| | |
|---|---|
| WSJ | Wall St. Journal |
| XFST | Xerox FST tools |
| XLE | Xerox Linguistic Environment |
| XTAG | TAG grammar from University of Pennsylvania |

# 1

# Introduction and Preliminaries

## 1.1 Introduction

Computational approaches to morphology and syntax are generally concerned with formal devices, such as grammars and stochastic models, and algorithms, such as tagging or parsing. They can range from primarily theoretical work, looking at, say, the computational complexity of algorithms for using a certain class of grammars, to mainly applied work, such as establishing best practices for statistical language modeling in the context of automatic speech recognition. Our intention in this volume is to provide a critical overview of the key computational issues in these domains along with some (though certainly not all) of the most effective approaches taken to address these issues. Some approaches have been known for many decades; others continue to be actively researched.

In many cases, whole classes of problems can be addressed using general techniques and algorithms. For example, finite-state automata and transducers can be used as formal devices for encoding many models, from morphological grammars to statistical part-of-speech taggers. Algorithms that apply to finite-state automata in general apply to these models. As much as possible, we will present specific computational approaches to syntax and morphology within the general class to which they belong. Much work in these areas can be thought of as variations on certain themes, such as finite-state composition or dynamic programming.

The book is organized into two parts: approaches to morphology and approaches to syntax. Since finite-state automata and transducers will figure prominently in much of the discussion in this book, in this chapter we introduce the basic properties of these devices as well as some of the algorithms and applications. For reasons of space we will only provide a high-level overview, but we will give enough references to recent work on finite automata and their applications so that interested readers can follow up on the details elsewhere.

One thing we hope to convey here is how to think of what automata compute in *algebraic* and *set-theoretic* terms. It is easy to get lost in the details of the algorithms and the machine-level computations. But what is really critical in understanding how finite automata are used in speech and language processing is to understand that they compute relations on sets. One of the critical insights of the early work by Kaplan and Kay from the 1970s (reported finally in Kaplan and Kay, 1994) was that in order to deal with complex problems such as the compilation of context-sensitive rewrite rules into transducers, one has to abandon thinking of the problem at the machine level and move instead to thinking of it at the level of what relations are being computed. Just as nobody can understand the wiring diagram of an integrated circuit, neither can one really understand a finite automaton of any complexity by simply looking at the machine. However, one can understand them easily at the algebraic level, and let algorithms worry about the details of how to compile that algebraic description into a working machine.

We assume that readers will be at least partly familiar with basic finite-state automata so we will only briefly review these. One can find reviews of the basics of automata in any introduction to the theory of computation such as Harrison (1978), Hopcroft and Ullman (1979), and Lewis and Papadimitriou (1981).

## 1.2 Finite-State Automata and Transducers

The study of finite-state automata (FSA) starts with the notion of a *language*. A language is simply a set of expressions, each of which is built from a set of symbols from an *alphabet*, where an alphabet is itself a set: typical alphabets in speech and language processing are sets of letters (or other symbols from a writing system), phones, or words.

The languages of interest here are *regular languages*, which are languages that can be constructed out of a finite alphabet – conventionally denoted $\Sigma$ – using one or more of the following operations:

| | | |
|---|---|---|
| • set union | denoted "$\cup$" | e.g., $\{a, b\} \cup \{c, d\} = \{a, b, c, d\}$ |
| • concatenation | denoted "$\cdot$" | e.g., $abc \cdot def = abcdef$ |
| • transitive closure (Kleene star) | denoted "$*$" | e.g., $a^*$ denotes the set of sequences consisting of 0 or more $a$'s |

Table 1.1 Phrasal reduplication in Bambara

| | | | |
|---|---|---|---|
| *wulu* | *o* | *wulu* | "whichever dog" |
| dog | MARKER | dog | |
| *wulunyinina* | *o* | *wulunyinina* | "whichever dog searcher" |
| dog searcher | MARKER | dog searcher | |
| *malonyininafilèla* | *o* | *malonyininafilèla* | "whichever rice searcher watcher" |
| rice searcher watcher | MARKER | rice searcher watcher | |

Any finite set of strings from a finite alphabet is necessarily a regular language, and using the above operations one can construct another regular language by taking the union of two sets $A$ and $B$; the concatenation of two or more such sets (i.e. the concatenation of each string in $A$ with each string $B$); or by taking the transitive closure (i.e. zero or more concatenations of strings from set $A$).

Despite their simplicity, regular languages can be used to describe a large number of phenomena in natural language including, as we shall see, many morphological operations and a large set of syntactic structures. But there are still linguistic constructions that cannot be described using regular languages. One well-known case from morphology is phrasal reduplication in Bambara, a language of West Africa (Culy, 1985), some examples of which are given in Table 1.1. Bambara phrasal reduplication constructions are of the form *X-o-X*, where *-o-* is a marker of the construction and *X* is a nominal phrase. The problem is that the nominal phrase is in theory unbounded, and so the construction involves unbounded copying. Unbounded copying cannot be described in terms of regular languages; indeed it cannot even be described in terms of context-free languages (which we will return to later in the book).

A couple of important regular languages are the *universal language* (denoted $\Sigma^*$) which consists of all strings that can be constructed out of the alphabet $\Sigma$, including the empty string, which is denoted $\epsilon$; and the *empty language* (denoted $\emptyset$) consisting of no strings.

The definition given above defines some of the *closure properties* for regular languages but regular languages are also closed under the following operations:

| | | |
|---|---|---|
| • intersection | denoted "$\cap$" | e.g., $\{a, b, c\} \cap \{c, d\} = \{c\}$ |
| • difference | denoted "$-$" | e.g., $\{a, b, c\} - \{c\} = \{a, b\}$ |
| • complementation | denoted "$\overline{X}$" | e.g., $\overline{\mathrm{A}} = \Sigma^* - \mathrm{A}$ |
| • string reversal | denoted "$X^R$" | e.g., $(abc)^R = cba$ |

Regular languages are commonly denoted via *regular expressions*, which involve the use of a set of reserved symbols as notation. Some of these reserved symbols we have already seen, such as "*", which denotes "zero or more" of the symbol that it follows: recall that $a^*$ denotes the (infinite) set of strings consisting of zero or more $a$'s in sequence. We can denote the repetition of multi-symbol sequences by using a parenthesis delimiter: $(abc)^*$ denotes the set of strings with zero or more repetitions of $abc$, that is, $\{\emptyset, abc, abcabc, abcabcabc, \ldots\}$. The following summarizes several additional reserved symbols used in regular expressions:

| | | |
|---|---|---|
| • "zero or one" | denoted "?" | e.g.,$(abc)$? denotes $\{\emptyset, abc\}$ |
| • disjunction | denoted "\|" or $\cup$ | e.g., $(a \mid b)$? denotes the set of strings with zero or one occurrence of either $a$ or $b$, i.e., $\{\emptyset, a, b\}$ |
| • negation | denoted "$\neg$" | e.g., $(\neg a)^*$ denotes the set of strings with zero or more occurrences of anything other than $a$ |

This is a relatively abbreviated list, but sufficient to understand the regular expressions used in this book to denote regular languages.

*Finite-state automata* are computational devices that compute regular languages. Formally defined:

**Definition 1** *A finite-state automaton is a quintuple $M = (Q, s, F, \Sigma, \delta)$ where:*

1. *$Q$ is a finite set of states*
2. *$s$ is a designated initial state*
3. *$F$ is a designated set of final states*
4. *$\Sigma$ is an alphabet of symbols*
5. *$\delta$ is a transition relation from $Q \times (\Sigma \cup \epsilon)$ to $Q$, where $A \times B$ denotes the cross-product[1] of sets $A$ and $B$*

*Kleene's theorem* states that every regular language can be recognized by a finite-state automaton; similarly every finite-state automaton recognizes a regular language.

[1] The cross-product of two sets creates a set of pairs, with each member of the first set paired with each member of the second set. For example, $\{a, b\} \times \{c, d\} = \{\langle a, c\rangle, \langle a, d\rangle, \langle b, c\rangle, \langle b, d\rangle\}$. Thus the transition relation $\delta$ is from state/symbol pairs to states.

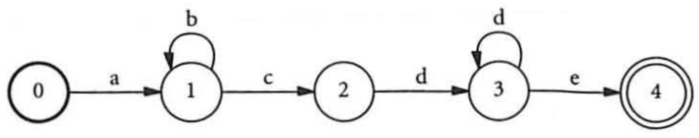

FIGURE 1.1 A simple finite-state automaton accepting the language $ab^*cdd^*e$

A diagram of a simple finite-state automaton, which accepts the language $ab^*cdd^*e$, is given in Figure 1.1. A string, say *abbcddde*, that is in the language of the automaton is matched against the automaton as follows: starting in the initial state (here, state 0), the match proceeds by reading a symbol of the input string and matching it against a transition (or arc) that leaves the current state. If a match is found, one moves to the destination state of the arc, and tries to match the next symbol of the input string with an arc leaving that state. If one can follow a path through the automaton in such a manner and end in a final state (denoted here with a double circle) with all symbols of the input read, then the string is in the language of the automaton; otherwise it is not. Note that the operation of *intersection* of two automata (see Section 1.5) follows essentially the same algorithm as just sketched, except that one is matching paths in one automaton against another, instead of a string. Note also that one can represent a string as a single-path automaton, so that the string-matching method we just described can be implemented as automata intersection.

We turn from regular languages and finite-state automata to *regular relations* and *finite-state transducers* (FST). A regular relation can be thought of as a regular language over n-tuples of symbols, but it is more usefully thought of as expressing relations between sets of strings. The definition of a regular n-relation is as follows:

1. $\emptyset$ is a regular n-relation
2. For all symbols $a \in [(\Sigma \cup \epsilon) \times \ldots \times (\Sigma \cup \epsilon)]$, $\{a\}$ is a regular n-relation
3. If $R_1$, $R_2$, and $R$ are regular n-relations, then so are
   (a) $R_1 \cdot R_2$, the *(n-way) concatenation* of $R_1$ and $R_2$: for every $r_1 \in R_1$ and $r_2 \in R_2$, $r_1 r_2 \in R_1 \cdot R_2$
   (b) $R_1 \cup R_2$
   (c) $R^*$, the *n-way transitive (Kleene) closure* of $R$.

For most applications in speech and language processing $n = 2$, so that we are interested in relations between pairs of strings.[2] In what follows we will be dealing only with 2-relations.

[2] An exception is Kiraz (2000), who uses n-relations, $n > 2$ for expressing non-concatenative Semitic morphology; see Section 2.2.9.

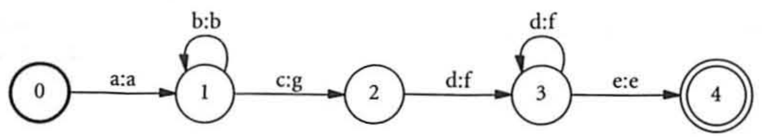

FIGURE 1.2 A simple finite-state transducer that computes the relation $(a{:}a)(b{:}b)^*$ $(c{:}g)(d{:}f)(d{:}f)^*(e{:}e)$

Analogous to finite-state automata are *finite-state transducers*, defined as follows:

**Definition 2** *A (2-way) finite-state transducer is a quintuple* $M = (Q, s, F, \Sigma \times \Sigma, \delta)$ *where:*

1. $Q$ *is a finite set of states*
2. $s$ *is a designated initial state*
3. $F$ *is a designated set of final states*
4. $\Sigma$ *is an alphabet of symbols*
5. $\delta$ *is a transition relation from* $Q \times (\Sigma \cup \epsilon \times \Sigma \cup \epsilon)$ *to* $Q$

A simple finite-state transducer is shown in Figure 1.2. With a transducer, a string matches against the input symbols on the arcs, while at the same time the machine is outputting the corresponding output symbols. Thus, for the input string *abbcddde*, the transducer in Figure 1.2 would produce *abbgfffe*. A transducer determines if the input string is in the domain of the relation, and if it is, computes the corresponding string, or set of strings, that are in the range of the relation.

The closure properties of regular relations are somewhat different from those of regular languages, and the differences are outlined in Table 1.2. The major differences are that relations are not closed under intersection, a point that will be important in Chapter 4 when we discuss the KIMMO morphological analyzer (see Section 4.2.1); and that relations are closed under a new property, namely *composition*. Composition – denoted $\circ$ – is to be understood in the sense of composition of functions. If $f$ and $g$ are two regular relations and $x$ a string, then $[f \circ g](x) = f(g(x))$. In other words, the output of the composition of $f$ and $g$ on a string $x$ is the output that would be obtained by first applying $g$ to $x$ and then applying $f$ to the output of that first operation.

Composition is a very useful property of regular relations. There are many applications in speech and language processing where one wants to factor a system into a set of operations that are cascaded together using composition. A case in point is in the implementation

TABLE 1.2 Closure properties for regular languages and regular relations

| Property | Languages | Relations |
|---|---|---|
| concatenation | yes | yes |
| Kleene closure | yes | yes |
| union | yes | yes |
| intersection | yes | no |
| difference | yes | no |
| composition | — | yes |
| inversion | — | yes |

of phonological rule systems. Phonological rewrite rules of the kind used in early Generative Grammar can be implemented using regular relations and finite-state transducers.[3] Traditionally such rule systems have involved applying a set of rules in sequence, each rule taking as input the output of the previous rule. This operation is implemented computationally by composing the transducers corresponding to the individual rules.

Table 1.2 also lists *inversion* as one of the operations under which regular relations are closed. Inversion consists of swapping the domain of the relation with the range; in terms of finite-state transducers, one simply swaps the input and output labels on the arcs. The closure of regular relations under composition and inversion leads to the following nice property: one can develop a rule system that compiles into a transducer that maps from one set of strings to another set of strings, and then invert the result so that the relation goes the other way. An example of this is, again, generative phonological rewrite rules. It is generally easier for a linguist to think of starting with a more abstract representation and using rules to derive a surface representation. Yet in a morphological analyzer, one generally wants the computation to be performed in the other direction. Thus one takes the linguist's description, compiles it into finite-state transducers, composes these together and then inverts the result.

Of course, regular relations resulting from such descriptions are likely to be many-to-one, as in many input strings mapping to one output string; for example, many underlying forms mapping to the

[3] We will not discuss these compilation algorithms here as this would take us too far afield; the interested reader is referred to Kaplan and Kay (1994) and Mohri and Sproat (1996).

same surface form. In such a case, the inversion yields a one-to-many relation, resulting in the need for disambiguation between the many underlying forms that could be associated with a particular surface form.

## 1.3 Weights and Probabilities

Disambiguation in morphological and syntactic processing is often done by way of stochastic models, in which weights can encode preferences for one analysis versus another. In this section, we will briefly review notation for weights and probabilities that will be used throughout the book.

Calculating the sum or product over a large number of values is very common, and a common shorthand is to use the sum ($\sum$) or product ($\prod$) symbols ranging over variables. For example, to sum the numbers one through nine, we can write:

$$\sum_{i=1}^{i<10} i = 1+2+3+4+5+6+7+8+9 = 45 \qquad (1.1)$$

Similarly, to multiply them:

$$\prod_{i=1}^{i<10} i = 1*2*3*4*5*6*7*8*9 = 362880 \qquad (1.2)$$

Logarithms (log) and exponentials (exp) are also very common, and we will by convention use natural logarithm,[4] base $e$. Recall the basic relationship between them: for any $x, y$

$$\log(\exp(x)) = \log(e^x) = x \qquad (1.3)$$

$$\exp(\log(y)) = e^{\log(y)} = y \qquad (1.4)$$

One of the nicest properties of logs is that the log of a product is a sum:

$$\log\left(\prod_{i=1}^{i<10} i\right) = \sum_{i=1}^{i<10} \log(i) \qquad (1.5)$$

This is nice because the log of each $i$ can be taken separately, prior to combination, rather than having to combine them before taking the log. This is critical when the product leads to extremely large or

[4] When multiple bases are used, natural log is sometimes denoted *ln*, but here we will just use log.

extremely small floating point numbers. Another nice property is that log is order preserving. That is, if $x > y$, then $\log(x) > \log(y)$.

Briefly, let us introduce simple empirically estimated probabilities of the sort we will mainly be considering in this book. All of the probabilistic models that we will be discussing are discrete distributions, where there are $k$ discrete outcomes (such as different words from a vocabulary $\Sigma$ of size $k$) each with its own parameter. When $k = 2$, this is known as a binomial distribution; when $k > 2$, this is a multinomial distribution. For example, we can assign a probability to each word $w$ in a vocabulary $\Sigma$; this is a multinomial distribution with $|\Sigma|$ parameters (one parameter $\mathrm{P}(w)$ for each word $w \in \Sigma$) where $\sum_{w\in\Sigma} \mathrm{P}(w) = 1$.

If we have a corpus of $N$ words taken from a vocabulary $\Sigma$, we can calculate the probability of any observed word $w \in \Sigma$ in that corpus using *relative frequency estimation*:

$$\mathrm{P}(w) = \frac{f(w)}{N} \qquad (1.6)$$

where $f(w)$ is the frequency of the word (its count). Note that using relative frequency estimation leads us to give zero probability to words that have not occurred in our corpus, a problem that is discussed later in the book.

We might want to find the most probable word in the corpus, that is, the word with the maximum probability. The maximum probability, here denoted $\hat{p}$, is

$$\hat{p} = \max_{w} \mathrm{P}(w) \qquad (1.7)$$

If we want to know the word that provides us with this maximum probability, we use "argmax":

$$\hat{w} = \operatorname*{argmax}_{w} \mathrm{P}(w) \qquad (1.8)$$

Then, by definition, $\mathrm{P}(\hat{w}) = \hat{p}$.

Of course, if we are using *negative log probabilities*, the order reverses, hence we will be more interested in the "min" and "argmin", which are defined similarly.

## 1.4 Weighted Finite-State Automata and Transducers

Finite-state automata and transducers can be extended to include *weights* or *costs* on the arcs. Such machines are termed *weighted finite-state automata* (WFSA) and *weighted finite-state transducers* (WFST).

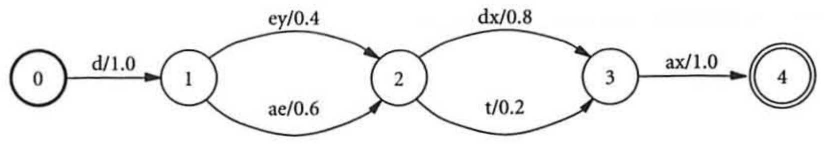

FIGURE 1.3 Representations of several pronunciations of the word *data*, with transcriptions in ARPAbet

The weights can serve a number of functions, but the most frequent use in speech and language processing is to represent probabilities, or more commonly, negative log probabilities, of different analyses.

An example is shown in Figure 1.3. In this example, several plausible pronunciations are shown for the word *data*, with associated probabilities; note that the probabilities for all arcs leaving a given state must sum to one. (The four pronunciations correspond to the IPA transcriptions /deɪtə/, /dætə/, /deɪɾə/, and /dæɾə/.) The probability of a particular path is given by multiplying the individual arc probabilities along the path. In this example, for instance, the pronunciation /d ey t ax/ has the probability $1 * 0.4 * 0.2 * 1 = 0.08$.

In a toy example like the one we have just examined, it is reasonable to represent probabilities as themselves, but in any realistic scenario this presents a computational problem since the probabilities along any given path can be very small and will generally lead to difficulties in floating point represention of the values. Thus it is more common to represent probabilities in the log domain and, more specifically, to represent them in terms of negative logs. Note that in this representation, smaller numbers correspond to more probable events. Recall that, if we use negative log probabilities, we must *sum* the weights along the path rather than multiply them.

When automata and transducers are given weights, the interpretation of those weights must be provided. In addition to specifying how weights are combined *along* a path (which for probabilities is by multiplication), one must also specify how weights are combined *between* paths. For example, suppose there are two paths in an automaton with the same symbols on the arc labels of the paths, and we want to collapse those into a single path. How are the weights combined? When dealing with straight probabilities, in order to ensure proper normalization, the weights (probabilities) of two paths are added together. Different kinds of weights – e.g., logs or probabilities – will have different ways of combining the weights along the path (which we will generally term the *times* operation) and between paths (which we will term the *plus* operation).

The different interpretations of weights are usefully unified in terms of *semirings*. Before we can introduce the notion of a semiring, we first need the definition of a *monoid*:

**Definition 3** *A monoid is a pair $(M, \bullet)$, where $M$ is a set and $\bullet$ is a binary operation on $M$, obeying the following rules:*

1. ***closure:*** *for all $a, b$ in $M$, $a \bullet b$ is in $M$*
2. ***identity:*** *there exists an element $e$ in $M$, such that for all $a$ in $M$, $a \bullet e = e \bullet a = a$. This is termed the neutral element.*
3. ***associativity:*** *$\bullet$ is an associative operation; that is, for all $a, b, c$ in $M$, $(a \bullet b) \bullet c = a \bullet (b \bullet c)$*

A monoid $(M, \bullet)$ is *commutative* if $a \bullet b = b \bullet a$ for all $a, b$ in $M$. We can take this notion of monoid and define semirings in terms of notional summation and multiplication as follows:

**Definition 4** *A semiring is a triple $(\mathbb{K}, \oplus, \otimes)$, where $\mathbb{K}$ is a set and $\oplus$ and $\otimes$ are binary operations on $\mathbb{K}$, obeying the following rules:*

1. *$(\mathbb{K}, \oplus)$ is a commutative monoid with neutral element denoted by* $0$
2. *$(\mathbb{K}, \otimes)$ is a monoid with neutral element denoted by* $1$
3. *The product $(\otimes)$ distributes with respect to the sum $(\oplus)$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$*
4. *For all $a$ in $\mathbb{K}$, $a \otimes 0 = 0 \otimes a = 0$*

Since most applications use real numbers as the semiring set, one typically denotes a particular semiring with a pair $(\oplus, \otimes)$ that specifies the actual instantiation of the notional plus and times operations. Common semirings used in speech and language processing are the $(+, \times)$ or "real" semiring, and the $(min, +)$ or "tropical" semiring.[5] The $(+, \times)$ semiring is appropriate for use with probabilities: to get the probability of a path, one *multiplies* along the path; to get the probability of a set of paths, one *sums* the probabilities of those paths. The $(min, +)$ semiring is appropriate for use with negative log probabilities: one *sums* the weights along the path, and one computes the minimum of a set of paths – which is useful if one is looking for the best scoring path, since lower scores are better with negative logs.

[5] So called because the mathematician who pioneered this semiring, Imre Simon, was from Brazil.

Weighted finite-state automata (and the corresponding weighted languages) are closed under intersection. When one combines two paths via intersection, the resulting cost is obtained by using the semiring $\otimes$ operation. Note that $\otimes$ is often referred to as the *extend* operation, since it is the operation that one uses when one extends a path by an additional arc.

A formal definition of a weighted finite automaton is as follows:

**Definition 5** *A weighted finite-state automaton is an octuple* $A = (Q, s, F, \Sigma, \delta, \lambda, \sigma, \rho)$, *where:*

1. $(Q, s, F, \Sigma, \delta)$ *is a finite-state automaton*
2. *An initial output function* $\lambda: s \rightarrow \mathbb{K}$ *assigns a weight to entering the automaton*
3. *An output function* $\sigma: \delta \rightarrow \mathbb{K}$ *assigns a weight to transitions in the automaton*
4. *A final output function* $\rho: F \rightarrow \mathbb{K}$ *assigns a weight to leaving the automaton*

For any transition $d \in \delta$, let $i[d] \in (\Sigma \cup \epsilon)$ be its label; $p[d] \in Q$ its origin state; and $n[d] \in Q$ its destination state. A path $\pi = d_1 \ldots d_k$ consists of $k$ transitions $d_1, \ldots, d_k \in \delta$, where $n[d_j] = p[d_{j+1}]$ for all $j$, i.e., the destination state of transition $d_j$ is the origin state of transition $d_{j+1}$. We can extend the definitions of label, origin and destination to paths: let $i[\pi] = i[d_1] \ldots i[d_k]$; $p[\pi] = p[d_1]$; and $n[\pi] = n[d_k]$. A *cycle* is a path $\pi$ such that $p[\pi] = n[\pi]$, i.e., a path that starts and ends at the same state. An *acyclic* automaton or transducer has no cycles.

We can also extend the definition of the $\sigma$ function of Definition 5 to paths: $\sigma[\pi] = \sigma[d_1] \otimes \cdots \otimes \sigma[d_k]$. Let $P(q, x, q')$ be the set of paths $\pi$ such that $p[\pi] = q$, $i[\pi] = x$, and $n[\pi] = q'$. Given a semiring $(\mathbb{K}, \oplus, \otimes)$, the weight associated by A to a string $x \in \Sigma^*$ can be defined as follows:

$$[[A]](x) = \bigoplus_{f \in F} \bigoplus_{\pi \in P(s,x,f)} \lambda(s) \otimes \sigma(\pi) \otimes \rho(f)$$

Weighted finite-state transducers are an obvious extension of finite-state transducers and weighted automata. A WFST computes a regular relation, but in addition it associates each mapping with a weight. For example, in a transducer encoding a rewrite rule system, the weights might represent the probabilities of a particular rule application.

## 1.5 A Synopsis of Algorithmic Issues

The basic texts on automata theory that we have already cited give algorithms for various finite-state operations including concatenation, Kleene closure, union, intersection, complementation, determinization, and minimization. While these algorithms obviously produce correct results and work fine for small automata and transducers, they are often not efficient enough to handle the very large machines that are typical of serious speech- and language-processing applications. Furthermore, the textbook algorithms do not deal with weighted automata, and the correct treatment of weights turns out to be critical for efficient processing. Some algorithmic issues that have been very important in the application to speech and language processing include efficient algorithms for composition, minimization, determinization, and epsilon removal. In this section we will provide a brief high-level overview of some of these algorithmic issues, with pointers to some papers that deal with them in depth.

One of the most fundamental algorithms that we will be assuming for much of our discussion of finite-state methods in this book is *composition*, so it is useful to have a basic understanding of how this works. At its core, composition is essentially the same as automata intersection as defined in standard texts. We start by reminding the reader how intersection works. The basic algorithm for intersection is as follows. Given two automata $M = (Q, s, F, \Sigma, \delta)$ and $M' = (Q', s', F', \Sigma', \delta')$, construct a new automaton $M''$ such that:

- Its set of states $Q'' = Q \times Q'$ is the cross-product of the states of the individual machines.
- $s'' = (s, s')$
- $F'' = F \times F'$
- $\Sigma'' = \Sigma \cap \Sigma'$
- $\delta''((p, p'), x) = (q, q')$ just in case $\delta(p, x) = q$ is in $M$ and $\delta'(p', x) = q'$ is in $M'$.

The basic algorithm for transducer composition is essentially the same, with the difference that with transducers one is matching the *output* label of one transducer with the *input* label of the other. The resulting arc has as its input label the input label of the arc from the first machine and as its output label the output label of the arc from the second machine. Automata can be seen as a special case of transducers, where the input and output symbols are always identical.

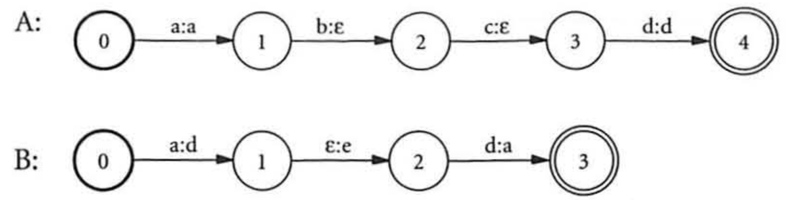

FIGURE 1.4 Two transducers with epsilons. Example taken from Pereira and Riley (1997)

When one is dealing with *weighted* intersection or composition, as we noted above, one computes the weights of the resulting path as the extend ($\otimes$) of the weights of the two input paths.

Even with the basic algorithms there are various efficiency issues. Computation of the transition function $\delta''$ for a new state $(p, p')$ and label $x$ requires efficient search. For example, suppose we are looking at transducer $M$, at state $p$, and at an arc with an output label $x$. We wish to find in $M'$, state $p'$, the set of arcs, if any, that have input labels $x$. If the arcs of the second machine are arranged in no particular order, then one has no choice but to search linearly through the arcs in $M'$ to find any that have input label $x$, and this will be inefficient if there are a large number of arcs exiting $p'$. A solution is to index $M'$ on the input side, so that for any state the arcs are sorted by input label, allowing for a more efficient search method, such as a binary search. Even more efficient methods are possible.

The complication with transducers involves epsilons: arcs where the output side (if on the first transducer) or the input side (if on the second transducer) are labeled with the empty-string symbol $\epsilon$. Epsilons allow one to implement *different-length* relations (as opposed to *same-length* relations) so that, for example, one can implement a rule that deletes symbols in certain contexts. In such a case the transducer would contain arcs labeled with the symbols in question on the input side and $\epsilon$ on the output side. The problem with epsilons is that they introduce non-determinism over and above the non-determinism that one would have due to one or both of the input machines being non-deterministic, and hence inefficiency.[6] With weighted transducers the situation is worse: one will actually get the wrong result.

[6] Note that the same issue arises with epsilons in the intersection of acceptors, and the same solution applies. However, note also that acceptors can always have their epsilons removed before intersection, whereas with transducers it is not generally possible to remove epsilons when the epsilon is only on one side of the arc label pair.

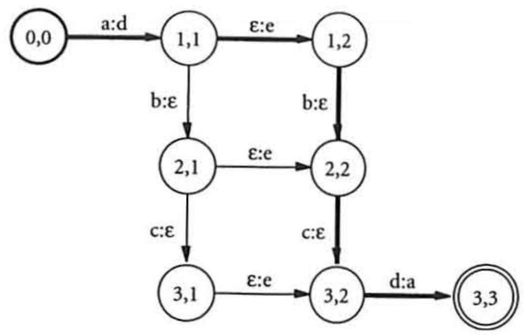

FIGURE 1.5 Naive composition of the transducers from Figure 1.4. Example taken from Pereira and Riley (1997)

To see this, consider the two transducers in Figure 1.4, from Pereira and Riley (1997). In composing $A$ with $B$, what is at issue is how to move from state 1 through 2 and 3 in machine $A$ and at the same time move from state 1 to 2 in machine $B$. Since these operations consume no output in $A$ or input in $B$, there are in principle a number of ways one could do this. One could, for example traverse both arcs from states 1 to 3 in $A$ before traversing any arc in $B$; or one could traverse the arc between 1 and 2 in $B$ before traversing any arcs in $A$; or one could choose to move from 1 to 2 in $A$, then stay in 2 in $A$ while moving from 1 to 2 in $B$, and then complete the transition to 3 in $A$. These various options are diagrammed in Figure 1.5. These multiple paths lead to inefficiency in unweighted transducers but are otherwise correct. In weighted transducers, however, they yield the wrong result for the simple reason that the weights from the two original paths will be combined in each of the alternatives (via $\otimes$); these multiple alternatives will then be combined (via $\oplus$), meaning that in many semirings the resulting cost of the intersection of the two original paths will be wrong. The solution to this problem is to insert an *epsilon filter* $F$ between the transducers $A$ and $B$ so that in effect one is composing $A \circ F \circ B$; the transducer $F$ forces the result to have just one of the paths, specifically the bold-marked path in Figure 1.5.[7]

Beyond composition, other algorithmic issues that arise relate to determinization, minimization and epsilon removal. Weighted automata and transducers (whether weighted or not) cannot in general be determinized, but certain types of machines, including acyclic

[7] The actual algorithm is somewhat more complicated than what we have sketched here, and the epsilon filter transducer is simulated rather than actually constructed; see Pereira and Riley (1997) for further details.

machines, can be. (See Mohri, 1997, for a rigorous characterization of the class of determinizable machines.) Since machine minimization requires a determinized machine (Harrison, 1978; Hopcroft and Ullman, 1979; Lewis and Papadimitriou, 1981), this also implies that not all weighted acceptors or transducers can be minimized, though, again, some classes of machine can be. Transducers and weighted acceptors that fall into the class of determinizable and minimizable machines include machines that are useful in speech and language processing. For example, a dictionary can be modeled as an acyclic transducer, mapping input words to some other property such as their part of speech or pronunciation; and a *lattice* of possible analyses output by a speech recognizer can be modeled as an acyclic weighted acceptor. Determinizing and minimizing such machines can provide large efficiency gains (Mohri and Riley, 1999).[8] Epsilon removal with weighted automata is an interesting algorithmic issue in particular because epsilon arcs may have weights, and one must therefore be careful to distribute the weights correctly once the epsilon arcs are removed (Mohri, 2002).

## 1.6 Computational Approaches to Morphology and Syntax

One might wonder why so much attention is paid to finite-state methods in this book, even in the sections that are devoted to syntax. Weren't finite-state techniques mostly relegated to the dustheap during the 1970s? That has certainly been one common view. In the mid 1990s one of the authors gave a talk at a major industrial research lab in the Seattle, Washington area. He presented some work on applications of finite-state methods to text analysis for text-to-speech synthesis. Several natural language researchers in the audience reacted negatively to his presentation, claiming that finite-state methods were outdated and belonged in the 1970s not the 1990s.

Developments over the past decade have proved this view to be ill-founded: there has been a veritable explosion of research in finite-state methods with applications in a number of areas of speech and language processing including morphology and phonology (in which there was already substantial work by the mid 1990s and as we shall discuss further below), the computational analysis of syntax (e.g., Voutilainen, 1994; Mohri, 1994, and see Chapter 6), language modeling for speech recognition (Pereira and Riley, 1997; Mohri et al., 2002), text normalization systems for speech synthesis (Sproat, 1997a), pronunciation modeling (Mohri et al., 2002), the analysis of document structure (Sproat et al., 1998), inter alia. Outside speech and language processing, finite-state methods have found applications in other fields, such as computational biology (Durbin et al., 1998). Thus, if we seem to dwell too much on finite-state methods in this book, it is for a reason: such methods have a broad range of applications and students of speech and language processing would do well to master them.

Despite the broad applicability of finite-state methods, however, there is a fundamental difference between computational approaches to morphology and computational approaches to syntax, in that the former (we shall argue) can be accomplished entirely with finite-state methods, while the latter cannot. Finite-state approaches to syntax can be extremely efficient and useful for many applications requiring some amount of syntactic processing, but it has been widely known since Chomsky (1957) that many syntactic phenomena simply cannot be described without context-free or even context-sensitive grammars. Grammars built for computational syntactic processing must typically trade-off the richness of syntactic description provided by the grammar with the computational cost of using it. Often the utility of a syntactic annotation will not justify – within the context of a particular application – the cost of annotating it. This is much less of an issue for morphological processing, since finite-state models and algorithms are generally sufficient for morphological description.

Computationally, a grammar may be used for syntactic processing in several ways. First, it may be used to generate word strings in the language described by the grammar. It may also be used to recognize (or accept) strings in the language and reject strings that are not in the language. The grammar may additionally be used to provide some useful annotation to the accepted strings, such as, labels, delimiters, or perhaps a numerical score. Very often it is this annotation, and not just acceptance or rejection, that makes grammars useful computationally.

The quality of a grammar (or syntactic model) is usually inversely related to the efficiency with which the model can be built and used. Very rich syntactic formalisms that find favor with syntacticians because they do a good job of accepting just those sentences that are grammatical in a language are often not used because explicit, detailed grammars require significant expertise and a relatively long time to write, and because the most efficient recognition algorithms that make use of such grammars are simply not efficient enough for particular

[8] Even for machines that cannot be determinized, it is often possible to *locally* determinize them (Mohri, 1997).

applications. The most commonly implemented syntactic models fall far short of what linguists would expect from a grammar in terms of describing languages, yet they provide useful information and are both easy to build and efficient to use. These latter considerations will carry the day, unless a compelling difference in application performance can be demonstrated.

A very important consideration is robustness in the face of noise. Unless the application is severely constrained, for example, machine translation of official weather reports, the language usage will be varied and noisy. In more natural, less constrained settings, any grammar will be presented with false starts, disfluencies, sentence fragments, out-of-vocabulary words, misspellings, run-on sentences, or just plain ungrammaticalities. A parser is usually expected to yield some useful information to an application beyond rejection, even in the face of these phenomena.

Issues of efficiency and robustness have made simple, weighted finite-state methods very popular. However, much research is currently focused on enriching robust syntactic models, and making richer syntactic formalisms more efficient and robust. Part II of this book (chapters 6–9) will look at various computational approaches to syntax, starting with the simplest and most efficient techniques before moving on to richer ones. The inclusion of scores among syntactic annotations will be particularly emphasized, since this particular annotation, as shall be seen, can make the difference between a useless model and a very useful one. It is in computational linguistics, more than any other subdiscipline of linguistics, where statistical and probabilistic approaches to disambiguation have been investigated, and it is this most of all that distinguishes computational approaches to syntax from other perspectives.

Many of the most successful computational models of syntax share much in common with constraint-based linguistic formalisms, such as Optimality Theory (OT), with some simple differences. What is shared is the notion of a feature or constraint that encodes some informative linguistic distinction. General, effective automatic feature induction methods are beyond the current state-of-the-art in Natural Language Processing, so that effective manual feature selection – often based on linguist-documented generalizations – is a key part of building effective syntactic analyzers. Computational approaches typically differ from OT approaches primarily in terms of how the evidence of various features/constraints is combined, but also in terms of the explicitness of the candidate generation mechanism, known as GEN in OT. Chapters 6–9 will present just how serious a problem efficient candidate generation can be, and a range of computational solutions. Disambiguation through candidate ranking will be presented in Chapter 9.

In Chapter 2, we will show that models of morphology can be implemented in a unified framework of finite-state transducers. That is not the case for syntax, which brings efficiency to center stage. As a result, Part II (Computational Approaches to Syntax) will have much more of a focus on the accuracy/efficiency trade-off than Part I (Computational Approaches to Morphology), with efficient approximations that provide useful annotations receiving much attention. Efficient algorithms will be explicitly presented, to clearly illustrate why computational linguists are forced to make the choices they do. Differences in focus between the two parts of the book reflect differences in the key issues driving the two topic areas.