have counterparts in the above publications. Table 8.1 shows how some of the semantics and terminology developed here coincide with other work. Some minor details have been omitted—for example, the Xerox rewrite rules define a left-arrow rule $a \leftarrow b$, which can be interpreted as $(a \rightarrow b)^{-1}$, the inverse of a right arrow transducer, and so is really a notational variant, as well as a double-arrow-rule $a \leftrightarrow b$, which is an intersection of the two rule semantics.

The multitape approach for compiling rules chosen here seems to provide more flexibility in defining rule types and in adding operations not found elsewhere. The multi-context compilation, for instance, seems not to be possible at all with the type of compilation approach in Kaplan and Kay (1994) and Kempe and Karttunen (1996). In such methods,

	optional rules	epenthesis rules	leftmost/ longest	leftmost/ shortest	markup rules	parallel rules	multiple contexts	input/ output contexts	two-level contexts
K&K	‘optional’	≈	≈	N/A	N/A	≈ batch rules	≈	≈	N/A
Xerox	(->)	≈ [..]	@->	@>	. . .	,,	,	|| \\/ \\\\ //	N/A

TABLE 8.1. *Comparison of notation and features for different types of rewrite compilation formulas.*

the core idea is to freely insert, by composition, a set of markers $<$ and $>$, then rewriting $< \phi > \rightarrow < \psi >$ and at some point in the process constraining the markers to only occur in positions where the left and right context are correct. Now, one can constrain $<$ and $>$ (which denote the end of the left context or the beginning of the right context) either before or after rewriting ϕ, yielding that whatever constraints one has on the left or right applies either to the input or the output. Performing both does not seem possible unless one has access to both string sets: that of pre-rewriting and post-rewriting. That, however, is not the case in either Kaplan and Kay (1994) and Kempe and Karttunen (1996), as they destroy the input when doing this rewrite step. Consider the following chain of intermediate representations in this approach for a rewrite rule $ab \rightarrow x$:

(1) abababa

(2) ab<ab><ab>a

(3) ab<x><x>a

(4) abxxa

Here (1) represents the input. After step (2) we have inserted brackets, and after step (3) contents within the brackets have been replaced, and after (4) the brackets have been removed. Now, one can constrain the occurrence of the left and right bracket either before the rewriting step (3), or after it. But if one does so after (3), one loses access to the original strings that existed prior to rewriting.

Similar problems appear to arise in the definition of longest-match and shortest-match rules with contextual requirements in the model of Kaplan and Kay (1994) and Kempe and Karttunen (1996). Karttunen (1996) provides a compilation method for longest-match formulas, but without contextual requirements, whereas the method in this chapter allows for arbitrary contexts as well.[5]

In contrast to the large amount of literature on the topic of compiling rewrite rules into transducers, there is a great paucity of actual implementations—commercial or non-commercial—that one can test, or use in the development of large scale grammars.[6] Given the extremely complex nature of compiling rewrite rules into transducers, one would assume that implementations exist—if for nothing else, at least for testing purposes, since without an actual implementation, testing the correctness of formulas of this magnitude of complexity appears impossible.[7]

8.13.1. Efficiency

One might be concerned that all the operations being done with conversions to and from multitape automata would seriously degrade the efficiency of the algorithm. This does not seem to happen, though. In comparing timing results against other rewrite rule compilation algorithms, we found that the algorithm here was in many cases by far the fastest in compiling various types of rules with variable-length contexts. As a point of comparison, we may look at the *xfst*-toolkit algorithm for compiling rules of the format ($k \in [100, 1000]$):

[5]The Xerox tools include the possibility of constraining longest-match rules with contextual requirements, but the method remains undocumented.

[6]Works that specify methods for compiling rules into transducers include Ritchie et al. (1991); Kaplan and Kay (1994); Grimley-Evans et al. (1996); Kempe and Karttunen (1996); Mohri and Sproat (1996); Laporte (1997); Kiraz (1997); Gerdmann and van Noord (1999); Skut et al. (2003); Vaillette (2004); Yli-Jyrä (2007, 2008). However, only Kempe and Karttunen (1996) and Gerdmann and van Noord (1999) appear to provide an implementation.

[7]Ron Kaplan (p.c.) reports that commercial finite-state transducer systems based on string rewriting were built and tested extensively, and the resulting transducers subsequently distributed commercially well before publication of Kaplan and Kay (1994).

$$a \rightarrow b \;/\; c^k \;\_ \tag{8.69}$$

$$a \rightarrow b \;/\; \_ \; c^k \tag{8.70}$$

given in figures 8.3 and 8.4. The rules chosen for these timing experiments may seem artificial, but the choice was made because such results have been reported in other publications dealing with rewrite rule compilation (Mohri and Sproat, 1996; Vaillette, 2004). A faithful implementation of Kempe and Karttunen (1996) (which *xfst* is based on) was available to compare against, but unfortunately there appears to be a serious discrepancy between that and the authors' reference implementation, *xfst*. The way the paper had described the compilation procedure caused rewrite rules with long right contexts to compile very slowly (not finishing in a reasonable time for right contexts of length greater than 20 symbols). Comparing against this would obviously have been unfair, since the problem has been remedied in the *xfst* software available from Xerox, which was therefore chosen as a primary comparison. Since the workings of *xfst* remain undocumented, it is not possible to reimplement the formulas in their fixed versions, and the best one can do is to compare the implementation here against *xfst*. Naturally, the rewrite algorithm is only a component in a large collection of algorithms, and so other parts (such as efficiency of determinization and minimization) have a bearing on the overall result. Hence, the overall timing result is a likely a combination of the efficiency of the fundamental algorithms and the rewrite rule compilation algorithm. Nevertheless, the timing results do indicate (at the least) that the current algorithm, compiled by a detour through a multitape automaton encoding, is viable in practice.

Other algorithms and timing results given in the literature such as Mohri and Sproat (1996); Vaillette (2004)—even taking into account hardware differences—do not come even remotely close in efficiency to that of *xfst* and the current implementation, and so were not taken into consideration in the comparison.[8]

[8]To be fair, the intention in Vaillette (2004) is not to provide a fast algorithm, but to provide one that is

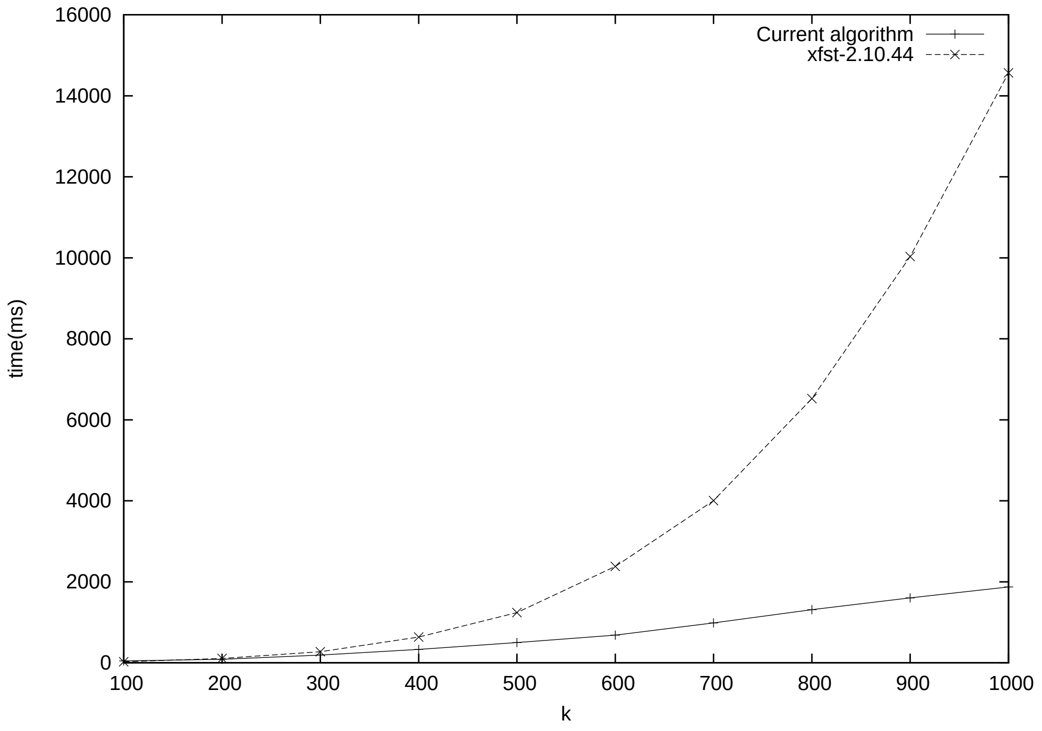


FIGURE 8.3. *Timing results for rules of the form* `a -> b || c`^{k} `_`.

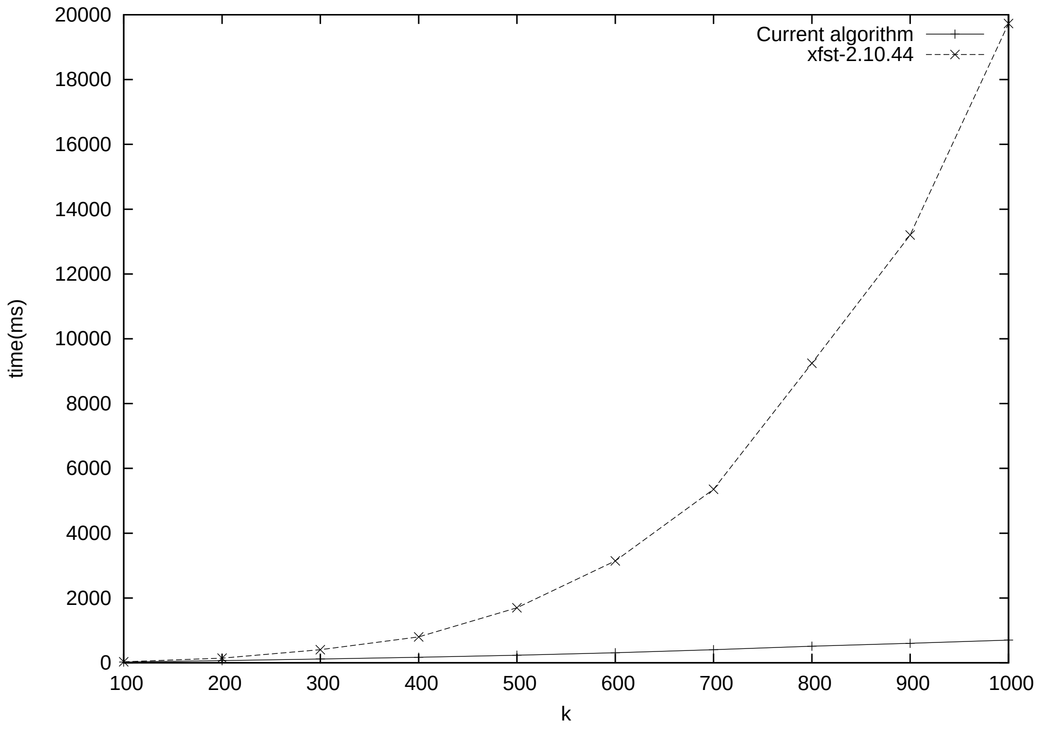


FIGURE 8.4. *Timing results for rules of the form* `a -> b || _ c`^{k}.

8.14. Discussion

This chapter has presented a method for compiling such rewrite rules as are motivated in phonology and morphology into finite-state transducers and described the logic of the general approach, along with some examples. We found that using a multitape automaton as an intermediate step in the compilation process seems to provide the power one needs to compile the major types of rewrite rules needed quite efficiently. Also, the encoding immediately offers some new perspectives and possibilities in defining additional types of rules.

As a particularly interesting extension, we have singled out the possibility of compiling rewrite rules that specify contexts on two levels separately, both the input side and output. It is very likely that one could, by such a rule formalism, using multiple parallel rules of the format

$$\begin{gathered}
\phi^1 \rightarrow \psi^1 \;/\; \lambda^1_{in} \times \lambda^1_{out}\,\_\; \rho^1_{in} \times \rho^1_{out} \\
\ldots \\
\phi^n \rightarrow \psi^n \;/\; \lambda^n_{in} \times \lambda^n_{out}\,\_\; \rho^n_{in} \times \rho^n_{out}
\end{gathered} \tag{8.71}$$

define an entire phonological or morphological grammar. Naturally, we would assume that the other modalities would be available, such as an optional rule ($\rightarrow$) or any of the directional variants we have defined. Such an approach would in a way be reminiscent of a two-level grammar, since we would only have one set of parallel rules, and hence, only two levels. The difference lies in that the two-level formalism (Koskenniemi, 1983) is quite strict in that one cannot define multi-segment correspondences—recall that of the

clear and verifiably correct, and so the timing results for the equivalent of figure 8.3 where $k = 6$ is over 1000 ms, whereas the same timing result for both the current algorithm and *xfst* is significantly less than one millisecond.

four possible rule types, the left-hand-side of a rule in a two-level grammar is always a symbol pair: $a{:}b$. With the current formalism, we can in fact describe pairings of arbitrary languages where the legitimate contexts are also pairs of arbitrary languages. Whether such a grammar formalism is useful is an empirical question; but in any case there is nothing that prevents us from writing such rules: a batch of them can be compiled into finite-state transducers exactly as a cascade of single rules or a set of parallel rules where the context always refers to either the input or the output, but not both, as in the Karttunen (1997) formalism.

An interesting observation is that this bi-context rewrite rule formalism seems to reach the boundaries of the types of alternations that are possible to describe with finite-state transducers. For example, one could easily be led to think that, since it is possible to compile rules where the left and right contexts are specified separately on the input and output side, or where the left and right context is specified only on the input and output side, one could also specify a context such as

$$\phi \rightarrow \psi \; / \; Id(\lambda)\_ \tag{8.72}$$

where $Id(\lambda)$ dictates that the left context is a string from λ in an identity relation. That is, the left context has to be the same string on the input side and output side, drawn from the language λ. But such a rule cannot be compiled to a transducer in the general case. In fact, if we could compile such a rule into a finite-state transducer, we could also solve the Post Correspondence Problem as a simple argument shows (see chapter 5).[9] The end result is that we can indeed compile rules that have separate, even quite complex, contextual requirements for the input and output sides separately, but we cannot compile a rule where

[9]We create a set of rules that (i) translates strings from the upper side of the PCP tiles to their lower side correspondents. Now, we add an extra rule (ii) $\epsilon \rightarrow x \; / \; Id(\Sigma^+) \; \_$. That is, we insert an x after identity relations. Now, we take the language that does not contain x and compose it with the rule transducers (i) and (ii) and examine the range of this transducer. If the range possibly contains x, the PCP has a solution. Hence, if such a rule set were compilable, we would have an effective procedure for solving the PCP.

we require that the input and output side of some context be the same string, even for very simple languages.

Linguistically, this ability to define a grammar through only parallel rules that are not (contra two-level grammars) constrained to consist of single-symbol pairings may be of interest. There has been a general trend in the more theoretical phonological literature to consider ‘parallel’ and ‘serial’ grammars fundamentally distinct approaches to describing phonological processes.[10] This has been true in generative models where serial models of phonology have sometimes been rejected in favor of parallel ones (the ‘unordered rule hypothesis’) and vice versa. That we can compile parallel multi-level rules into transducers without restrictions of any sort on the arguments of the rules, and likewise compose rewrite rules serially into the same kinds of transducers, argues for the likelihood that ‘parallel’ and ‘serial’ models, are at least in models of phonological alternation, of equal descriptive power. And since it has been found empirically that the finite-state transducer model is (at least) sufficient for modeling phonological processes, the choice of ‘serial’ vs. ‘parallel’ is largely one of taste.

[10]For example, Kenstowicz and Kisseberth (1979) devote much space to arguing that phonology must be ‘serial’ based on the predictions from the opposite possibility they call ‘the direct mapping hypothesis’. Similar argumentation can be found throughout the phonological literature. More recent work on Optimality Theory almost invariably bring up the notion of ‘opacity’—something that stems from interaction in ordered rules—to argue for or against some model of phonology, either serial or parallel (Kager, 1999).

9. MORPHOLOGICAL GRAMMARS AND MULTITAPE AUTOMATA

9.1. Introduction

In this chapter we shall be concerned with practical grammar development using multitape automata. To this end, we use the model developed in chapter 7 and the associated operations on multitape automata to explore the development of complete morphological grammars using this formalism.

The primary reason for our suggestion of using k-tape automata directly to construct language models instead of the more familiar (2-level) finite-state transducer model is the lack of sufficient information alignment in 2-tape models. As has been noted already, the paradigm task of morphological analysis is to map a word-form represented as a string to its analysis, also represented as a string. The string representing the analysis may of course be of arbitrary complexity—analyses consisting of hundreds of symbols are not unheard of in actual applications. This linear string-to-string mapping has one shortcoming, however: grammatical information encoded in a finite-state transducer analysis provides little or no alignment between components in the word-form (surface form) and the analysis.

In figure 9.1 we see a Finnish word and its analysis; going from one type of string to the other could easily be encoded in a finite-state grammar and the encoding here is fairly standard. In the figure we have also marked some information alignment completely absent from the string-to-string transduction. That is, some of the segments in the string clearly

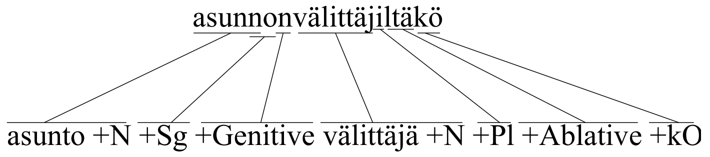


FIGURE 9.1. *Alignment of information in a word and its parse.*

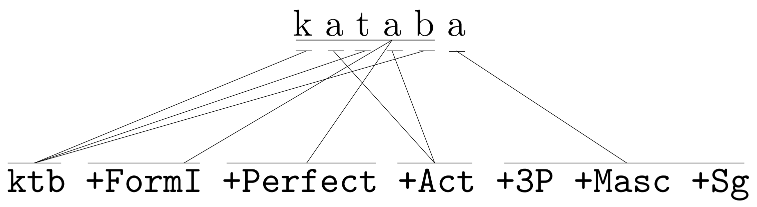


FIGURE 9.2. *Non-aligned information in a string-to-string Arabic verb parse.*

correspond to some components in the analysis, but this knowledge need not be represented in the finite-state transducer encoding the grammar. The job of the transducer is to translate analyses into surface words and the other way around, not directly to provide information about which segments contribute to which parts of the analysis and how. Since Finnish morphology is more or less concatenative one can include such alignment information in the transduction itself—that is, we may in some way or other force the transducer performing the transduction to align the letters in the input roughly with the corresponding categories in the output (as seen in figure 9.1). Even so, there are cases where one single segment directly contributes to several elements in a parse, as well as cases where the absence of a segment in a specific location can be interpreted as being morphologically meaningful. Both phenomena can be difficult to convey in a 2-level model.

When we consider other morphologies, such as verb formation in Semitic languages where morphemes are discontinuous and may be scattered in a string of segments, the situation becomes drastically more difficult to treat with finite-state transducers. A similar surface-word parse pair for Arabic is seen in figure 9.2 where the intricate relationship between the segments and symbols in the parse is far more conspicuous. The relationships explicitly drawn in the figure are completely absent from a simple string-to-string mapping: *kataba* → `ktb +FormI +Perfect +Act +3P +Masc +Sg`.

The problem of treating nonconcatenative morphology and the nonlinear alignment between words and the desired parses is severe enough to cause difficulty in merely constructing a finite-state grammar for the task, let alone providing correctly aligned word-parse pairs. This is also the problem to which k-tape automata were first suggested as a remedy (Kay, 1987).

Since the possibilities provided by k-tape automata for morphological analysis are difficult to describe in the abstract, we shall in this chapter primarily look at a more practical example–a verb grammar of Arabic—at the same time as we present the basic technique of grammar construction.

The fact that k-tape automata seem particularly suitable for nonconcatenative morphologies is really no surprise from a linguistic point of view. Over the years, various linguistic theories to deal with such phenomena have relied on multi-tiered representations of information in words. Multiple tiers in linguistic theories translate for the most part quite naturally to multiple tapes in a finite-state world. Although nonconcatenative morphologies provide perhaps the strongest case for moving from two tapes to multiple tapes in finite-state morphology, we will argue that the approach is not without merit even for highly concatenative morphologies.

The chapter is laid out as follows: first, we look at previous work done with multitape automata and natural language processing, in particular morphological analysis, in section 9.2. Section 9.4 discusses how k-tape automata allow one to capture discontinuous/non-concatenative morphology in such a way that morphemes are aligned with the semantic information they convey across multiple tapes. Section 9.5 shows the fundamental layout of an Arabic verb grammar with particular emphasis on the interaction of the root and the pattern in such morphologies. In section 9.6, the method of parsing and generation with the model is described and section 9.7 discusses some aspects of grammar design that is relevant for producing efficient parsers and generations as well as automata of manageable size.

9.2. Previous work

The special problems and challenges embodied by nonconcatenative morphologies have been recognized from the early days of applying finite-state methods to natural language morphological analysis. The language model which finite-state methods have been most successful in describing—a model where morphemes concatenate in mostly strict linear order—does not translate congenially to all morphologies. The type of root-and-pattern morphology found in Arabic and Hebrew has often been singled out as difficult to treat with the same techniques as concatenative morphologies (Lavie et al., 1988; Kataja and Koskenniemi, 1988; Sproat, 1992).

An early suggestion to use multitape automata in handling verb constructions in Semitic languages appears in Kay (1987). This model has later been pursued in different variants by Kiraz (1994, 2000) among others. Interestingly, large-scale multitape solutions containing the magnitude of information in standard Arabic dictionaries such as Wehr (1979) have not been reported.

From the literature on the topic, it appears that two wide-coverage morphological analyzers for Arabic that strive for reasonable completeness have been built: one by Xerox and one by Tim Buckwalter/LDC (Buckwalter, 2004). The Xerox analyzer relies on complex extensions to the finite-state calculus of one and two-tape automata (transducers) as documented in Beesley and Karttunen (2003), while Buckwalter's system—which many other Arabic NLP projects rely on—is a procedural approach written in Perl which decomposes a word and simultaneously consults lexica for constraining the possible decompositions. Also, in a similar vein to Xerox's Arabic analyzer, Yona and Wintner (2008) report on a large-scale system for Hebrew built on transducer technology. Most importantly, none of these very large systems are built around multi-tape automata even though such a construction from a linguistic perspective would appear to be a fitting choice when dealing with root-and-pattern morphology.

The multitape model by Kiraz (2000) is arguably the most detailed and complete one of previous multitape models. Kiraz's model is essentially the transducer model augmented to handle k-tuples as transitions. This is also basically the model that was rejected in chapter 7 because of complexity concerns.

9.3. Root-and-pattern morphology and finite-state systems

Before going further with the analysis of templatic morphology, let us briefly review the type of phenomena that need to be captured if one wants to analyze an Arabic verb. We shall focus almost exclusively on the root-and-pattern interaction common to most Semitic languages without going into great detail about the concatenative processes that also occur.

In Arabic, as in most Semitic languages, verbs have for a long time been analyzed as consisting of three elements: a (most often) triconsonantal root, such as *k-t-b* (ك ت ب), *f-ʕ-l* (ف ع ل), or *b-r-j* (ب ر ج), a vowel pattern containing grammatical information such as voice (e.g. the vowel *a*), and a derivational template, such as CVCVC indicating the class of the verb, all of which are interdigitated to build a stem, such as *katab* (كَتَب) (McCarthy, 1979). This stem is in turn subject to more familiar morphological constructions including prefixation and suffixation, yielding information such as number, person, etc, such as *kataba* (كَتَبَ), the third person singular masculine perfect form.

The process of forming a stem can be described as a three-way-merger of the vocalization pattern (for example *a*), the form (for example CVCCVC which is usually called form II), and a root (for example *ktb*). Figure 9.3 shows the analysis for how the stem *kattab* (كَتَّب) is formed like this.

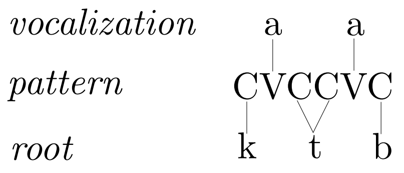


FIGURE 9.3. *The standard analysis of the makeup of Arabic verb stems.*

Form	Active(a)	Passive(u)	Pattern
Form I	katab	kutib	CVCVC
Form II	kattab	kuttib	CVCCVC
Form III	kaatab	kuutib	CVVCVC
Form IV	ˀaktab	ˀuktib	ˀVCCVC
Form V	takattab	tukuttib	tVCVCCVC
Form VI	takaatab	tukuutib	tVCVVCVC
Form VII	nkatab	nkutib	nCVCVC
Form VIII	ktatab	ktutib	CtVCVC
Form X	staktab	stuktib	stVCCVC

TABLE 9.1. *Derived Arabic verb stems using the root /ktb/.*

This is not the only possible analysis of the phenomenon. One can, for instance, make the simplifying assumption that the template exists as pre-merged with the vocalization leading to a two-way merge. Doing so would entail that instead of having to model the merging of `CVCVC` and *a*, we have a single lexicon of templates and vocalizations, containing entries such as `CaCaC` (Harris, 1941) . The three-way-analysis is the autosegmental analysis introduced by McCarthy (1979, 1981) and we shall roughly follow that model here as it provides for a better illustration of the interaction of multiple tapes in a multi-tape morphology. Ultimately, of course, which analysis to choose largely a matter of convenience and preference.

9.4. Semitic verb formation and a multitape analysis

Multi-tape descriptions of natural language morphology are appealing not only because such solutions seem to be able to handle Semitic verbal interdigitation, but also because a multi-tape solution allows for a natural *alignment* of information regarding segments and their grammatical features, which it does simply by virtue of its construction.

In what follows, we shall analyze the Arabic verb formation using an 8-tape automaton, where each tape carries distinct information as presented in figure 9.4. In the figure, we have marked the functions of these tapes in the leftmost column. The first tape (*input*) is

T_{input}	k	a	t	a	b	a
T_{root}	k		t		b	
T_{form}	**Form I**					
T_{ptrn}	C	V	C	V	C	
T_{paff}						a
T_{affp}						`+3P` `+Masc` `+Sg`
T_{voc}		a		a		
T_{vocp}						`+Act`

...

FIGURE 9.4. *An 8-tape representation of Arabic verbal morphology.*

actually what would normally be called the surface tape, representing actual surface forms of a fully inflected verb. In the illustration, the radicals on the *root* tape are aligned with the input, as is the pattern on the *pattern* tape, the suffix *-a* on the suffix tape, which again is aligned with the parse for the suffix on the affix parse tape (*affp*), and finally the vocalization *a* is aligned with the input and the pattern.

9.4.1. Goals

We assume here that what we want to do in designing the grammar is roughly as follows. For each input word (which is assumed to be a verb), we want to:

- provide the root and align the root with the input word so that it is clear which consonants (which may be discontinuous in the input word) belong to elements in the root
- provide the form that the verb was constructed from
- provide the pattern of this form, and show how the pattern is present in the input word
- provide the affixes involved and show by alignment where they occur in the input

- provide grammatical information about the affixes
- provide the vocalization pattern of the verb and show which segments in the input partake in the vocalization
- provide grammatical information about the vocalization (active/passive)

In short, what we want to do is write a grammar so that, given a verb, we can produce multitape parses exactly in the format of figure 9.4.

9.5. Grammar construction

The underlying encoding of the implementation is exactly as presented in chapter 7, where we construct a single-tape automaton that interleaves information about each of the k tapes. Since we have already developed an abstract formalism for combining individual tapes and expressing constraints across tapes, we need not worry about the actual encoding here, and will lean on the notation developed in chapter 7.

We construct a finite-state 8-tape simulation grammar in two steps. First, we populate each 'tape' with all grammatically possible strings for that tape. That means that, for our Arabic representation, the root tape should allow all possible roots we wish to accept, the template tape all the possible templates, the *form* tape all possible forms, the *pattern* tape all possible patterns, etc. At this point, we pay no attention what contents we allow across tapes: the root tape may contain a root which is a complete mismatch between the input tape or any other tape, but that is allowed so long as the individual tapes contain only the type of information they are intended to contain.

Naturally, we will need to have a prespecified set of lexica for each of the tapes. So, the regular language lexicon for the pattern tape $L_{pattern}$ will contain the strings `CVCVC`, `CVCCVC`, `CVVCVC`, etc. while the lexicon for the 'form' tape L_{form} contains the strings `Form I`, `Form II`, etc., and the root tape L_{root} all the possible roots in our lexicon, and so forth.

We call this language where each tape has been declared to contain all the individually possible strings for that tape, R_{base}. The second step is to constrain the co-occurrence of symbols on the individual tapes. We will perform this by a set of cross-tape constraints we call R_{rules}. This set of rules constrains for instance the root tape in such a way that it is aligned with the same consonant on the input tape as well as the symbol `C` on the pattern tape, among other things.

Our grammar then consists of all the permitted combinations of tape symbols allowed by both a) the R_{base} and b) R_{rules}. The resulting grammar is simply the intersection of the base and the rules viz.:

$$R_{base} \cap R_{rules}$$

9.5.1. Populating the tapes

Recall from chapter 7 (page 199) that we can align strings in many different ways on a tape depending on how we distribute the blanks. We will here use all three of the example alignment functions presented:

- $\text{Align}_{any}(L)$
- $\text{Align}_{rightblanks}(L)$
- $\text{Align}_{edgeblanks}(L)$

The reason is fairly obvious: when we declare the possible contents of each tape individually, we need not align blanks arbitrarily. Figure 9.4 should make it clear that, for instance, pattern strings, such as `CVCVC`, that occur on the pattern tape will be continuous and without intervening blanks. On the other hand, we will want to allow for blanks before and after the pattern strings since Arabic verbs may contain prefixes and suffixes which occur before and after the pattern on the other tapes.

The particular alignment we will want to follow for each of the tapes is as follows:

- tape 1 (*inputs*): Align$_{rightblanks}$
- tape 2 (*roots*): Align$_{any}$
- tape 3 (*forms*): Align$_{rightblanks}$
- tape 4 (*templates*): Align$_{edgeblanks}$
- tape 5 (*affixes*): Align$_{any}$
- tape 6 (*affixparses*): Align$_{edgeblanks}$
- tape 7 (*vocalization*): Align$_{any}$
- tape 8 (*vocparses*): Align$_{rightblanks}$

As we have settled for an appropriate alignment of the different tapes, we can proceed to declare the entire contents of R_{base}. We assume here that each tape's possible contents are declared in the regular languages L_{inputs}, L_{roots}, L_{forms}, $L_{templates}$, $L_{affixes}$, $L_{affparses}$, $L_{vocalization}$, $L_{vocparses}$. With this information, we can now declare the R_{base} language using the tape insertion functions:

$$\begin{array}{ll}
R_{base} = & \\
\mathfrak{T}_8^1(\text{Align}_{any}(L_{inputs})) & \cap \\
\mathfrak{T}_8^2(\text{Align}_{any}(L_{roots})) & \cap \\
\mathfrak{T}_8^3(\text{Align}_{rightblanks}(L_{forms})) & \cap \\
\mathfrak{T}_8^4(\text{Align}_{edgeblanks}(L_{templates})) & \cap \\
\mathfrak{T}_8^5(\text{Align}_{any}(L_{affixes})) & \cap \\
\mathfrak{T}_8^6(\text{Align}_{edgeblanks}(L_{affparses})) & \cap \\
\mathfrak{T}_8^7(\text{Align}_{any}(L_{vocalization})) & \cap \\
\mathfrak{T}_8^8(\text{Align}_{rightblanks}(L_{vocparses})) & \cap
\end{array}$$

This will produce the 8-tape language R_{base} where individual tape contents are acceptable, but where strings on each tape may incorrectly co-occur with the wrong strings on other tapes. For instance, the pattern tape may contain a `C` aligned with a vowel on the input tape. In effect, the next task is to rule out such combinations by a the set of constraints collectively referred to as R_{rules}.

9.5.2. Constructing the rules

When constructing the rules that constrain the co-occurrence of symbols on the various tapes we shall make use of the logical notation developed in chapter 7. Indeed, all of our rules will consist exclusively of such statements, and the strategy is to intersect all the individual rules as use this as a filtering language to remove from R_{base} illegitimate co-occurrences across tapes.

Let us begin with a simple rule: in order to constrain the template against the input and root tapes, we need a rule that effectively says that every C symbol occurring on the template tape (tape 4) must be matched by a) a consonant on the root tape (tape 2) and b) a consonant on the input tape (tape 1). That is:

$$(\forall_8 x)(\mathfrak{T}_8^4(x \in C) \rightarrow (\mathfrak{T}_8^2(x \in L_{cons}) \wedge \mathfrak{T}_8^1(x \in L_{cons}))) \tag{9.1}$$

We assume here that the language L_{cons} contains all the consonant segments in the language.

Similarly, we want to constrain the `Forms` parse tape that contains symbols such as `Form I`, `Form II` etc., so that if, for example, `Form I` occurs on that tape, the pattern `CVCVC` must occur on the pattern tape. Hence, we need a rule like:

$$(\forall_8 x)(\mathfrak{T}_8^3(x \in Form\ I) \rightarrow ((\mathfrak{T}_8^4(x \in C) \wedge (S(x, VCVC))))) \tag{9.2}$$

and similarly for the other patterns.

There is one interesting constraint that we have not yet discussed which is necessary for proper treatment of Arabic orthography. Arabic words are usually written unvocalized and it is left up to the reader to figure out, if there is ambiguity, which of the possible forms is intended. In most cases, only ambiguities that are difficult to resolve even knowing the context are marked with the disambiguating vocalization markings. In a morphological system, we will want to be able to match and parse fully vocalized words such as *wadarasat* (وَدَرَسَتْ), fully unvocalized ones, such as *wdrst* (ودرست), or partly vocalized words. This is the reason we have allowed *any* alignment for the input word: since the pattern tape will contain patterns such as `CVCVC`, any unvocalized words on the input tape must be able to match the pattern. However, if we required a `V` on the pattern tape to always align with a vowel on the input tape, we would lose this flexibility. To address this, what we can do is declare 'vowels' on the input tape to be optionally blank and use two constraints to the effect that a) a blank on the input tape must be matched by a vowel on the pattern tape, and b) a `V` on the pattern tape must be matched by either a blank or a vowel on the input tape. Constraint a) in effect rules out blanks on the input tape being in any other position except for unvocalized vowels within the pattern. The constraints are then:

$$(\forall_8 x)(\mathfrak{T}^1_8(x \in \Box) \rightarrow (\mathfrak{T}^4_8(x \in V))) \tag{9.3}$$

$$(\forall_8 x)(\mathfrak{T}^4_8(x \in V) \rightarrow (\mathfrak{T}^1_8(x \in (\Box \cup L_{vowel}))) \tag{9.4}$$

$$\tag{9.5}$$

To give an overview of some of the subsequent constraints that are still necessary, we include here a few descriptions and examples (where the starred (***) tape snippets exemplify illegal configurations):

- Every root consonant has a matching consonant on the input tape

T_1	k	a	t	a	b	a
T_2	k		t		b	

T_1	k	a	t	a	b	a
T_2***	d		r		s	

- A vowel or blank in the input which is matched by a V in the pattern, must have a corresponding vocalization vowel

T_1	k	a	t	a	b	a
T_4	C	V	C	V	C	
T_7		a		a		

T_1	k	a	t	a	b	a
T_4	C	V	C	V	C	
T_7***		u		i		

- A position where there is a symbol in the input either has a symbol on the pattern tape or a symbol on the affix tape (but not both)

T_1	k	a	t	a	b	a
T_4	C	V	C	V	C	
T_5						a

T_1	k	a	t	a	b	a
T_4	C	V	C	V	C	
T_5***						

9.5.3. The final automaton

Assuming we now have a set of rules encoding the cross tape constraints of the type discussed above, our rule set the becomes

$$R_{rules} = R_1 \cap \ldots \cap R_n$$

and we can construct the final grammar as

$$G = R_{base} \cap R_{rules} \tag{9.6}$$

9.6. Parsing and generation

With a complete grammar encoded as an 8-tape automaton, the question of parsing a word proceeds exactly as discussed in chapter 7: we construct an automaton containing arbitrary sequences of strings on all of the other tapes except the input tape, where we place the word to be parsed. In the grammar under discussion, we have chosen the format of the input tape to be such that it can contain blanks (in lieu of vowels), and so these blanks must be taken into account as well. However, we can use the alignment functions to handle this. In effect, we create a machine that accepts the input word w on the input tape (with any alignment of blanks), and intersect this machine with the grammar. The resulting automaton contains all the legal parses:

$$\mathfrak{T}_8^1(\text{Align}_{any}(w)) \cap G \tag{9.7}$$

For the case of generation, we proceed in exactly the same way. Generation and parsing are formally indistinguishable operations under the model—in both cases we supply information on one or more tapes, intersect that against the grammar, and inspect what the other tapes can legally contain. If we want to know all the possible grammatical forms given some set of contents on tapes other than the input tape, we create a multitape automaton where the different tapes carry this information, after which we intersect that with the grammar and inspect the contents of the result.

For example, if we wanted to know all the possible grammatical combinations of `Form I` and the root `drs`, we can do so by calculating

$$\mathfrak{T}_8^2(\text{Align}_{any}(\texttt{drs})) \cap \mathfrak{T}_8^3(\text{Align}_{rightblanks}(\texttt{Form I})) \cap G \tag{9.8}$$

producing a large number of forms of tape 1: *drs, drsat, darst, darasat, durst, durisat, durisa*, etc. etc.

9.7. Efficiency considerations in grammar design

When constructing morphological grammars in this fashion, there are a number of things one can do to avoid unnecessary growth in the size of the grammars in their automaton representations. As is the case when one constructs finite-state transducer grammars, in multitape grammars too there are certain types of constructions that can quickly cause the automata involved to explode in size when dealing with multitape grammars as well.

9.7.1. Information alignment across tapes

The most crucial concern is to keep related information across different tapes aligned. This is self-evident from the point of view that alignment is one of the main reasons we are pursuing multitape grammars in the first place, but there are more compelling reasons for this outside of grammar elegance. Non-aligned constraints across multiple tapes immediately cause the automaton to grow very quickly, just as happens with a single tape. If there is a string sequence pqr, p being on a different tape than r, with q intervening, the set of states that accept the string q should be kept as small as possible since each of these need to remember, for different possibilities of p, which string r should be allowed at the end. The situation is analogous to the way in which a large number of circumfixes markedly augment the size of a traditional finite-state transducer grammar.[1]

Now, for the grammar given above, it should be noted that most constraints are very strictly local to within a few symbols, depending slightly on the ordering and function of the tapes. Rule (9.1), for instance, which constrains consonants and `C` symbols, is strictly local and any dependencies are resolved within the same ‘column’ of the multitape machine. Had we placed the pattern strings of the type `CVCVC` completely nonaligned with the actual consonants and vowels they refer to in the input, the constraints would of course be much harder to write and the resulting automaton much larger in size.

Of course, some long-distance constraints will be inevitable. For example, `Form II` is generally described as a `CVCCVC` pattern, where the extra consonant is a geminate, as in the stem `kattab`, where the *t* of the root associates with both `C`’s in the pattern. To distinguish this `C` behavior from that of `Form X` which is also commonly described with two adjacent `C` symbols where, however, there is no such association (as in the stem *staktab*)

[1]In general, every circumfix doubles the number of states of the automata that encode the intervening material.

we need to introduce another symbol. This symbol $\mathtt{C}_2$ occurs in `Form II`, which becomes $\mathtt{CVCC}_2\mathtt{VC}$. We then introduce a constraint to the effect that any $\mathtt{C}_2$-symbol must be matched on the input by a consonant, which is identical to the previous consonant on the input tape.[2] This of course creates a dependency across columns, since we must ensure that the two consonants are identical in the input if matched by the sequence $\mathtt{CC}_2$ in the pattern. These non-immediate dependencies can be avoided to some extent by grammar engineering, but not in all cases.

Naturally, the overarching goal of quickly resolving dependencies by aligning strings that bear on each other is a goal of linguistic interest as well: a grammar which does not align grammatical information with segments in the input is likely not a good grammar. However, there are still a couple of ways in which one can go astray. These usually involve cases where nonalignment is not necessarily a blemish in the grammar, but still causes unnecessary growth in the automaton encoding it.

For instance, in the running example we have presented, one of the parse tapes has included the symbol `+3P +Masc +Sg`, aligned with the affix that represents the grammatical information:

...

T_5						a
T_6						+3P +Masc +Sg

...

If, however, it be the case that what the parse tape reflects is a prefix or a circumfix, as will be the case with the imperfective, subjunctive and jussive forms, the following alignment would be somewhat inefficient:

[2]The idea to preserve the gemination in the grammar is similar to the solutions regarding gemination and spreading of Forms II, V, and IX documented in e.g. Beesley (1998).

...

T_5	t	a						
T_6								+3P +Fem +Sg

...

This is because the morpheme that partly indicates third person singular feminine in the imperfective is the prefixation of *ta*, whereas the same grammatical function in the perfective is the suffix *at*. So if we adhere to a static scheme in the annotation and always place the related parse in the same slot, we will create a long-distance dependency and hence duplicates of states in the automaton for all the intervening material. A more efficient strategy is to place the parse or annotation tape material as close as possible to the segments which have a bearing on it, i.e., in the imperfective we would prefer:

...

T_5	t	a						
T_6	+3P +Fem +Sg							

...

This alignment can be achieved by a more general constraint in the grammar to the effect that the first non-blank symbol on the affix tape is in the same position as the first non-blank symbol on the affix parse tape.

9.8. Discussion

In this chapter we have implemented the notation and construction strategies developed in earlier chapters to illustrate the design of actual morphological grammars using multi-tape automata as the model. We have shown how nonconcatenative phenomena, root-and-pattern morphology in particular, can be treated with the tools outlined in chapter 7.

```
INPUT     s    t    a    k    t    a    b    a    t
ROOT                     k    t         b
FORM      X
PATTERN   s    t    V    C    C    V    C
AFFIX                                        a    t
AFFPARSE                                     TFS
VOC                 a              a
VOCPARSE  Act
```

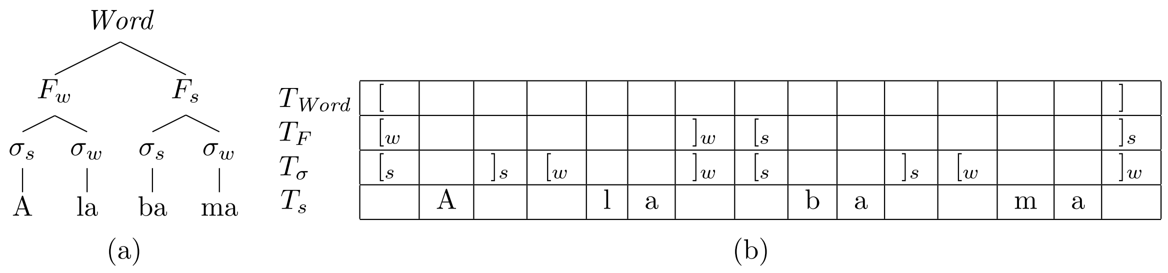
FIGURE 9.5. *Example parse of the third person female singular active form based on pattern X and the root /ktb/.*

The topic and central phenomenon chosen to serve as the case study in the chapter is that of Arabic verb formation, where we have highlighted the central parts of an actual multitape implementation. The reason for this choice is that the patterns exhibited in root-and-pattern morphology are very difficult to capture through standard finite-state transducer grammar construction methods.

Given the construction method presented for nonlinear morphology, it should be fairly straightforward to apply the same technique for concatenative morphologies as well. Although the direct need for a multitape construction of concatenative phenomena may not be as acute, the method at hand offers advantages for such cases as well. As already mentioned, the richer annotation one gains by distributing different kinds of information—even different morphemes—across various tapes, is in itself an advantage that is not present in the now-standard models of finite-state morphology.

There is perhaps one additional question that deserves comment with respect to linguistic theory. As was mentioned earlier, there is a strong connection between multitape automata and different hierarchical theories of phonology and morphology—in particular 'autosegmental' phonologies. Would it not be more profitable to pursue a notation based on already established linguistic theories and show how such a notation could be implemented in practice?

While this may be useful in some respects, particularly with implementations of lin-



T_{Word}	[														]
T_F	$[_w$						$]_w$	$[_s$							$]_s$
T_σ	$[_s$		$]_s$	$[_w$			$]_w$	$[_s$			$]_s$	$[_w$			$]_w$
T_s		A			l	a			b	a			m	a	

FIGURE 9.6. *Suprasegmental features represented as multitape encodings.*

guistic theory, it is not the intention here to show that various theories of linguistics can be expressed in the language of finite-state machines. There is a vast literature on the topic, and judging from the empirical coverage of morphological and phonological grammars implemented in the finite-state vein, it would be unlikely that linguistic theories of phonology and morphology could fail to translate to a finite-state model. Linguistic theory is a moving target—the basic methods of applying finite-state machinery to phonology and morphology are nearly three decades old by now, and as such, have outlived numerous linguistic theories. Further, one may argue that keeping the notation on a more abstract level allows for flexibility in modeling various linguistic theories and using the finite-state tools available for linguistic research. That is, by not confining the notation to capture a very restricted kind of model, it could be hoped that future pursuits in the realm of theoretical development could take advantage of an abstract notation that can be fitted for various purposes.

It is of course obvious that the multitape encoding and application suggested here is open-ended enough to accommodate a number of annotations. Suprasegmental features, such as tone, syllables, feet, metrical units—indeed most of the word-internal structure assumed in many theories of linguistics—can easily be interpreted as a formalism where certain types of information appear on different tapes which are strictly regulated in their internal and cross-tape configurations. One such encoding is suggested in figure 9.6, where a structure of prosodic categories, roughly following the labeling of Selkirk (1980) is shown together with a multitape representation of the same structure. Here, there is a separate tape for words, feet, syllables, and segments, and the syllable tape shows the alternation of strong and weak syllables, as does the foot tape.

The above example, and the different phenomena treated in this chapter together suggest that one may extend the technique to model, as a whole, linguistic structures outside the domain of the word as well. If we move away from the word as our unit of computational analysis questions are of course raised regarding the generative power of finite-state models. That finite-state approaches cannot model unlimited recursion would seem to hamper the extension of this approach to larger and larger domains. However, it seems very likely that some strategy of finite-state approximation[3] of phrase structure would be successful in this respect.

[3]See e.g. Langendoen and Langsam (1984); Mohri and Nederhof (2000); Nederhof (2000); Hulden (2009c) among others.

10. CONCLUDING REMARKS

The investigations presented here fall into three broad categories in the domain of finite-state machines: fundamental algorithms used in finite automata manipulation, open problems in finite-state grammar development, and new methods for natural language modeling. Each of these three categories draws on distinct areas of previous work, but all attempt to facilitate the practical and theoretical modeling of natural language with finite state machines.

The first category is that of implementing and modifying fundamental algorithms found in computer science for manipulating finite automata. The availability of such modifications is a cornerstone of the successful deployment of natural language grammars and of systems based on finite-state automata and transducers. The modifications include the development of efficient variants of determinization and minimization algorithms for finite automata, as well as the introduction of a number of practical algorithms necessary for finite-state transducer construction.

The second important contribution is the solving of a number of open problems inherent in the already well-established aspects of finite-state grammar development. The ability (or non-ability) to model reduplication phenomena, for instance, has been a long-standing problem in finite-state morphology. In chapter 4, an operator was developed for asserting the equivalence of substrings in a language. While the definition of the operator is deliberately made quite simple with the motivation that it should be easy to integrate into well-designed grammars, its construction requires quite an intricate manipulation of simpler automaton and transducer operations.

Continuing with this theme, the investigations in chapter 5 focused on decision properties and algorithms relating primarily to finite-state transducers. The core new algorithms here—deciding functionality, deciding ambiguity, deciding functional and unambiguous transducer equivalence, and extracting unambiguous and ambiguous parts of transducers—

are likely to find applications both in language engineering as well as linguistic research. Dealing with ambiguity—knowing when and how it occurs in a grammar—is a prerequisite for the finite-state grammar designer. Testing the equivalence of two different grammars constructed in different formalisms (in the circumstances that it is possible to do so) and extracting non-equivalent parts of complex grammars should be fundamental tools in the formal investigation of systems and theories of phonology and morphology. As has been shown, most morphological and phonological formalisms and methods can be quite easily translated into a finite-state notation of the type presented in chapters 3 and 8.

One of the overarching goals of the implementation of all these algorithms is to create a tool that would serve as an aid to 'pencil-and-paper' investigations into natural language generalization and phenomena. New linguistic theories and notations are continually being developed and researched. It remains, however, very difficult for human researchers to answer simple questions about the relationship between two proposals, such as: does the new theory cover the phenomena that the old theory covers, and does it give the same predictions? The results in chapter 3, combined with previous experience of many researchers in the field, strengthen the argument that finite-state machines are viable candidates to serve as a neutral intermediary for linguistic investigation. Finite-state machines can model most phenomena and theories in morphology and phonology and—because of their mathematical simplicity—make available a number of testing tools by which their behavior can be investigated. Very few computational models enjoy such properties. Context-free grammars, for instance, which serve as a backbone for many theories of syntax, are much more restricted in the types of questions that can be asked about them: even the equivalence of two context-free grammars is not decidable by any algorithm. The equivalence of two augmented models of context-free grammars—say, one that allows for transformations (or movement) of constituents, and another that allows for unification of features—lies well beyond the type of questions that can be answered by computational methods. Seen from this perspective, finite-state models stand on solid ground as a neutral device for implementing various notations and formalisms and thus as a basic research tool in formal linguistics. However, as was noted in chapter 5, not all questions that would be interesting from the point of view of finite-state transducer based linguistic investigation can be answered by algorithms: equivalence of two unrestricted finite-state transducers is an example of such a question. Further investigations in this domain may reveal a satisfactory subset of finite-state translation devices that retains all the linguistically useful properties of the unrestricted finite-state transducer model, yet is sufficiently restricted to allow for algorithms to answer all the fundamental questions one may want to ask about one or several such machines. The current state of finite-state technology is quite close to attaining some of the ideals one could wish for in as a generic computational model of morphology and phonology: a model that enjoys flexible closure properties and affords the development of linguistically motivated operators, and one where questions, such as equivalence and inclusion, can be answered in regards to different instances of the model.

The third substantial part of the investigation has been concerned with new finite-state methods and formalisms for natural language modeling. The foremost contribution here is the development of a predicate logic formalism that can be used instead of, and together with, more classical regular expression-type operations to construct finite-state machines.

The basic technique that makes the predicate logic construction possible—manipulation of auxiliary symbols in automata—has been a common way of solving difficult-to-construct language problems. However, it has not been previously elucidated or formalized, nor has it been described in terms of first-order predicate logic over substrings. The logical formalism, which takes advantage of quantifiers and propositions as well as of regular expressions themselves, offers a systematic abstraction away from the more low-level regular expression-type building blocks discussed in the early chapters. As has been seen, constructing complex finite-state machinery that models natural language requires that one can abstract away from the building blocks of automata (states, arcs, and symbol labels) and introduce a high-level notation of regular expressions in their place. The predicate logic, however, argues for still another layer of abstraction, away from regular expression-based notation, and toward a more constraint-based, easily verifiable formalism.

Patterns in the predicate logic over substrings arguably possess a certain kind of naturalness which is not present in regular expressions: a simple statement such as "every substring A is followed by some substring B" translates almost as is into predicate logic over substrings $(\forall x)(x \in A \rightarrow S(x, B))$. The corresponding language, described as a regular expression using double negation is quite convoluted and has a certain air of unintuitiveness to it: $\neg(\Sigma^*\ A\ \neg(B\ \Sigma^*))$. Yet, this kind of an if-then expression is an omnipresent type of generalization as regards linguistic phenomena. Facts about morphotactics and phonotactics in a particular language are very often precisely of this nature: 'every closed syllable has property X'; 'every morpheme of type X must be followed by a morpheme of type Y'; 'A stop in the onset of a syllable is never followed by a continuant,' etc. This observation, together with the practical problems that are solved with the predicate logic, argues for a central placement of the formalism in linguistic modeling of morphology and phonology.

Another new model, a method of manipulating multitape automata, was developed in chapter 7, and applications derived from it in chapters 8 and 9. The use of multitape automata for encoding phonology and morphology has been previously motivated and attempted in the literature, but the direct extension of the finite-state transducer model involves technical problems not easily overcome. By contrast, the new multitape model presented relies on simulation of multitape automata by a single tape automaton which allows for the direct transfer of all the relevant algorithms and construction methods for single-tape automata to the multitape model.

The multitape model uses the predicate logic over substrings to express constraints across the different tapes (or tiers). This joint multitape-logic approach can be seen as an alternative formalism for constructing morphological and phonological grammars and expressing linguistic generalizations within the domain of the word-formation processes. The formalism inherently allows for capturing multi-linear grammatical facts by placing different kinds of morphological and phonological information on different tapes.

The multitape model also offers a solution to other problems in the domain of finite-state language processing. The task of constructing complex transducers, such as those encoding replacement rules, can be made significantly easier by employing a multi-tape intermediate stage in such construction. A fairly complete implementation of compiling various types of string rewriting formalisms to finite-state transducers using the multitape model was introduced in chapter 8.

The long-term linguistic usefulness of the multitape automaton model is slightly difficult to assess and compare against established models. Other established models of finite-state morphology—the two-level model and the cascaded transducer model—have been developed and refined over a number of years as feedback has been received from the construction of a large number of wide-coverage grammars. Subsequent investigation may reveal shortcomings or necessary additions to this logic-multitape description of morphology and phonology. However, to reach the point where feedback from several actual grammar implementations finds its way into an implementation requires a fairly thorough practical implementation of the logic-multitape formalism in the first place. To this end, it would be a desirable endeavor to develop a complete implementation of all the aspects and conceivably useful operators related to the logic-multitape method. Preferably, this implementation would be shaped in a form distinct from the regular expression based tool that has been developed in conjunction with this dissertation. The reason for this suggestion lies in the desire to maintain a limited, well-circumscribed set of operations by which to model natural language, and which would be accessible to linguists and computational linguists alike.

A. Foma

Foma is a finite-state machine compiler and programming language written in C that implements all the basic operators described in chapter 3, as well as the predicate logic formalism of chapter 6. The following is a brief introduction to the functionality of the compiler to help illustrate the subsequent example scripts in appendix B.[1]

A.1. Brief examples

To enter basic regular expressions and compile them to a finite-state machine, the command `regex` is used. For example, entering:

```
foma[0]: regex [a|b]*;
```

foma responds by giving some statistics about the network constructed:

```
1 states, 2 arcs, Cyclic.
```

More information can be extracted by the command `print net` (or simply `net`, an equivalent abbreviated form):

```
foma[1]: net
Sigma: a b
Size: 2.
Net: 66334873
Flags: deterministic pruned minimized epsilon_free
Arity: 1
Sfs0: a -> fs0, b -> fs0.
```

The command `view net` can be used to view a graphic representation of the finite-state machine.

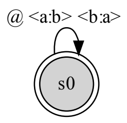
Foma, by default minimizes all the finite-state automata it constructs. Finite-state transducers are minimized in the FSA-sense, where label pairs are considered single symbols.

To describe a simple transducer that, for instance, changes all `a` symbols to `b`, and `b` symbols to `a`, leaving everything else unchanged, you can enter:

```
foma[0]: regex [?-a-b | a:b | b:a]*;
foma[1]:
```

producing the network:

[1]The application and more documentation is available at `http://foma.sf.net`.



which takes advantage of the cross-product operator and the special symbol `?` which matches any symbol at all. *foma* has extensive support for constructing complex transducers that rewrite sequences of strings into other strings, and the above example could have been created more easily by:

```
foma[0]: regex a -> b , b -> a;
foma[1]:
```

using the string rewriting operators.

As regular expressions are entered with the `regex` command, *foma* stores them on an internal stack. The number at the prompt refers to the number of networks stored on the stack. Many automata operations apply by default to the last network defined, i.e. the top one on the stack. For instance, to run a word through the above automaton, one can give the command:

```
foma[1]: down abxa
baxb
```

if the intention is to test the output of several words and the machine, one can type the commands `up` or `down` to enter a subshell where words can be typed one after the other, applying them to the top network on the stack in either the downward or upward (inverse) direction.

```
foma[1]: up
apply up> baxb
abxa
apply up> bbaa
aabb
apply up>
```

CTRL-D returns to the main interface.

A.2. Labeling networks

Regular expressions can be compiled into finite-state networks, and then labeled to reuse them in later expressions. For instance, one can define two networks:

```
foma[0]: define ContainsA ?* A ?*;
defined ContainsA: 2 states, 4 arcs, Cyclic.
foma[0]: define ContainsB ?* B ?*;
defined ContainsB: 2 states, 4 arcs, Cyclic.
```

and then use them in further definitions or regular expressions, e.g.:

```
foma[0]: regex ContainsA & ContainsB;
```

yielding the language that contains at least one `A` and at least one `B`.

These previously defined networks can be listed with:

```
foma[1]: print defined
ContainsA 2 states, 4 arcs, Cyclic.
ContainsB 2 states, 4 arcs, Cyclic.
```

A.3. Declaring functions

Regular expression functions can also be defined in a similar way. The format for declaring functions is as follows:

```
define FunctionName(Prototype) Regular expression;
```

For example:

```
foma[1]: define Contains(X) [?* X ?*];
defined Contains(@1)
```

defines a function of one variable. One can then use this function in more complex regular expressions:

```
foma[1]: regex x Contains(a b c) x;
6 states, 26 arcs, Cyclic.
```

producing the language that starts and ends with `x` and contains at least one instance of `abc` somewhere in between.

The command `print defined` prints not only the labeled networks, but also all user-defined functions.

```
foma[1]: print defined
ContainsA 2 states, 4 arcs, Cyclic.
ContainsB 2 states, 4 arcs, Cyclic.
Contains(@1) [?* @ARGUMENT01@ ?*];
```

Foma contains a number of built-in functions that follow the same calling conventions as user-defined ones (see the section on regular expression operators). These all begin with the underscore (_) character. For example, the regular expression:

```
foma[0]: regex _sigma(X);
```

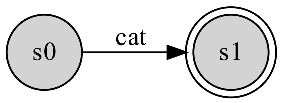
creates a new language that is a single-symbol string from the alphabet of `X`.

A.4. Symbols

By default, *foma* matches strings inside a regular expression by longest-match. This means that a regular expression such as:

```
foma[0]: regex cat;
```

will produce the network



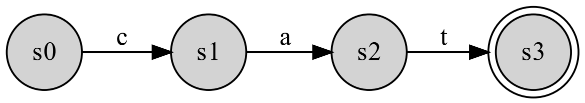
which is probably not what was intended, since the string `cat` is now a single symbol. If what was desired was to have the concatenation of the three symbols `c`, `a`, and `t`, one way to enforce this is to say:

```
foma[0]: regex c a t;
```

or equivalently

```
foma[0]: regex {cat};
```

producing the intended:



A.5. Automata and transducers

Foma makes no serious distinction in its operation between finite-state automata and finite-state transducers, seen for instance in the application of words against a network.

Supposing we use a few of the complex operators to define the language that models the old spelling rule: *'i' before 'e' except after 'c'*:

```
foma[0]: regex ~$[\c e i|c i e];
5 states, 18 arcs, Cyclic.
```

We now have an automaton that accepts all words that follows the rule, and rejects all that do not. If we now apply a word to the automaton, it is simply echoed back if it is accepted:

```
foma[1]: down friend
friend
```

because the automaton acts like a finite-state transducer, where every input label is identical to the output label.

If we apply a word not in the language, the following happens:

```
foma[1]: down weird
???
```

The only distinguishing mark between an automaton and a transducer is the **arity** number of the network, displayed with `print net`: 1 (automaton) or 2 (transducer).

A.6. Complex operators

Foma provides a number of complex operators built in, as seen in the above example, which took advantage of `$X` (contains an instance of X) and `\X` (any single symbol but `X`). Going with the above example and other complex operators we can easily create a transducer that "marks" every violation of the *i before e* rule with brackets.

```
foma[0]: regex \c e i | c i e -> "[" ... "]";
11 states, 44 arcs, Cyclic.
foma[1]: down
apply down> friend
friend
apply down> weird
[wei]rd
```

A.7. Scripts

Sequences of command definitions and regular expressions can be stored in script-files and consulted by either starting up foma with `foma -l <filename>`, or the command `source <filename>` in the interface.

A.8. Unicode

Foma assumes that all input is UTF-8. This is the only encoding supported. Almost all non-alphanumeric characters in the ASCII range are reserved, and so need to be escaped. For example, to create the language that accepts only the string: **a+b**, one has to enter:

```
foma[0]: regex a %+ b;
```

Foma also supports multiple variants for most operators, with the variant operators taken mostly from the Unicode ‘mathematical operators’ block. This means that one can say things like:

```
foma[0]: regex [A ∩ B] ∪ [C⁻¹ ∘ D₂] ∪ ¬[E × ε]⋆;
```

or

```
foma[0]: regex [A & B] | [C.i .o. D.l] | ˜[E .x. 0]⋆;
```

and the two statements are equivalent. See the section on regular expression operators for a listing of some of the operators and the variants.

To use unicode characters beyond the ASCII range in a string, one can either enter them directly, or through the `"\uXXXX"` expression. For example, if we wanted to enter the reserved RING OPERATOR (which is the composition operator) in a regular expression, we could enter it as:

```
foma[0]: regex "\u2218";
```

or

```
foma[0]: regex "∘";
```

or

```
foma[0]: regex %∘;
```

A.9. Other formalisms: lexc

Foma supports the reading of scripts in some other formalisms, such as the *lexc*-language for defining lexicons. Such files are read and compiled by the command `read lexc`. For example:

```
foma[0]: read lexc lexicon.lex
...
Building lexicon...Determinizing...Minimizing...Done!
188181 states, 315786 arcs, Cyclic.
foma[1]: define MyLexicon;
defined MyLexicon: 188181 states, 315786 arcs, Cyclic.
foma[0]:
```

reads the *lexc*-format file in `lexicon.lex`, compiles it into a finite-state machine, after which it is named `MyLexicon`.

A.10. The alphabet

Foma uses three special symbols on the transitions of its networks: `EPSILON` (`0`), `IDENTITY` (`@`), and `UNKNOWN` (`?`).

However, only two of these have special meaning in regular expressions: `EPSILON` (`0`) and `ANY` (`?`). This discrepancy between special symbols on arc labels and regular expressions is an artifact of the way *foma* compiles regular expressions and dynamically determines the alphabets. From the point of view of writing regular expressions, the semantics are simple: `0` is the empty string and `?` means any single symbol.

When compiling a regular expression, the alternate symbols `0`, `[]`, or ε may all be used to denote the empty string. For instance:

```
regex a -> 0 || c _ ;
regex a -> [] || c _ ;
regex a -> ε || c _ ;
```

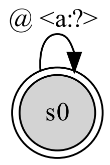
all mean the same thing and compile to the same network.

As for the other two special symbols, `@` and `?`: the symbol `@` on a transition is interpreted as the identity relation of any symbol not in the alphabet of the network, i.e. `a:a` `b:b`, etc., assuming `a` and `b` are not part of the alphabet. The special symbol `?` likewise refers to any symbol not in the alphabet of the network, however, it only occurs on one side of a label, e.g. `a:?`. The symbol `a:?` will translate an `a` into any symbol not in the alphabet. The special combination `?:?` will translate any symbol to any other (nonidentical) symbol.

Consider a regular expression:

```
regex a -> [? - a];
```

i.e. rewrite the symbol `a` as any single symbol except itself, which compiles into the network:



For the replacement rule to be encoded properly into the transducer, we need the two different transition symbols `@` and `?`, `@` to pass through any symbol except `a` unchanged, and `?` to refer to any possible single symbol (which is this case does not include `a` since it is in the alphabet of the network).

A.11. Regular expression operators

Optionality `(X)`

Defines the language or relation. (X) is equivalent to `[X | 0]`.

Substitution `` `[X,Y,Z] ``

The language `X` where symbols `Y` are substituted for `Z`. The result may be non-deterministic, in particular if `Z` is `0`. The result is determinized and minimized and any substituted symbols are purged from the alphabet.

Term negation `\X`

Any single symbol, except `X`. Equivalent to `[? - X]`. Note that `.#.`—the abstract boundary marker used in context restriction rules and replacement rules—is not considered a symbol.

Cross product/Cartesian product `X:Y`

Produces a transducer that represents the relation that maps any string from `X` to any string in `Y`.

Kleene Star `X*`

Zero or more iterations of `X`. Equivalent to `[X+ | 0]`.

Kleene Plus `X+`

One or more iterations of `X`. Equivalent to `[X X*]`.

Iteration operators: m,n-ary iterations `X^n, X^>n, X^<n, X^{m,n}`

Denotes the languages or relations where `X` is concatenated with itself `n` times (`X^n`), more than `n` times (`X^>n`), less than `n` times (`X^<n`), or from `m` to `n` times (`X^{m,n}`).

Domain/range extraction `X.u`, `X.l`

Extracts from a transducer/relation the domain (.u) or range (.l), also called the upper (1st) and lower (2nd) projections, respectively. The Unicode equivalents X_1 and X_2 may also be used. The expressions `X.1` and `X.2` are also equivalent.

Inversion `X.i`

Inverts a transducer. The Unicode equivalent is X^{-1}.

Flag elimination `X.f`

Eliminates all flag diacritics from the FSM `X` and returns the equivalent non-flag-containing FSM.

Complement `~X`

Returns the complement of language `X`, in which case the expression is equivalent to `?* - X`. The operation is not well defined for transducers. The Unicode equivalent `¬X` may also be used.

Containment operators `$X`, `$.X`, `$?X`

The operator `$X` denotes the language that contains a substring drawn from the language `X`. Equivalent to `[?* X ?*]`. The operator `$.X` denotes the language that contains exactly one substring drawn from the language `X`, while `$?X` denotes the language that contains at most one substring from `X`.

Ignore `X/Y`

Denotes the language where instances of `Y` are arbitrarily interspersed in the language `X`. Not well-defined for transducers.

Ignore inside `X./.Y`

Denotes the language where instances of `Y` are arbitrarily interspersed in the language `X`, except that the first symbol and last symbol belong to `X`. Not well-defined for transducers.

Quotients `X///Y` (right), `X\\\Y` (left)

The operation `X///Y` is defined as:

$$\{w \mid \exists x((x \in Y) \wedge (wx \in X)\}$$

Informally, this is the set of prefixes one can add to `Y` to get strings from `X`.
The operation `X\\\Y` is defined as:

$$\{w \mid \exists x((x \in X) \wedge (xw \in Y)\}$$

Informally: the set of suffixes one can add to strings in `X` to get strings from `Y`.

Precedes, follows `X<Y`, `X>Y`

Denotes the languages where every string from `X` precedes every string from `Y` (<), or, where every string from `X` follows every string from `Y` (>). The precedence need not be immediate.

Concatenation `X Y`

The language or relation `X` concatenated with `Y`. The operator is not overtly signaled by spacing, etc., and two adjacent regular expressions will be concatenated regardless of whitespace when found at the level of precedence of the concatenation operator by the regular expression parser.

Union `X|Y`

The union of languages or relations `X` and `Y`. Associative. Unicode equivalents are: $X \cup Y$ and $X \vee Y$.

Intersection `X & Y`

The intersection of languages `X` and `Y`. Associative. Unicode equivalents are: $X \cap Y$ and $X \wedge Y$. For transducers, this denotes the intersection of the paths of `X` and `Y` which may or may not be equivalent to the intersection of the relations `X` and `Y`. Regular relations are in the general case not closed under intersection operation.

Set subtraction `X - Y`

For automata, the set of words from `X` minus the words from `Y`. Equivalent to `[X & ~Y]`. For transducers, this represents the subtraction of paths in `Y` from `X`, which may or may not be equivalent to the subtraction of relations `X` and `Y`. Regular relations are in the general case not closed under subtraction operation.

Priority unions `X .P. Y`, `X .p. Y`

The "upper-side priority union" `X .P. Y` denotes the union of relations `X` and `Y`, with relations in `Y` discarded if a relation in `X` have the same input (domain). Equivalent to `[X | [~[X.u] .o. Y]]`. The lower-side is similar, except a relation in `X` has precedence over a relation in `Y` based on the range, not the domain.

Context restriction `X => L`$_1$ `_ R`$_1$`, ..., L`$_n$ `_ R`$_n$

The language where every instance of a string from `X` is surrounded by string from some pair `L`$_i$ and `R`$_i$ on the left and right, respectively.

Shuffle `X <> Y`

The shuffle (or asynchronous) product of `X` and `Y`, i.e. the set of words formed by any method of 'shuffling' the contents of `X` with `Y`. The shuffle is not perfect.

Composition `X .o. Y`

The composition of relation `X` with `Y`. The unicode equivalent $X \circ Y$ may be used.

Lenient composition `X .O. Y`

The composition of `X` with `Y`. For those relations where strings in the domain of `Y` does not include some possible string from the range of `X`, the relation `X` is not composed with `Y`. Equivalent to `[X .o. Y] .P. Y`.

Replacement operators

All replacement rule operators follow the same template:

$\mathtt{X}_1$ **OP** $\mathtt{Y}_1$,..., $\mathtt{X}_n$ **OP** $\mathtt{Y}_n$ **DIR**$\mathtt{L}_1$ _ $\mathtt{R}_1$,..., $\mathtt{L}_n$ _ $\mathtt{R}_n$

where **OP** is one of:

`->`	Unconditional replacement
`<-`	Unconditional inverse replacement
`<->`	Unconditional replacement and inverse replacement
`(->)`	Optional replacement
`(<-)`	Optional inverse replacement
`(<->)`	Optional replacement and inverse replacement
`@->`	Left-to-right longest-match replacement
`@>`	Left-to-right shortest-match replacement
`<-@`	Left-to-right longest-match inverse replacement
`<@`	Left-to-right shortest-match inverse replacement
`(@->)`	Optional left-to-right longest-match replacement
`(@>)`	Optional left-to-right shortest-match replacement
`(<-@)`	Optional left-to-right longest-match inverse replacement
`(<@)`	Optional left-to-right shortest-match replacement

The directionality constraint **DIR** can be one of:

`||`	Left & right contexts must hold on upper side
`\\`	Left context holds on upper side, right context holds on lower side
`//`	Left context holds on lower side, right context holds on upper side
`\/`	Left & right contexts must hold on lower side

Additionally, the special modifier `[.X.]` may be used on the left-hand side of a rule. This signifies that, in case the language `X` contains the empty string, replacement is constrained to a *maximum* of one insertion for each legitimate location. For example:

```
foma[0]: regex [.(a).] -> b
```

Predicate logic quantifiers $(\forall x), (\exists x)$

The universal quantifier $(\forall x)$ and the existential quantifier $(\forall x)$ declares a variable name x as bound to a quantifier in subsequent statements. Inside a logical sentence the optionality operator $()$ is suspended and parentheses indicate grouping, e.g.:

```
foma[0]: regex (∀x)((x ∈ a b) → (S(c,x) ∨ S(x,d)));
```

Predicate logic connectives $\neg, \rightarrow, \vee, \wedge$

The connectives have the traditional meaning and are equivalent to their regular expression counterparts. The connective $X \rightarrow Y$ is logically equivalent to $\neg X \vee Y$.

Predicate logic predicates $S(t_1, t_2), x \in L, x = y, x \succ y, x \prec y$

The successor-of predicate $S(t_1, t_2)$ is true for all strings where t_1 is immediately followed by t_2. The predicate $x \in L$ is true for all strings where a bound variable x refers to a substring which is member of the language L. The predicate $x = y$ is true for all strings where the bound variables x and y share both the same beginning and ending positions. The predicates $x \succ y$ and $x \prec y$ are true for those strings where the quantified variable x follows resp. precedes (not necessarily immediately) y.

Built-in functions

There are a number of built-in functions that may be used in regular expressions. These all begin with an underscore and are as follows:

- `_isunambiguous(T)`: returns ϵ if T is unambiguous, else $\emptyset$
- `_isidentity(T)`: returns ϵ if T is an identity relation, else $\emptyset$
- `_isfunctional(T)`: returns ϵ if T is a function, else $\emptyset$
- `_notid(T)`: returns all the strings in the domain of T which are not in an identity relationship
- `_loweruniq(T)`: determinized the output side of T using arbitrary symbols
- `_allfinal(T)`: makes all states in T final
- `_unambpart(T)`: extracts the unambiguous paths from T
- `_ambpart(T)`: extracts the ambiguous paths from T
- `_ambdom(T)`: extracts from the domain of T all strings which are unambiguously mapped in T
- `_eq(T,L1,L2)`: removes from the output side of T all strings where substrings occurring between different instances $L1$ and $L2$ are unequal in content

A.12. Operator precedence

Unicode	ASCII
[ ] ( )	[ ] ( )
∀ ∃	
\ ‘	\ ‘
:	:
+ ∗	+ ∗
ˆ<n ˆ>n ˆ{m,n}	ˆ<n ˆ>n ˆ{m,n}
$_1$ $_2$ $^{-1}$	.1 .2 .u .l .i
.f	.f
¬ $ $. $?	˜ $ $. $?
/ ./. /// \\\ /\/	/ ./. /// \\\ /\/
(concatenation)	(concatenation)
∈ ∉ = ≠	
≻ ≺	> <
∨ ∪ ∧ ∩ - .P. .p.	| & − .P. .p.
=> -> (->) @->	=> -> (->) @->
‖	<>
× ∘ .O.	.x. .o. .O.

A.13. Compiler variables

Global variables control the behavior of the foma compiler. These may be modified by the command

```
set [name of variable] [value]
```

The following is a list of variables availble and their possible values.

flag-is-epsilon (0|1)

Controls whether flag diacritic symbols are treated as epsilon symbols in composition. Doing so makes it possible to write replacement rules that ignore the presence of flag symbols. To function correctly it requires that two different flags never occur as a pair on a transition, e.g. `@U.A.ON@:@U.A.OFF@`.

Default value: **0**

minimal (0|1)

Controls whether networks are minimized during and after construction. Turning the flag off will most likely slow down operations dramatically.
Default value: **1**

obey-flags (0|1)

Controls whether the application algorithm (apply down/up) checks for and disallows paths where flags are in conflict.
Default value: **1**

print-pairs (0|1)

Controls if apply down/up prints both sides of the strings.
Default value: **0**

print-sigma (0|1)

Controls if the alphabet is printed together with other network information in “print net”/“net”
Default value: **1**

print-space (0|1)

Controls if spaces are printed between symbols when printing or applying words.
Default value: **0**

quit-on-fail (0|1)

When `quit-on-fail` is set, *foma* quits immediately upon encountering an error in a script launched with `-l`.
Default value: **1**

show-flags (0|1)

Controls whether flag diacritic symbols are printed when printing or applying words.
Default value: **0**

hopcroft-min (0|1)

Controls whether to use Hopcroft’s $O(n\ log\ n)$ algorithm for minimization. If the variable is 0, Brzozowski’s minimization is used instead, i.e. X is minimized by

```
determinize(reverse(determinize(reverse(X))))
```

Hopcroft's algorithm is far superior for the majority of cases.
Default value: **1**

compose-tristate (0|1)

Controls the strategy by which the composition algorithm avoids creating multiple paths for the same input/output pairings. Tristate composition is similar to the filter transducer descibed in Mohri, Pereira and Riley(1997). The default is to use a simpler 'bistate' filtering, which is faster, also avoids multiple paths, and often creates smaller transducers than the tristate approach. However, the tristate algorithm may produce better alignments in some cases, such as:

```
a:0 b:0 c:0 .o. 0:d 0:e 0:f
```

for which the bistate algorithm fails to achieve optimal alignment.
Default value: **0**

med-limit (0-MAXINT)

Sets the limit for how many words should be printed by the minimum edit distance lookup command `apply med`.
Default value: **3**

med-cutoff (0-MAXINT)

Sets the limit for how when to stop searching for minimum edit distance solutions when doing `apply med`.
Default value: **15**

A.14. Additional interface commands

```
ambiguous upper                   returns the set of input words which
                                  produce multiple paths in a transducer
apply up <string>                 apply <string> up to the top network on
                                  stack
apply down <string>               apply <string> down to the top network
                                  on stack
apply med <string>                find approximate matches to string in
                                  top network by minimum edit distance
apply up                          enter apply up mode (Ctrl-D exits)
apply down                        enter apply down mode (Ctrl-D exits)
apply med                         enter apply med mode (Ctrl-D exits)
apropos <string>                  search help for <string>
clear stack                       clears the stack
```

`compact sigma`	removes redundant symbols from FSM
`complete net`	completes the FSM
`define <name> <r.e.>`	define a network
`define <fname>(<v1,..,vn>) <r.e.>`	define function
`determinize net`	determinizes top FSM on stack
`echo <string>`	echo a string
`eliminate flag <name>`	eliminate flag <name> diacritics from the top network
`eliminate flags`	eliminate all flag diacritics from the top network
`export cmatrix (filename)`	export the confusion matrix as an AT&T transducer
`extract ambiguous`	extracts the ambiguous paths of a transducer
`extract unambiguous`	extracts the unambiguous paths of a transducer
`help license`	prints license
`help warranty`	prints warranty information
`label net`	extracts all attested symbol pairs from FSM
`load stack <filename>`	Loads networks and pushes them on the stack
`load defined <filename>`	Restores defined networks from file
`minimize net`	minimizes top FSM
`name net <string>`	names top FSM
`pop stack`	remove top FSM from stack
`print cmatrix`	prints the confusion matrix associated with the top network in tabular format
`print defined`	prints defined symbols and functions
`print dot (>filename)`	prints top FSM in Graphviz dot format
`print lower-words`	prints words on the lower-side of top FSM
`print name`	prints the name of the top FSM
`print net`	prints all information about top FSM
`print random-lower`	prints random words from lower side
`print random-upper`	prints random words from upper side
`print random-words`	prints random words from top FSM
`print sigma`	prints the alphabet of the top FSM
`print size`	prints size information about top FSM
`print shortest-string`	prints the shortest string of the top FSM
`print shortest-string-size`	prints length of shortest string
`prune net`	makes top network coaccessible
`push (defined) <name>`	adds a defined FSM to top of stack
`quit`	exit foma
`read att <filename>`	read a file in AT&T FSM format and add to top of stack
`read cmatrix <filename>`	read a confusion matrix and associate it with the network on top of the stack

read prolog <filename>	reads prolog format file
read lexc <filename>	read and compile lexc format file
rotate stack	rotates stack
save defined <filename>	save all defined networks to binary file
save stack <filename>	save stack to binary file
set <variable> <ON|OFF>	sets a global variable (see show variables)
show variables	prints all variable/value pairs
sigma net	Extracts the alphabet and creates a FSM that accepts all single symbols in it
source <file>	read and compile script file
sort net	sorts arcs lexicographically on top FSM
substitute symbol X for Y	substitutes all occurrences of Y in an arc with X
system <cmd>	execute a system command
test unambiguous	test if top FST is unambiguous
test equivalent	test if the top two FSMs are equivalent
test functional	test if the top FST is functional
test identity	test if top FST represents identity relations only
test lower-universal	test if lower side is ?*
test upper-universal	test if upper side is ?*
test non-null	test if top machine is not the empty language
test null	test if top machine is the empty language (∅)
test star-free	test if top FSM is star-free
turn stack	turns stack upside down
twosided flag-diacritics	changes flags to always be identity pairs
undefine <name>	remove <name> from defined networks
upper-side net	upper projection of top FSM
view net	display top network (if supported)
write prolog (> filename)	writes top network to prolog format file/stdout
write att (> <filename>)	writes top network to AT&T format file/stdout

B. Selected example scripts

B.1. Fundamental operations

B.1.1. Filtered cross products

The following script illustrates the aligned cross product in figure 3.4.

```
# Two ways of performing a cross-product:
# naive (Filtered), and aligned (NonFiltered)

define NonFiltered [a .x. [b c]*];
define Filtered [a .x. [b c]*] & [?:?]* [[?:0]* | [0:?]*];

regex NonFiltered;
print net
regex Filtered;
print net
```

B.1.1.1. Output

```
defined NonFiltered: 257 bytes. 3 states, 4 arcs, Cyclic.
defined Filtered: 336 bytes. 4 states, 4 arcs, Cyclic.
257 bytes. 3 states, 4 arcs, Cyclic.
Sigma: a b c
Size: 3.
Net: 6B8B4567
Flags: deterministic pruned minimized epsilon_free
Arity: 2
Ss0:   <a:b> -> s1, <a:0> -> fs2.
fs2:   <0:b> -> s1.
s1:    <0:c> -> fs2.
336 bytes. 4 states, 4 arcs, Cyclic.
Sigma: ? @ a b c
Size: 3.
Net: 327B23C6
Flags: deterministic pruned minimized epsilon_free
Arity: 2
Ss0:   <a:b> -> s2, <a:0> -> fs1.
fs1:   (no arcs).
s2:    <0:c> -> fs3.
fs3:   <0:b> -> s2.
```

B.1.2. Composition strategies

The following script performs the composition of $T1$ and $T2$ with both the bimode and trimode strategies

```
define T1 [a:0 b:0];
define T2 [0:c 0:d];
regex T1 .o. T2;
print net
set compose-tristate ON
regex T1 .o. T2;
print net
```

B.1.2.1. Output

```
defined T1: 227 bytes. 3 states, 2 arcs, 1 path.
defined T2: 227 bytes. 3 states, 2 arcs, 1 path.
287 bytes. 5 states, 4 arcs, 1 path.
Sigma: a b c d
Size: 4.
Net: 6B8B4567
Flags: deterministic pruned minimized epsilon_free loop_free
Arity: 2
Ss0: <a:0> -> s1.
s1: <b:0> -> s2.
s2: <0:c> -> s3.
s3: <0:d> -> fs4.
fs4: (no arcs).
variable compose-tristate = ON
255 bytes. 3 states, 2 arcs, 1 path.
Sigma: a b c d
Size: 4.
Net: 327B23C6
Flags: deterministic pruned minimized epsilon_free loop_free
Arity: 2
Ss0: <a:c> -> s1.
s1: <b:d> -> fs2.
fs2: (no arcs).
```

B.1.3. The shortest string problem

The following script illustrates the extraction of the set of shortest strings from a language L, as described in section 3.8.2.5.

```
# Illustration of the shortest string extraction by definition
# of a function that extracts the set of shortest strings

define ShortestString(X) [X - ?+ [X .o. [?:?]*].l];
define L [a b c (d) (e) (f) (g)];

regex ShortestString(L);
print words
```

B.1.3.1. Output

```
defined ShortestString(@1)
defined L: 473 bytes. 8 states, 13 arcs, 16 paths.
376 bytes. 4 states, 3 arcs, 1 path.
abc
```

B.1.4. Edit distance

A simple example script to illustrate how to construct a machine L' from a lexicon L, where L' contains all the words exactly one edit distance away from words in L (see section 3.8.2.6).

```
# Calculate all the words exactly one edit distance away
# from words in L

define L {cat}|{dog};
define ED1(X) [X .o. ?* [?:?-?|?:0|0:?] ?*].l;
regex ED1(L);
print words
```

B.1.4.1. Output

```
defined L: 347 bytes. 6 states, 6 arcs, 2 paths.
defined ED1(@1)
1.6 kB. 24 states, 82 arcs, 92 paths.
ccat
cct
c@at
c@t
ca
...
dcg
dcog
d@g
d@og
dag
...
```

B.1.5. Longest common substring and subsequence

The following script illustrates the calculation of the longest common substrings and subsequences as descibed in section 3.8.2.7.

```
# The following script illustrates the calculation of
# the longest common substrings and subsequences
# by declaring functions for that purpose

# The example solves both problems for
# s = abcaa, t = dbcadaa
```

```
define Substring(X) [X .o. ?:0* ?* ?:0*].l;
define Subsequence(X) [X .o. [?|?:0]*].l;
define Longest(X) X - [[X .o. ?:a* ?:0+].l .o. a:?*].l;
define LCSubstr(X,Y) [Longest(Substring(X) & Substring(Y))];
define LCSubseq(X,Y) [Longest(Subsequence(X) & Subsequence(Y))];

define S [a b c a a];
define T [d b c a d a a];

regex LCSubstr(S,T);
print words
regex LCSubseq(S,T);
print words
```

B.1.5.1. Output

```
defined Substring(@1)
defined Subsequence(@1)
defined Longest(@1)
defined LCSubstr(@2)
defined LCSubseq(@2)
defined S: 289 bytes. 6 states, 5 arcs, 1 path.
defined T: 335 bytes. 8 states, 7 arcs, 1 path.
334 bytes. 4 states, 3 arcs, 1 path.
bca
350 bytes. 5 states, 4 arcs, 1 path.
bcaa
```

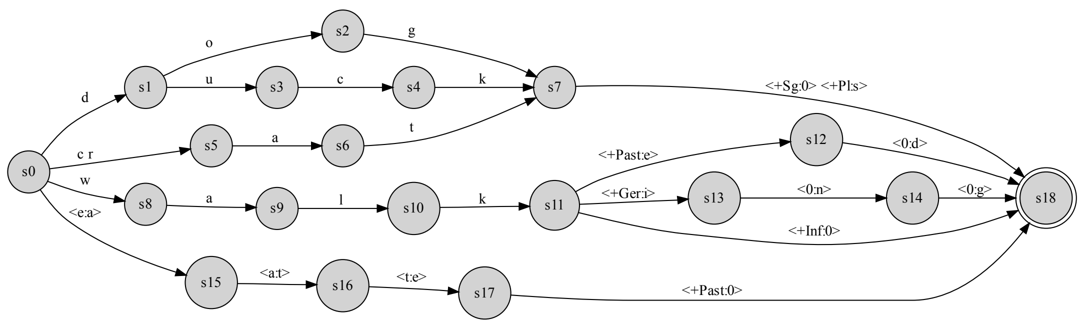
B.1.6. Lexicon compilation

The following lexicon script illustrates the contents of figure 3.11 as well as the subsequent determinized and minimized transducer.

```
Multichar_Symbols %+Inf %+Past %+Ger %+Pl %+Sg

Lexicon Root
Noun;
Verb;

Lexicon Noun
cat  N;
dog  N;
duck N;
rat  N;

Lexicon Verb
walk         V;
eat%+Past:ate #;
```

```
Lexicon V
%+Inf:0   #;
%+Past:ed #;
%+Ger:ing #;

Lexicon N
%+Pl:s #;
%+Sg:0 #;
```

B.1.6.1. Output



B.2. Predicate Logic

B.2.1. Basic examples

The following script compiles the predicate logic examples given in section 6.3. Also, for the regular expression given in 6.1, it is compared against example (vii) in section 6.3, using $(ab \cup ba)$ as L.

```
# Logic examples

define L [a b | b a];
define i   (∃x)(x ∈ a b);
define ii  (∃x)(x ∈ a b) ∧ (∃y)(y ∈ c d);
define iii (∀x)(x ∈ a b);
define iv  (∀x)((x ∈ a b) → S(x,d));
define v   (∀x)((x ∈ a b) → (S(c,x) ∨ S(x,d)));
define vi  (∀x)((x ∈ a b) → S(x, ?⋆ d));

define vii (∃x)(x ∈ L ∧ (∀y)((y ∈ L) → (x = y)));
# vii should be equivalent to the regex
define viiregex [?⋆ L ?⋆] - [?⋆ [[?+ L ?⋆ & L ?⋆] | [L ?+ & L]] ?⋆];

regex vii;
regex viiregex;
test equivalent
```

B.2.1.1. Output

```
defined L: 259 bytes. 4 states, 4 arcs, 2 paths.
defined i: 355 bytes. 3 states, 9 arcs, Cyclic.
defined ii: 879 bytes. 8 states, 40 arcs, Cyclic.
defined iii: 136 bytes. 1 states, 0 arcs, 0 paths.
defined iv: 369 bytes. 3 states, 9 arcs, Cyclic.
defined v: 495 bytes. 4 states, 16 arcs, Cyclic.
defined vi: 417 bytes. 3 states, 12 arcs, Cyclic.
defined vii: 467 bytes. 6 states, 16 arcs, Cyclic.
defined viiregex: 467 bytes. 6 states, 16 arcs, Cyclic.
467 bytes. 6 states, 16 arcs, Cyclic.
467 bytes. 6 states, 16 arcs, Cyclic.
1 (1 = TRUE, 0 = FALSE)
```

B.2.2. Context restriction

The following example illustrates two methods of compiling the example context restriction statement (6.13): through the method given in Yli-Jyrä and Koskenniemi (2004) and through predicate logic presented in chapter 6.

```
# Compiling x => a _ b , c _ d

# The logic method
regex (∀y)((y ∈ x) → (S(a,y) ∧ S(y,b)) ∨ (S(c,y) ∧ S(y,d)));
# Yli-jyrä & Koskenniemi method
regex ˜`[[\◇⋆ ◇ x ◇ \◇⋆ - [\◇⋆ a ◇ \◇⋆ ◇ b \◇⋆ | \◇⋆ c ◇ \◇⋆ ◇ d \◇⋆]],◇,0];
test equivalent
```

B.2.2.1. Output

```
557 bytes. 5 states, 19 arcs, Cyclic.
557 bytes. 5 states, 19 arcs, Cyclic.
1 (1 = TRUE, 0 = FALSE)
```

B.3. Reduplication

B.3.1. Simple duplication of a lexicon

The following script illustrates the use of the EQ operator to create from a closed lexicon, a lexicon that contains all reduplicated forms as well.

```
# Reduplicating a lexicon with _eq()

# We first define a lexicon
define Lex {cat}|{dog}|{horse}|{mouse};

# Then we define the language that contains one or
# two words from the lexicon (with a hyphen in the middle)
# The two words may be different at this point
```

```
define BracketedLexicon [%< Lex %> (%- %< Lex %>)];

# Then we constrain the two bracketed words to be equal with _eq()
# ruling out invalid reduplications
define Duplicates _eq(BracketedLexicon, %<, %>);

# And remove the auxiliary brackets
regex [Duplicates .o. %<|%> -> 0].l ;

print words
```

B.3.1.1. Output

```
defined Lex: 559 bytes. 12 states, 14 arcs, 4 paths.
defined BracketedLexicon: 905 bytes. 28 states, 33 arcs, 20 paths.
defined Duplicates: 1.1 kB. 42 states, 44 arcs, 8 paths.
984 bytes. 32 states, 34 arcs, 8 paths.
cat
cat-cat
dog
dog-dog
horse
horse-horse
mouse
mouse-mouse
```

B.3.2. Total reduplication

The following script illustrates how to capture total reduplication against a lexicon using EQ as described in (4.14).

```
define N {orang}|{buku};
define Lexicon N %+Noun %+Sg | 0:%< N 0:%> 0:%< 0:N 0:%> %+Noun %+Pl;
define RemoveDiacritics %<|%>|%+Noun|%+Sg|%+Pl -> 0;
define Grammar _eq(Lexicon, %<,%>) .o. RemoveDiacritics;
regex Grammar;
echo "analysis of buku"
apply up buku
echo analysis of bukubuku
apply up bukubuku
```

B.3.2.1. Output

```
defined N: 423 bytes. 9 states, 9 arcs, 2 paths.
defined Lexicon: 917 bytes. 32 states, 35 arcs, 6 paths.
defined RemoveDiacritics: 388 bytes. 1 states, 6 arcs, Cyclic.
defined Grammar: 788 bytes. 21 states, 23 arcs, 4 paths.
788 bytes. 21 states, 23 arcs, 4 paths.
"analysis of buku"
buku+Noun+Sg
analysis of bukubuku
buku+Noun+Pl
```

B.3.3. Warlpiri reduplication

The following script captures the Warlpiri mini-grammar outlined in section 4.6.3.1. Here we have a small lexicon of words which can possibly undergo a reduplication process that follows a $CV(C)(C)V$ template.

```
# Warlpiri reduplication example implemented with the _eq() operator
# Examples come from Nash(1980), pp. 142-144 and Sproat (1992), p. 58

# Warlpiri reduplication operates on the base form of a word and copies
# a "prosodic foot" or "reduplication skeleton" from the base.
# The prosodic foot is of the shape C V (C) (C) V.
# This reduplication process yields, for example:

# Base form    Reduplicated form
# --------     -----------------
# pakarni      pakapakarni
# pangurnu     pangupangurnu
# wantimi      wantiwantimi

# The script has a base lexicon.
# All words in the lexicon may be followed by the tag +Redup.

# We first mark the prosodic template with < ... > and,
# if the tag +Redup is present, prefix a < [C|V]* > sequence
# (i.e., anything surrounded by < and > ) after which we apply
# _eq() which filters out all substrings ... < X > ... < Y > ...
# where X is not equal to Y in content

# The idea is to get strings in the composition sequence such as:

# 1. pangurnu+Redup          <- Lexicon
# 2. <pangu>rnu+Redup        <- enclose initial prosodic foot in brackets
# 3. <...><pangu>rnu+Redup   <- insert a prefix of "anything" in brackets
#                               if +Redup is present
# 4. <pangu><pangu>rnu       <- after _eq()
# 5. pangupangurnu           <- after auxiliary symbol removal

define C p|{ng}|{rn}|{rl}|k|j|w|n|t|m;
define V a|e|i|o|u;
define Lexicon {pangurnu}|{tiirlparnkaja}|{wantimi}|{pakarni};
define Morphology Lexicon ("+Redup");
define MarkTemplate C V (C) (C) V -> "<" ... ">" || .#. _ ;
define InsertPrefix [..] ->  "<" [C|V]* ">" || .#. _ ?* "<" ?* "+Redup";
define RemoveTags "+Redup" -> 0;
define RemoveBrackets "<"|">" -> 0;
define PreEq Morphology .o. MarkTemplate .o. InsertPrefix .o. RemoveTags;
regex _eq(PreEq, "<" , ">") .o. RemoveBrackets;

echo analysis of "pakarni"
```

```
apply up pakarni
echo analysis of "pakapakarni"
apply up pakapakarni
```

B.3.3.1. Output

```
defined C: 483 bytes. 4 states, 11 arcs, 10 paths.
defined V: 317 bytes. 2 states, 5 arcs, 5 paths.
defined Lexicon: 861 bytes. 30 states, 32 arcs, 4 paths.
defined Morphology: 896 bytes. 31 states, 33 arcs, 8 paths.
defined MarkTemplate: 4.6 kB. 24 states, 269 arcs, Cyclic.
defined InsertPrefix: 2.7 kB. 9 states, 145 arcs, Cyclic.
defined RemoveTags: 265 bytes. 1 states, 2 arcs, Cyclic.
defined RemoveBrackets: 290 bytes. 1 states, 3 arcs, Cyclic.
defined PreEq: 2.4 kB. 73 states, 120 arcs, Cyclic.
1.7 kB. 76 states, 82 arcs, 8 paths.
analysis of "pakarni"
pakarni
analysis of "pakapakarni"
pakarni+Redup
```

B.4. Multitape automata

B.4.1. Declaring tape contents individually

The script below illustrates and defines some of the operations in section 7.6.1, and exemplifies formulas (7.2) and (7.8) by creating a 4-tape machine from individually declared tape contents.

```
# We create a 4-tape simulation with
# low-level operators and model the configuration

# a b c   or   a b c
# d            d
# e f          e f
# g h i        g h j

# by declaring each of the tape contents separately
# and aligning blanks to the right.

define AlignRightblanks(L) [L .o. ?* 0:□*].l;

define Tape14(L) [L .o. [? 0:? 0:? 0:?]*].l;
define Tape24(L) [L .o. [0:? ? 0:? 0:?]*].l;
define Tape34(L) [L .o. [0:? 0:? ? 0:?]*].l;
define Tape44(L) [L .o. [0:? 0:? 0:? ?]*].l;

regex Tape14(AlignRightblanks(a b c)) &
      Tape24(AlignRightblanks(d))     &
      Tape34(AlignRightblanks(e f))   &
      Tape44(AlignRightblanks(g h i | g h j)) &
      ~[?^4* □ □ □ □ ?*] ;

print words
```

B.4.1.1. Output

```
defined AlignRightblanks(@1)
defined Tape14(@1)
defined Tape24(@1)
defined Tape34(@1)
defined Tape44(@1)
594 bytes. 13 states, 13 arcs, 2 paths.
adegb□fhc□□i
adegb□fhc□□j
```

B.5. Transducer properties

B.5.1. Tests for functionality and ambiguity

The following is an illustration of various algorithms in chapter 5 such as tests for identity (algorithm 5.1), functionality (5.6), ambiguity (5.10), extraction of the ambiguous domain (5.11), and splicing a transducer into its ambiguous and unambiguous components (5.16).

```
# Illustrations of test of identity, functionality,
# ambiguity, and extraction of the ambiguous domain
# as well as splicing a transducer into its unambiguous
# and ambiguous parts

# 1. An identity transducer over "a b c" despite non-identity alignments
regex [a:0 b:a c:0 0:b 0:c]*;
test identity

# 2. A functional, but ambiguous transducer
regex [a:0 0:b | a:b];
test functional
test unambiguous

# 3. A non-functional, ambiguous transducer
regex a b -> x \/ a b _ a ;
test functional
test unambiguous

# 4. A functional, non-ambiguous transducer
regex a b -> x || a b _ a;
test functional
test unambiguous
```

```
# 5. Extract the ambiguous domain from 3.
regex _ambdom(a b -> x \/ a b _ a);
# print the shortest string in the domain that has
# an ambiguous path
print shortest-string

# 6. Splice 3. into an (a) unambiguous and (b) ambiguous part
define UNAMB3 _unambpart(a b -> x \/ a b _ a);
define AMB3   _ambpart(a b -> x \/ a b _ a);

# 7. the union of 6(a) and 6(b) should equal 3.

regex UNAMB3 | AMB3;
regex a b -> x \/ a b _ a ;
test equivalent
```

B.5.1.1. Output

```
273 bytes. 5 states, 5 arcs, Cyclic.
1 (1 = TRUE, 0 = FALSE)
243 bytes. 3 states, 3 arcs, 2 paths.
1 (1 = TRUE, 0 = FALSE)
0 (1 = TRUE, 0 = FALSE)
624 bytes. 7 states, 23 arcs, Cyclic.
0 (1 = TRUE, 0 = FALSE)
0 (1 = TRUE, 0 = FALSE)
624 bytes. 7 states, 23 arcs, Cyclic.
1 (1 = TRUE, 0 = FALSE)
1 (1 = TRUE, 0 = FALSE)
564 bytes. 8 states, 24 arcs, Cyclic.
abababa
defined UNAMB3: 608 bytes. 7 states, 22 arcs, Cyclic.
defined AMB3: 1.1 kB. 16 states, 55 arcs, Cyclic.
624 bytes. 7 states, 23 arcs, Cyclic.
624 bytes. 7 states, 23 arcs, Cyclic.
1 (1 = TRUE, 0 = FALSE)
```

REFERENCES

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice Hall.

Alegria, I., Etxeberria, I., Hulden, M., and Maritxalar, M. (2009). Porting Basque morphological grammars to foma, an open-source tool. In *FSMNLP 2009*.

Almeida, M., Moreira, N., and Reis, R. (2007). On the performance of automata minimization algorithms. Technical report, Universidade do Porto.

Antimirov, V. M. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–219.

Antworth, E. (1991). *PC-KIMMO: a two-level processor for morphological analysis*. Number 16 in Occasional Publications in Academic Computing. Summer Institute of Linguistics.

Bassino, F., David, J., and Nicaud, C. (2009). On the average complexity of Moore's state minimization algorithm. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, volume 3 of *Leibniz International Proceedings in Informatics*, pages 123–134.

Béal, M.-P., Carton, O., Prieur, C., and Sakarovitch, J. (2000). Squaring transducers: an efficient procedure for deciding functionality and sequentiality of transducers. In *LNCS 1776*. Springer.

Béal, M.-P. and Crochemore, M. (2008). Minimizing incomplete automata. In *Finite-State Methods and Natural Language Processing (FSMNLP'08)*.

Beesley, K. R. (1998). Consonant spreading in Arabic stems. In *ACL 98 Proceedings*.

Beesley, K. R. and Karttunen, L. (2000). Finite-state non-concatenative morphotactics. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*.

Beesley, K. R. and Karttunen, L. (2003). *Finite State Morphology*. CSLI Publications, Stanford, CA.

Bergroth, L., Hakonen, H., and Raita, T. (2000). A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval SPIRE'00*, pages 39–48.

Berstel, J. and Carton, O. (2005). On the complexity of Hopcroft's state minimization algorithm. *Lecture Notes in Computer Science*, 3317:35–44.

Bird, M. (1973). The equivalence problem for deterministic two-tape automata. *Journal of Computer and System Sciences*, 7:218–236.

Blattner, M. and Head, T. (1977). Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):328–353.

Brzozowski, J. A. (1963). Canonical regular expressions and minimal state graphs for definite events. In *Proceedings of the Symposium on Mathematical Theory of Automata, New York, NY, April 24-26, 1962*, volume 12 of *MRI Symposia Series*, pages 529–561, Brooklyn, NY. Polytechnic Press of the Polytechnic Institute of Brooklyn.

Büchi, J. R. (1960). Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 6:66–92.

Buckwalter, T. (2004). Arabic Morphological Analyzer 2.0. *Linguistics Data Consortium (LDC).*

Câmpeanu, C., Salomaa, K., and Yu, S. (2003). A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(6):1007–1017.

Castiglione, G., Restivo, A., and Sciortino, M. (2008). Hopcroft's algorithm and cyclic automata. *Lecture Notes in Computer Science*, 5196:172–183.

Champarnaud, J.-M., Khorsi, A., and T., P. (2002). Split and join for minimizing: Brzozowski's algorithm. In *Proceedings of PSC 2002 (Prague Stringology Conference).*

Choffrut, C. (1978). *Contributions à l'étude de quelques familles remarquables de fonctions rationnelles*. PhD thesis, Université Paris 7, Paris, France.

Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. Harper & Row.

Cohen-Sygal, Y. and Wintner, S. (2006). Finite-state registered automata for non-concatenative morphology. *Computational Linguistics*, 32(1):49–82.

Culik II, K. (1978). Some decidability results about regular and push down translations. Research Report CS-78-09, University of Waterloo, Waterloo, Ontario, Canada.

Culy, C. (1985). The complexity of the vocabulary of Bambara. *Linguistics and Philosophy*, 8(3):345–351.

Daciuk, J., Mihov, S., Watson, B. W., and Watson, R. E. (1998). Incremental construction of minimal acyclic finite-state automata. In *FSMNLP'98: International Workshop on Finite State Methods in Natural Language Processing*.

Elgot, C. C. (1961). Decision problems of finite automata and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–51.

Gerdmann, D. and van Noord, G. (1999). Transducers from rewrite rules with backreferences. In *Proceedings of EACL 1999*.

Gerdmann, D. and van Noord, G. (2000). Approximation and exactness in finite state optimality theory. In *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*.

Ghomeshi, J., Jackendoff, R., Rosen, N., and Russell, K. (2004). Contrastive focus reduplication in English (the salad-salad paper). *Natural Language & Linguistic Theory*, 22(2):307–357.

Glushkov, V. M. (1961). The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53.

Gries, D. (1972). Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–107.

Griffiths, T. V. (1968). The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *Journal of the Association for Computing Machinery*, 15(3):409–413.

Grimley-Evans, E., Kiraz, G. A., and Pulman, S. G. (1996). Compiling a partition-based two-level formalism. In *Proceedings of the 16th conference on Computational linguistics*, pages 454–459.

Harju, T. and Karhumäki, J. (1991). The equivalence problem of multitape finite automata. *Theoretical Computer Science*, 78(2):347–255.

Harris, Z. (1941). Linguistic structure of Hebrew. *Journal of the American Oriental Society*, 61(3):143–167.

Hopcroft, J. (1971). An n log n algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford University.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hulden, M. (2006). Finite-state syllabification. *Lecture Notes in Artificial Intelligence*, 4002:86–96.

Hulden, M. (2009a). Fast approximate string matching with finite automata. *Procesamiento del lenguaje natural*, 43:57–64.

Hulden, M. (2009b). Foma: a finite-state compiler and library. In *EACL 2009 Proceedings*, pages 29–32.

Hulden, M. (2009c). Parsing CFGs and PCFGs with a Chomsky-Schützenberger representation. *Proceedings of LTC 2009*, pages 95–99.

Hulden, M. (2009d). Regular expressions and predicate logic in finite-state language processing. In Piskorski, J., Watson, B., and Yli-Jyrä, A., editors, *Finite-State Methods and Natural Language Processing—Post-proceedings of the 7th International Workshop FSMNLP 2008*, volume 191 of *Frontiers in Artificial Intelligence and Applications*, pages 82–97. IOS Press.

Hulden, M. (2009e). Revisiting multi-tape automata for Semitic morphological analysis and generation. *Proceedings of the EACL 2009 Workshop on Computational Approaches to Semitic Languages*, pages 19–26.

Hulden, M. and Bischoff, S. T. (2007). A simple formalism for capturing order and co-occurrence in computational morphology. *Procesamiento del lenguaje natural*, 39:21–26.

Hulden, M. and Bischoff, S. T. (2008). Annotating reduplication in finite-state morphology. *Proceedings of FSMNLP 2008*, pages 165–169.

Ilie, L. and Yu, S. (2003). Follow automata. *Information and Computation*, 186(1):146–162.

Inkelas, S. and Zoll, C. (2005). *Reduplication: Doubling in Morphology*. Cambridge University Press.

Johnson, C. D. (1972). *Formal aspects of phonological description*. Mouton, The Hague.

Joshi, A. K. and Hopely, P. (1997). A parser from antiquity: An early application of finite state transducers to natural language parsing. In *Extended Finite State Models of Language*. Cambridge University Press.

Kager, R. (1999). *Optimality Theory*. Cambridge University Press.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, L. (1993). Finite-state lexicon compiler. Technical Report ISTL-NLTT2993-04-02, Xerox Palo Alto Research Center.

Karttunen, L. (1994). Constructing lexical transducers. In *COLING '94*.

Karttunen, L. (1996). Directed replacement. In *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 108–115.

Karttunen, L. (1997). The replace operator. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*. MIT Press.

Karttunen, L. (1998). The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*.

Karttunen, L. (2003). Computing with realizational morphology. *Lecture Notes in Computer Science*, 2588:205–216.

Karttunen, L., Koskenniemi, K., and Kaplan, R. M. (1987). A compiler for two-level phonological rules. In Dalrymple, M., Kaplan, R., Karttunen, L., Koskenniemi, K., Shaio, S., and Wescoat, M., editors, *Tools for Morphological Analysis*. CSLI Publications.

Kataja, L. and Koskenniemi, K. (1988). Finite-state description of Semitic morphology: A case study of ancient Akkadian. In *COLING '88*.

Kay, M. (1987). Nonconcatenative finite-state morphology. In *Proceedings of EACL 1987*.

Kempe, A. and Karttunen, L. (1996). Parallel replacement in finite state calculus. In *Proceedings of the 34th annual meeting of the Association for Computational Linguistics*.

Kenstowicz, M. and Kisseberth, C. (1979). *Generative Phonology*. Academic Press.

Kiparsky, P. (1986). *The Phonology of Reduplication. (Ms.)*. Stanford University.

Kiparsky, P. (2009). On the architecture of Pānini's grammar. *Lecture Notes in Artificial Intelligence*, 5402.

Kiraz, G. A. (1994). Multi-tape two-level morphology: A case study in Semitic non-linear morphology. In *COLING '94*.

Kiraz, G. A. (1997). Compiling regular formalisms with rule features into finite-state automata. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 329–336.

Kiraz, G. A. (2000). Multi-tiered nonlinear morphology using multitape finite automata: A case study on Syriac and Arabic. *Computational Linguistics*, 26(1):77–105.

Kiraz, G. A. and Grimley-Evans, E. (1998). Multi-tape automata for speech and language systems: A prolog implementation. *Lecture Notes in Computer Science*, 1436:87–103.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley.

Knuutila, T. (2001). Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250:333–363.

Koskenniemi, K. (1983). *Two-level morphology: A general computational model for word-form recognition and production*. University of Helsinki, Department of General Linguistics, Helsinki.

Kozen, D. C. (1997). *Automata and Computability*. Springer.

Langendoen, D. T. (1981). The generative capacity of word-formation components. *Linguistic Inquiry*, 12:320–322.

Langendoen, D. T. and Langsam, Y. (1984). On the design of finite transducers for parsing phrase-structure languages. *Mathematics of Language: Proceedings of a conference held at the University of Michigan, Ann Arbor, October 1984*, pages 191–236.

Laporte, E. (1997). Rational transductions for phonetic conversion and phonology. In *Finite-State Language Processin*. MIT Press.

Lavie, A., Itai, A., and Ornan, U. (1988). On the applicability of two level morphology to the inflection of Hebrew verbs. In *Proceedings of ALLC III*.

Leslie, T. (1995). Efficient approaches to subset construction. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada.

MacDonald, R. R. and Darjowidjojo, S. (2001). *A student's reference grammar of modern formal Indonesian*. Georgetown University Press.

Marantz, A. (1982). Re reduplication. *Linguistic Inquiry*, 13(3):435–482.

McCarthy, J. J. (1979). *Formal Problems in Semitic Phonology and Morphology*. PhD thesis, MIT, Cambridge, MA.

McCarthy, J. J. (1981). A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry*, 12(3):373–418.

McNaughton, R. and Papert, S. (1971). *Counter-free Automata*. MIT Press.

Mohanan, K. P. (1986). *The Theory of Lexical Phonology*. Reidel, Dordrecht, Holland.

Mohri, M. (1997a). Finite-state transducers in speech and language processing. *Computational Linguistics*, 23(2):269–311.

Mohri, M. (1997b). On the use of sequential transducers in natural language processing. *Finite-State Language Processing*, pages 355–382.

Mohri, M. and Nederhof, M.-J. (2000). Regular approximation of context-free grammars through transformations. In Junqua, J.-C. and van Noord, G., editors, *Robustness in Language and Speech Technology*, pages 251–261. Kluwer Academic Publishers.

Mohri, M., Pereira, F., and Riley, M. (1996). Weighted automata in text and speech processing. In *ECAI 96, 12th European Conference on Artificial Intelligence*.

Mohri, M. and Sproat, R. (1996). An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 231–238.

Moore, E. F. (1956). Gedanken-experiments on sequential machines. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, volume 2 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press.

Moravcsik, E. A. (1978). Reduplicative constructions. In Greenberg, J., editor, *Universals of Human Language*, volume 3, pages 297–334. Stanford University Press, Stanford, CA.

Myhill, J. (1957). Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson AFB, Ohio.

Nash, D. G. (1980). *Topics in Warlpiri Grammar*. PhD thesis, MIT.

Nederhof, M.-J. (2000). Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44.

Nerode, A. (1958). Linear automaton transformations. *Proceedings of the AMS*, 9:541–544.

Payne, D. L. (1981). *The phonology and morphology of Axininca Campa*. University of Texas at Arlington.

Post, E. L. (1946). A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268.

Rabin, M. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal*, 3(2):114–125.

Reichard, G. A. (1938). Coeur d'Alene. In Boaz, F., editor, *Handbook of American Indian Languages*, volume 3, pages 515–707. J. J. Augustin, New York.

Revuz, D. (1992). Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(2):181–189.

Ritchie, G. D., Russell, G. J., Black, A. W., and Pulman, S. G. (1991). *Computational morphology: practical mechanisms for the English lexicon*. MIT Press.

Roark, B. and Sproat, R. (2007). *Computational Approaches to Morphology and Syntax*. Oxford University Press.

Roche, E. and Schabes, Y. (1997). Introduction. *Finite-State Language Processing*.

Russell, K. (1997). Optimality theory and morphology. In Archangeli, D. and Langendoen, T. D., editors, *Optimality Theory: an Overview*, Explaining Linguistics, pages 102–133. Blackwell, Oxford.

Schützenberger, M. P. (1976). Sur les relations rationnelles entre monoïdes libres. *Theoretical Computer Science*, 3:243–259.

Schützenberger, M. P. (1977). Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4:47–57.

Selkirk, E. (1980). The role of prosodic categories in English word stress. *Linguistic Inquiry*, 11(3):563–605.

Sipser, M. (2006). *Introduction to The Theory of Computation*. Thomson, 2nd edition.

Skut, W., Ulrich, S., and Hammervold, K. (2003). A generic finite state compiler for tagging rules. *Machine Translation*, 18(3):239–250.

Sommer, B. (1981). The shape of Kunjen syllables. In Goyvaerts, Didier, L., editor, *Phonology in the 1980's*. Story-Scientia, Ghent.

Sproat, R. (1992). *Computational Morphology*. MIT Press.

Stevens, A. M. (1968). *Madurese Phonology and Morphology*. American Oriental Society, New Haven, CT.

Stevens, A. M. (1985). Reduplication in Madurese. In Choi, S., Devitt, D., Janis, W., McCoy, T., and Sheng-Sheng, Z., editors, *Proceedings of the Annual Eastern States Conference on Linguistics*, pages 232–242.

Stump, G. T. (2001). *Inflectional Morphology: A Theory of Paradigm Structure*. Cambridge University Press.

Thomas, W. (1997). Languages, automata, and logic. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer.

Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

Vaillette, N. (2003). Logical specification of regular relations for NLP. *Natural Language Engineering*, 9(1):65–85.

Vaillette, N. (2004). *Logical Specification of Finite-State Transductions for Natural Language Processing*. PhD thesis, The Ohio State University.

Valmari, A. and Lehtinen, P. (2008). Efficient minimization of DFAs with partial transition functions. In *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*.

Van Noord, G. (2000). Treatment of epsilon moves in subset construction. *Computational Linguistics*, 26(1):61–76.

Walther, M. (2000). Finite-state reduplication in one-level prosodic morphology. In *Proceedings of the first conference of the North American chapter of the Association for Computational Linguistics*.

Watson, B. W. (1995). *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technische Universiteit Eindhoven.

Watson, B. W. and Daciuk, J. (2003). An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64.

Wehr, H. (1979). *A Dictionary of Modern Written Arabic*. Spoken Language Services, Inc., Ithaca, NY.

Wilbur, R. (1973). *The Phonology of Reduplication*. PhD thesis, University of Illinois at Urbana-Champaign.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Yli-Jyrä, A. (2003). Describing syntax with star-free regular expressions. In *11th EACL 2003, Proceedings of the Conference*, pages 379–386.

Yli-Jyrä, A. (2007). A new method for compiling parallel replace rules. *Lecture Notes in Computer Science*, 4783.

Yli-Jyrä, A. (2008). Transducers from parallel replace rules and modes with generalized lenient composition. In *Proceedings of FSMNLP 2007*.

Yli-Jyrä, A. and Koskenniemi, K. (2004). Compiling contextual restrictions on strings into finite-state automata. In *The Eindhoven FASTAR Days Proceedings*.

Yona, S. and Wintner, S. (2008). A finite-state morphological grammar of Hebrew. *Natural Language Engineering*, 12(2):173–190.