discuss computational models of **phonological learning**: how phonological rules can be automatically induced by machine learning algorithms.

Finally, we apply the transducer-based model of phonology to an important problem in text-to-speech systems: mapping from strings of letters to strings of phones. We first survey the issues involved in building a large pronunciation dictionary, and then show how the transducer-based lexicons and spelling rules of Chapter 3 can be augmented with pronunciations to map from orthography to pronunciation.

This chapter focuses on the non-probabilistic areas of computational linguistics and pronunciations modeling. Chapter 5 will turn to the role of probabilistic models, including such areas as probabilistic models of pronunciation variation and probabilistic methods for learning phonological rules.

4.1 SPEECH SOUNDS AND PHONETIC TRANSCRIPTION

The study of the pronunciation of words is part of the field of **phonetics**, the study of the speech sounds used in the languages of the world. We will be modeling the pronunciation of a word as a string of symbols which represent **phones** or **segments**. A phone is a speech sound; we will represent phones with phonetic symbols that bears some resemblance to a letter in an alphabetic language like English. So for example there is a phone represented by *l* that usually corresponds to the letter *l* and a phone represented by *p* that usually corresponds to the letter *p*. Actually, as we will see later, phones have much more variation than letters do. This chapter will only briefly touch on other aspects of phonetics such as **prosody**, which includes things like changes in pitch and duration.

PHONETICS

PHONES

This section surveys the different phones of English, particularly American English, showing how they are produced and how they are represented symbolically. We will be using two different alphabets for describing phones. The first is the **International Phonetic Alphabet (IPA)**. The IPA is an evolving standard originally developed by the International Phonetic Association in 1888 with the goal of transcribing the sounds of all human languages. The IPA is not just an alphabet but also a set of principles for transcription, which differ according to the needs of the transcription, so the same utterance can be transcribed in different ways all according to the principles of the IPA. In the interests of brevity in this book we will focus on the symbols that are most relevant for English; thus Figure 4.1 shows a subset of the IPA symbols for transcribing consonants, while Figure 4.2 shows a subset of the IPA

IPA

IPA Symbol	ARPAbet Symbol	Word	IPA Transcription	ARPAbet Transcription
[p]	[p]	parsley	['pɑrsli]	[p aa r s l iy]
[t]	[t]	tarragon	['tærəgɑn]	[t ae r ax g aa n]
[k]	[k]	catnip	['kætnɨp]	[k ae t n ix p]
[b]	[b]	bay	[beɪ]	[b ey]
[d]	[d]	dill	[dɪl]	[d ih l]
[g]	[g]	garlic	['gɑrlɨk]	[g aa r l ix k]
[m]	[m]	mint	[mɪnt]	[m ih n t]
[n]	[n]	nutmeg	['nʌtmɛg]	[n ah t m eh g
[ŋ]	[ng]	ginseng	['dʒɪnsɨŋ]	[jh ih n s ix ng]
[f]	[f]	fennel	['fɛnl̩]	[f eh n el]
[v]	[v]	clove	[kloʊv]	[k l ow v]
[θ]	[th]	thistle	['θɪsl̩]	[th ih s el]
[ð]	[dh]	heather	['hɛðɚ]	[h eh dh axr]
[s]	[s]	sage	[seɪdʒ]	[s ey jh]
[z]	[z]	hazelnut	['heɪzl̩nʌt]	[h ey z el n ah t]
[ʃ]	[sh]	squash	[skwɑʃ]	[s k w a sh]
[ʒ]	[zh]	ambrosia	[æm'broʊʒə]	[ae m b r ow zh ax]
[tʃ]	[ch]	chicory	['tʃɪkɚi]	[ch ih k axr iy]
[dʒ]	[jh]	sage	[seɪdʒ]	[s ey jh]
[l]	[l]	licorice	['lɪkɚɨʃ]	[l ih k axr ix sh]
[w]	[w]	kiwi	['kiwi]	[k iy w iy]
[r]	[r]	parsley	['pɑrsli]	[p aa r s l iy]
[j]	[y]	yew	[yu]	[y uw]
[h]	[h]	horseradish	['hɔrsrædɪʃ]	[h ao r s r ae d ih sh]
[ʔ]	[q]	uh-oh	[ʔʌʔoʊ]	[q ah q ow]
[ɾ]	[dx]	butter	['bʌɾɚ]	[b ah dx axr]
[ɾ̃]	[nx]	wintergreen	[wɪɾ̃ɚgrin]	[w ih nx axr g r i n]
[l̩]	[el]	thistle	['θɪsl̩]	[th ih s el]

Figure 4.1 IPA and ARPAbet symbols for transcription of English consonants.

symbols for transcribing vowels.[1] These tables also give the ARPAbet symbols; ARPAbet (Shoup, 1980) is another phonetic alphabet, but one that is specifically designed for American English and which uses ASCII symbols;

[1] For simplicity we use the symbol [r] for the American English "r" sound, rather than the more-standard IPA symbol [ɹ].

it can be thought of as a convenient ASCII representation of an American-English subset of the IPA. ARPAbet symbols are often used in applications where non-ASCII fonts are inconvenient, such as in on-line pronunciation dictionaries.

IPA Symbol	ARPAbet Symbol	Word	IPA Transcription	ARPAbet Transcription
[i]	[iy]	lily	[ˈlɪli]	[l ih l iy]
[ɪ]	[ih]	lily	[ˈlɪli]	[l ih l iy]
[eɪ]	[ey]	daisy	[ˈdeɪzi]	[d ey z i]
[ɛ]	[eh]	poinsettia	[pɔɪnˈsɛɾiə]	[p oy n s eh dx iy ax]
[æ]	[ae]	aster	[ˈæstɚ]	[ae s t axr]
[ɑ]	[aa]	poppy	[ˈpɑpi]	[p aa p i]
[ɔ]	[ao]	orchid	[ˈɔrkɨd]	[ao r k ix d]
[ʊ]	[uh]	woodruff	[ˈwʊdrʌf]	[w uh d r ah f]
[oʊ]	[ow]	lotus	[ˈloʊɾəs]	[l ow dx ax s]
[u]	[uw]	tulip	[ˈtulɨp]	[t uw l ix p]
[ʌ]	[uh]	buttercup	[ˈbʌɾɚˌkʌp]	[b uh dx axr k uh p]
[ɝ]	[er]	bird	[ˈbɝd]	[b er d]
[aɪ]	[ay]	iris	[ˈaɪrɨs]	[ay r ix s]
[aʊ]	[aw]	sunflower	[ˈsʌnflaʊɚ]	[s ah n f l aw axr]
[oɪ]	[oy]	poinsettia	[poɪnˈsɛɾiə]	[p oy n s eh dx iy ax]
[ju]	[y uw]	feverfew	[fivɚfju]	[f iy v axr f y u]
[ə]	[ax]	woodruff	[ˈwʊdrəf]	[w uh d r ax f]
[ɨ]	[ix]	tulip	[ˈtulɨp]	[t uw l ix p]
[ɚ]	[axr]	heather	[ˈhɛðɚ]	[h eh dh axr]
[ʉ]	[ux]	dude[2]	[dʉd]	[d ux d]

Figure 4.2 IPA and ARPAbet symbols for transcription of English vowels.

Many of the IPA and ARPAbet symbols are equivalent to the Roman letters used in the orthography of English and many other languages. So for example the IPA and ARPAbet symbol [p] represents the consonant sound at

[2] The last phone, [ʉ]/[ux], is quite rare in general American English and indeed is an "extension" not present in the original ARPAbet. Labov (1994) notes that the realization of a fronted [uw] as [ux] has made it more common in (at least) Western and Northern Cities dialects of American English starting in the late 1970s. This fronting was first called to public by imitations and recordings of 'Valley Girls' speech by Moon Zappa (Zappa and Zappa, 1982). Nevertheless, for most speakers [uw] is still much more common than [ux] in words like *dude*.

the beginning of *platypus*, *puma*, and *pachyderm*, the middle of *leopard*, or the end of *antelope* (note that the final orthographic *e* of *antelope* does not correspond to any final vowel; the *p* is the last sound).

The mapping between the letters of English orthography and IPA symbols is rarely as simple as this, however. This is because the mapping between English orthography and pronunciation is quite opaque; a single letter can represent very different sounds in different contexts. Figure 4.3 shows that the English letter *c* is represented as IPA [k] in the word *cougar*, but IPA [s] in the word *civet*. Besides appearing as *c* and *k*, the sound marked as [k] in the IPA can appear as part of *x* (*fox*), as *ck* (*jackal*), and as *cc* (*raccoon*). Many other languages, for example Spanish, are much more transparent in their sound-orthography mapping than English.

Word	jackal	raccoon	cougar	civet
IPA	[ˈdʒæ.kl̩]	[ræ.ˈkun]	[ˈku.gɚ]	[ˈsɪ.vɨt]
ARPAbet	[jh ae k el]	[r ae k uw n]	[k uw g axr]	[s ih v ix t]

Figure 4.3 The mapping between IPA symbols and letters in English orthography is complicated; both IPA [k] and English orthographic [c] have many alternative realizations.

The Vocal Organs

ARTICULATORY PHONETICS

We turn now to **articulatory phonetics**, the study of how phones are produced, as the various organs in the mouth, throat, and nose modify the airflow from the lungs.

Sound is produced by the rapid movement of air. Most sounds in human spoken languages are produced by expelling air from the lungs through the windpipe (technically the **trachea**) and then out the mouth or nose. As it passes through the trachea, the air passes through the **larynx**, commonly known as the Adam's apple or voicebox. The larynx contains two small folds of muscle, the **vocal folds** (often referred to non-technically as the **vocal cords**) which can be moved together or apart. The space between these two folds is called the **glottis**. If the folds are close together (but not tightly closed), they will vibrate as air passes through them; if they are far apart, they won't vibrate. Sounds made with the vocal folds together and vibrating are called **voiced**; sounds made without this vocal cord vibration are called **unvoiced** or **voiceless**. Voiced sounds include [b], [d], [g], [v], [z], and all the English vowels, among others. Unvoiced sounds include [p], [t], [k], [ʃ], [z], and others.

GLOTTIS

VOICED

UNVOICED

VOICELESS

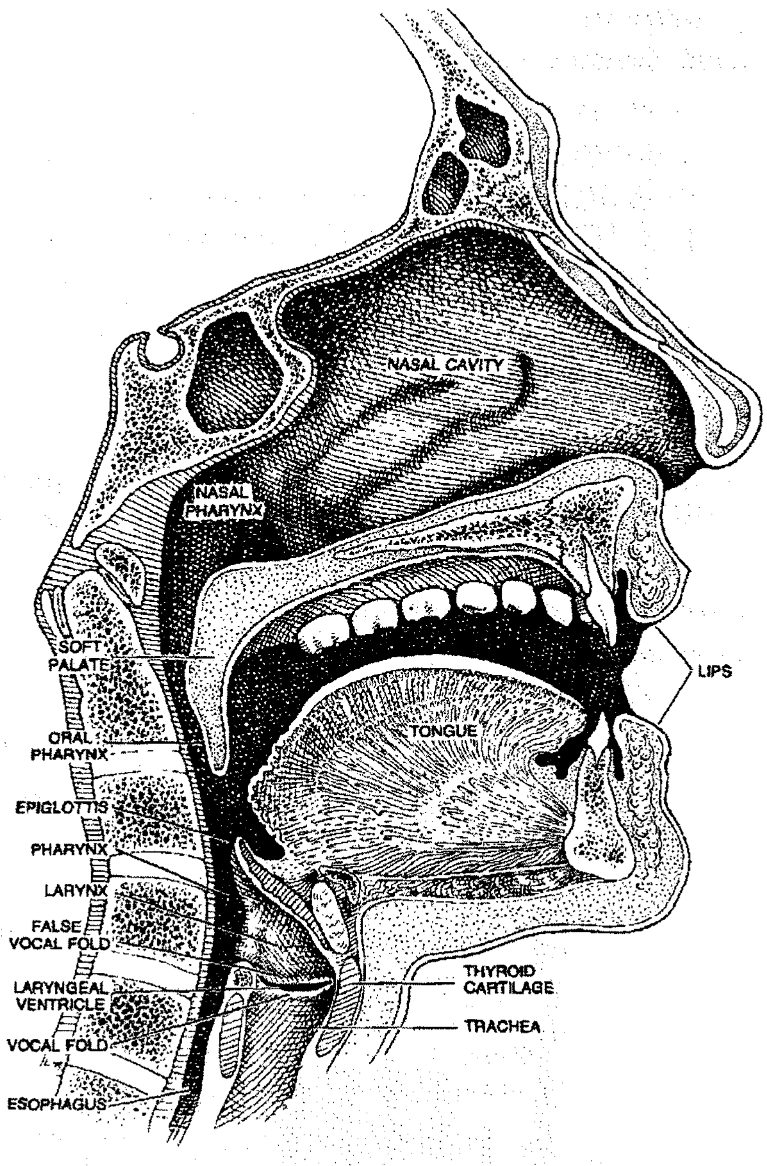

Figure 4.4 The vocal organs, shown in side view. Drawing by Laszlo Kubinyi from Sundberg (1977), ©Scientific American.

The area above the trachea is called the **vocal tract**, and consists of the **oral tract** and the **nasal tract**. After the air leaves the trachea, it can exit the body through the mouth or the nose. Most sounds are made by air passing through the mouth. Sounds made by air passing through the nose are called **nasal sounds**; nasal sounds use both the oral and nasal tracts as resonating cavities; English nasal sounds include *m*, and *n*, and *ng*. NASAL SOUNDS

Phones are divided into two main classes: **consonants** and **vowels**. CONSONANTS Both kinds of sounds are formed by the motion of air through the mouth, VOWELS throat or nose. Consonants are made by restricting or blocking the airflow in some way, and may be voiced or unvoiced. Vowels have less obstruction, are usually voiced, and are generally louder and longer-lasting than consonants. The technical use of these terms is much like the common usage; [p], [b], [t], [d], [k], [g], [f], [v], [s], [z], [r], [l], etc., are consonants; [aa], [ae], [aw], [ao], [ih], [aw], [ow], [uw], etc., are vowels. **Semivowels** (such as [y] and [w]) have some of the properties of both; they are voiced like vowels, but they are short and less syllabic like consonants.

Consonants: Place of Articulation

Because consonants are made by restricting the airflow in some way, consonants can be distinguished by where this restriction is made: the point of maximum restriction is called the **place of articulation** of a consonant. PLACE Places of articulation, shown in Figure 4.5, are often used in automatic speech recognition as a useful way of grouping phones together into equivalence classes:

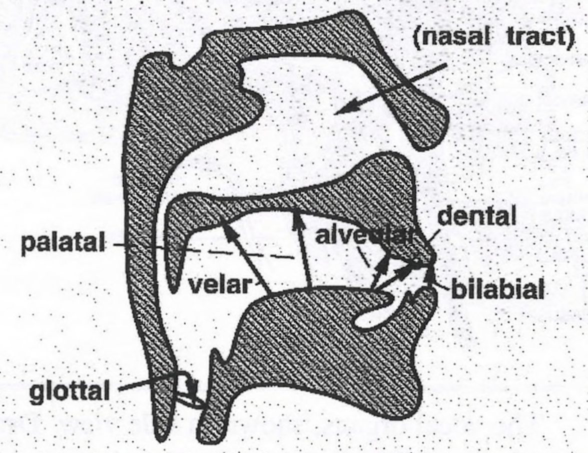

Figure 4.5 Major English places of articulation.

LABIAL

- **labial:** Consonants whose main restriction is formed by the two lips coming together have a **bilabial** place of articulation. In English these include [p] as in *possum*, [b] as in *bear*, and [m] as in *marmot*. The English **labiodental** consonants [v] and [f] are made by pressing the bottom lip against the upper row of teeth and letting the air flow through the space in the upper teeth.

DENTAL

- **dental:** Sounds that are made by placing the tongue against the teeth are dentals. The main dentals in English are the [θ] of *thing* or the [ð] of *though*, which are made by placing the tongue behind the teeth with the tip slightly between the teeth.

- **alveolar:** The alveolar ridge is the portion of the roof of the mouth just behind the upper teeth. Most speakers of American English make the phones [s], [z], [t], and [d] by placing the tip of the tongue against the alveolar ridge. ALVEOLAR
- **palatal:** The roof of the mouth (the **palate**) rises sharply from the back of the alveolar ridge. The **palato-alveolar** sounds [ʃ] (*shrimp*), [tʃ] (*chinchilla*), [ʒ] (*Asian*), and [dʒ] (*jaguar*) are made with the blade of the tongue against this rising back of the alveolar ridge. The palatal sound [y] of *yak* is made by placing the front of the tongue up close to the palate. PALATAL PALATE
- **velar:** The **velum** or soft palate is a movable muscular flap at the very back of the roof of the mouth. The sounds [k] (*cuckoo*), [g] (*goose*), and [ŋ] (*kingfisher*) are made by pressing the back of the tongue up against the velum. VELAR VELUM
- **glottal:** The glottal stop [ʔ] is made by closing the glottis (by bringing the vocal folds together). GLOTTAL

Consonants: Manner of Articulation

Consonants are also distinguished by *how* the restriction in airflow is made, for example whether there is a complete stoppage of air, or only a partial blockage, etc. This feature is called the **manner of articulation** of a consonant. The combination of place and manner of articulation is usually sufficient to uniquely identify a consonant. Here are the major manners of articulation for English consonants: MANNER

- **stop:** A stop is a consonant in which airflow is completely blocked for a short time. This blockage is followed by an explosive sound as the air is released. The period of blockage is called the **closure** and the explosion is called the **release**. English has voiced stops like [b], [d], and [g] as well as unvoiced stops like [p], [t], and [k]. Stops are also called **plosives**. It is possible to use a more narrow (detailed) transcription style to distinctly represent the closure and release parts of a stop, both in ARPAbet and IPA-style transcriptions. For example the closure of a [p], [t], or [k] would be represented as [pcl], [tcl], or [kcl] (respectively) in the ARPAbet, and [p˺], [t˺], or [k˺] (respectively) STOP in IPA style. When this form of narrow transcription is used, the unmarked ARPABET symbols [p], [t], and [k] indicate purely the release of the consonant. We will not be using this narrow transcription style in this chapter.

NASALS

- **nasals:** The nasal sounds [n], [m], and [ŋ] are made by lowering the velum and allowing air to pass into the nasal cavity.

FRICATIVE

- **fricative:** In fricatives, airflow is constricted but not cut off completely. The turbulent airflow that results from the constriction produces a characteristic "hissing" sound. The English labiodental fricatives [f] and [v] are produced by pressing the lower lip against the upper teeth, allowing a restricted airflow between the upper teeth. The dental fricatives [θ] and [ð] allow air to flow around the tongue between the teeth. The alveolar fricatives [s] and [z] are produced with the tongue against the alveolar ridge, forcing air over the edge of the teeth. In the palato-alveolar fricatives [ʃ] and [ʒ] the tongue is at the back of the alveolar ridge forcing air through a groove formed in the tongue. The higher-pitched fricatives (in English [s], [z], [ʃ] and [ʒ]) are called **sibilants**. Stops that are followed immediately by fricatives are called **affricates**; these include English [tʃ] (*chicken*) and [dʒ] (*giraffe*).

SIBILANTS

APPROXIMANT

- **approximant:** In approximants, the two articulators are close together but not close enough to cause turbulent airflow. In English [y] (*yellow*), the tongue moves close to the roof of the mouth but not close enough to cause the turbulence that would characterize a fricative. In English [w] (*wormwood*), the back of the tongue comes close to the velum. American [r] can be formed in at least two ways; with just the tip of the tongue extended and close to the palate or with the whole tongue bunched up near the palate. [l] is formed with the tip of the tongue up against the alveolar ridge or the teeth, with one or both sides of the tongue lowered to allow air to flow over it. [l] is called a **lateral** sound because of the drop in the sides of the tongue.

TAP
FLAP

- **tap:** A tap or **flap** [ɾ] is a quick motion of the tongue against the alveolar ridge. The consonant in the middle of the word *lotus* ([loʊɾəs]) is a tap in most dialects of American English; speakers of many British dialects would use a [t] instead of a tap in this word.

Vowels

Like consonants, vowels can be characterized by the position of the articulators as they are made. The two most relevant parameters for vowels are what is called vowel **height**, which correlates roughly with the location of the highest part of the tongue, and the shape of the lips (rounded or not). Figure 4.6 shows the position of the tongue for different vowels.

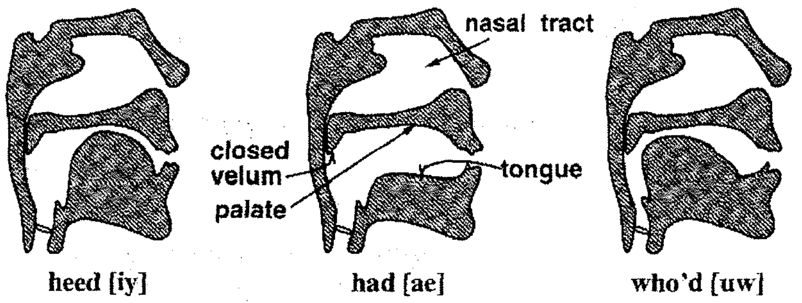

Figure 4.6 Positions of the tongue for three English vowels, high front [iy], low front [ae] and high back [uw]; tongue positions modeled after Ladefoged (1996).

In the vowel [i], for example, the highest point of the tongue is toward the front of the mouth. In the vowel [u], by contrast, the high-point of the tongue is located toward the back of the mouth. Vowels in which the tongue is raised toward the front are called **front vowels**; those in which the tongue is raised toward the back are called **back vowels**. Note that while both [ɪ] and [ɛ] are front vowels, the tongue is higher for [ɪ] than for [ɛ]. Vowels in which the highest point of the tongue is comparatively high are called **high vowels**; vowels with mid or low values of maximum tongue height are called **mid vowels** or **low vowels**, respectively. FRONT BACK HIGH

Figure 4.7 shows a schematic characterization of the vowel height of different vowels. It is schematic because the abstract property **height** only correlates roughly with actual tongue positions; it is in fact a more accurate reflection of acoustic facts. Note that the chart has two kinds of vowels: those in which tongue height is represented as a point and those in which it is represented as a vector. A vowels in which the tongue position changes markedly during the production of the vowel is **diphthong**. English is particularly rich in diphthongs; many are written with two symbols in the IPA (for example the [eɪ] of *hake* or the [oʊ] of *cobra*). DIPHTHONG

The second important articulatory dimension for vowels is the shape of the lips. Certain vowels are pronounced with the lips rounded (the same lip shape used for whistling). These **rounded** vowels include [u], [ɔ], and the diphthong [oʊ]. ROUNDED

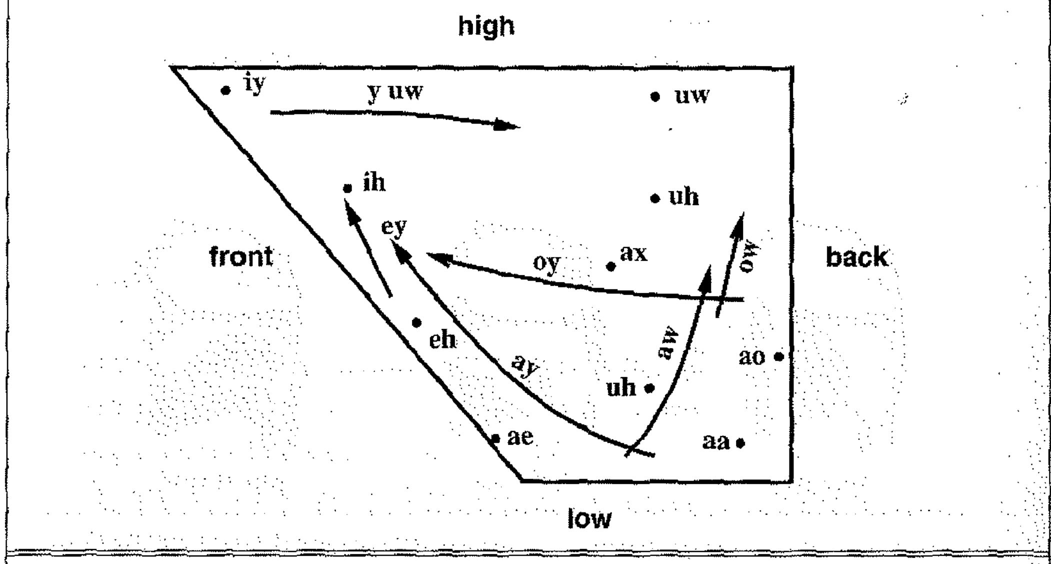

Figure 4.7 Qualities of English vowels (after Ladefoged (1993)).

Syllables

SYLLABLE

Consonants and vowels combine to make a **syllable**. There is no completely agreed-upon definition of a syllable; roughly speaking a syllable is a vowel-like sound together with some of the surrounding consonants that are most closely associated with it. The IPA period symbol [.] is used to separate syllables, so *parsley* and *catnip* have two syllables (['pɑr.sli] and ['kæt.nɪp] respectively), *tarragon* has three ['tæ.rə.gɑn], and *dill* has one ([dɪl]). A syllable is usually described as having an optional initial consonant or set of consonants called the **onset**, followed by a vowel or vowels, followed by a final consonant or sequence of consonants called the **coda**. Thus d is the onset of [dɪl], while l is the coda. The task of breaking up a word into syllables is called **syllabification**. Although automatic syllabification algorithms exist, the problem is hard, partly because there is no agreed-upon definition of syllable boundaries. Furthermore, although it is usually clear how many syllables are in a word, Ladefoged (1993) points out there are some words (meal, teal, seal, hire, fire, hour) that can be viewed either as having one syllable or two.

ONSET

CODA

SYLLABIFICATION

In a natural sentence of American English, certain syllables are more **prominent** than others. These are called **accented** syllables. Accented syllables may be prominent because they are louder, they are longer, they are associated with a pitch movement, or any combination of the above. Since accent plays important roles in meaning, understanding exactly why a speaker

ACCENTED

chooses to accent a particular syllable is very complex. But one important factor in accent is often represented in pronunciation dictionaries. This factor is called **lexical stress**. The syllable that has lexical stress is the one that will be louder or longer if the word is accented. For example the word *parsley* is stressed in its first syllable, not its second. Thus if the word *parsley* is accented in a sentence, it is the first syllable that will be stronger. We write the symbol [ˈ] before a syllable to indicate that it has lexical stress (e.g. [ˈpar.sli]). This difference in lexical stress can affect the meaning of a word. For example the word *content* can be a noun or an adjective. When pronounced in isolation the two senses are pronounced differently since they have different stressed syllables (the noun is pronounced [ˈkɑn.tɛnt]) and the adjective [kən.ˈtɛnt]. Other pairs like this include *object* (noun [ˈɑb.dʒɛkt] and verb [əb.ˈdʒɛkt]); see Cutler (1986) for more examples. Automatic disambiguation of such **homographs** is discussed in Chapter 17. The role of prosody is taken up again in Section 4.7.

LEXICAL STRESS

HOMOGRAPHS

4.2 THE PHONEME AND PHONOLOGICAL RULES

'Scuse me, while I kiss the sky
Jimi Hendrix, *Purple Haze*
'Scuse me, while I kiss this guy
Common mis-hearing of same lyrics

All [t]s are not created equally. That is, phones are often produced differently in different contexts. For example, consider the different pronunciations of [t] in the words *tunafish* and *starfish*. The [t] of *tunafish* is **aspirated**. Aspiration is a period of voicelessness after a stop closure and before the onset of voicing of the following vowel. Since the vocal cords are not vibrating, aspiration sounds like a puff of air after the [t] and before the vowel. By contrast, a [t] following an initial [s] is **unaspirated**; thus the [t] in *starfish* ([stɑrfɪʃ]) has no period of voicelessness after the [t] closure. This variation in the realization of [t] is predictable: whenever a [t] begins a word or unreduced syllable in English, it is aspirated. The same variation occurs for [k]; the [k] of *sky* is often mis-heard as [g] in Jimi Hendrix's lyrics because [k] and [g] are both unaspirated. In a very detailed transcription system we could use the symbol for aspiration [^{h}]after any [t] (or [k] or [p]) which begins a word or unreduced syllable. The word *tunafish* would be transcribed [t^{h}unəfɪʃ] (the ARPAbet does not have a way of marking aspiration).

UNASPIRATED

There are other contextual variants of [t]. For example, when [t] occurs between two vowels, particularly when the first is stressed, it is pronounced as a tap. Recall that a tap is a voiced sound in which the top of the tongue is curled up and back and struck quickly against the alveolar ridge. Thus the word *buttercup* is usually pronounced [bʌɾɚkʌp]/[b uh dx axr k uh p] rather than [bʌtɚkʌp]/[b uh t axr k uh p].

Another variant of [t] occurs before the dental consonant [θ]. Here the [t] becomes dentalized ([t̪]). That is, instead of the tongue forming a closure against the alveolar ridge, the tongue touches the back of the teeth.

PHONEME

ALLOPHONES

How do we represent this relation between a [t] and its different realizations in different contexts? We generally capture this kind of pronunciation variation by positing an abstract class called the **phoneme**, which is realized as different **allophones** in different contexts. We traditionally write phonemes inside slashes. So in the above examples, /t/ is a phoneme whose allophones include [t^h], [ɾ], and [t̪]. A phoneme is thus a kind of generalization or abstraction over different phonetic realizations. Often we equate the phonemic and the lexical levels, thinking of the lexicon as containing transcriptions cxpressed in terms of phonemes. When we are transcribing the pronunciations of words we can choose to represent them at this broad phonemic level; such a **broad transcription** leaves out a lot of predictable phonetic detail. We can also choose to use a **narrow transcription** that includes more detail, including allophonic variation, and uses the various diacritics. Figure 4.8 summarizes a number of allophones of /t/; Figure 4.9 shows a few of the most commonly used IPA diacritics.

NARROW TRANSCRIPTION

Phone	Environment	Example	IPA
[t^h]	in initial position	*toucan*	[t^{h}uk^{h}æn]
[t]	after [s] or in reduced syllables	*starfish*	[stɑrfɨʃ]
[ʔ]	word-finally or after vowel before [n]	*kitten*	[k^{h}ɪʔn]
[ʔt]	sometimes word-finally	*cat*	[k^{h}æʔt]
[ɾ]	between vowels	*buttercup*	[bʌɾɚk^{h}ʌp]
[t̚]	before consonants or word-finally	*fruitcake*	[frut̚k^{h}eɪk]
[t̪]	before dental consonants ([θ])	*eighth*	[eɪt̪θ]
▯	sometimes word-finally	*past*	[pæs]

Figure 4.8 Some allophones of /t/ in General American English.

The relationship between a phoneme and its allophones is often captured by writing a **phonological rule**. Here is the phonological rule for dentalization in the traditional notation of Chomsky and Halle (1968):

$$/\mathrm{t}/ \rightarrow [\text{t̪}] \,/\, \_\_\_\, \theta \qquad (4.1)$$

In this notation, the surface allophone appears to the right of the arrow, and the phonetic environment is indicated by the symbols surrounding the underbar (___). These rules resemble the rules of two-level morphology of Chapter 3 but since they don't use multiple types of rewrite arrows, this rule is ambiguous between an obligatory or optional rule. Here is a version of the flapping rule:

$$/\left\{\begin{matrix} t \\ d \end{matrix}\right\}/ \rightarrow [\text{ɾ}] \,/\, \acute{\mathrm{V}} \,\_\_\_\, \mathrm{V} \qquad (4.2)$$

Diacritics			Suprasegmentals		
̥	Voiceless	[ḁ]	ˈ	Primary stress	[ˈpu.mə]
ʰ	Aspirated	[pʰ]	ˌ	Secondary stress	[ˈfouɾəˌgræf]
̩	Syllabic	[l̩]	ː	Long	[aː]
̃	Nasalized	[æ̃]	ˑ	Half long	[aˑ]
̚	Unreleased	[t̚]	.	Syllable break	[ˈpu.mə]
̪	Dental	[t̪]			

Figure 4.9 Some of the IPA diacritics and symbols for suprasegmentals.

4.3 PHONOLOGICAL RULES AND TRANSDUCERS

Chapter 3 showed that spelling rules can be implemented by transducers. Phonological rules can be implemented as transducers in the same way; indeed the original work by Johnson (1972) and Kaplan and Kay (1981) on finite-state models was based on phonological rules rather than spelling rules. There are a number of different models of **computational phonology** that use finite automata in various ways to realize phonological rules. We will describe the **two-level morphology** of Koskenniemi (1983) used in Chapter 3, but the interested reader should be aware of other recent models.[3] While Chapter 3 gave examples of two-level rules, it did not talk about the

[3] One example is Bird and Ellison's (1994) model of the multi-tier representations of autosegmental phonology in which each phonological tier is represented by a finite-state automaton, and autosegmental association by the synchronization of two automata.

motivation for these rules, and the differences between traditional ordered rules and two-level rules. We will begin with this comparison.

As a first example, Figure 4.10 shows a transducer which models the application of the simplified flapping rule in (4.3):

$$/\text{t}/ \rightarrow [\text{ɾ}] \,/\, \acute{\text{V}} \_\_ \text{V} \tag{4.3}$$

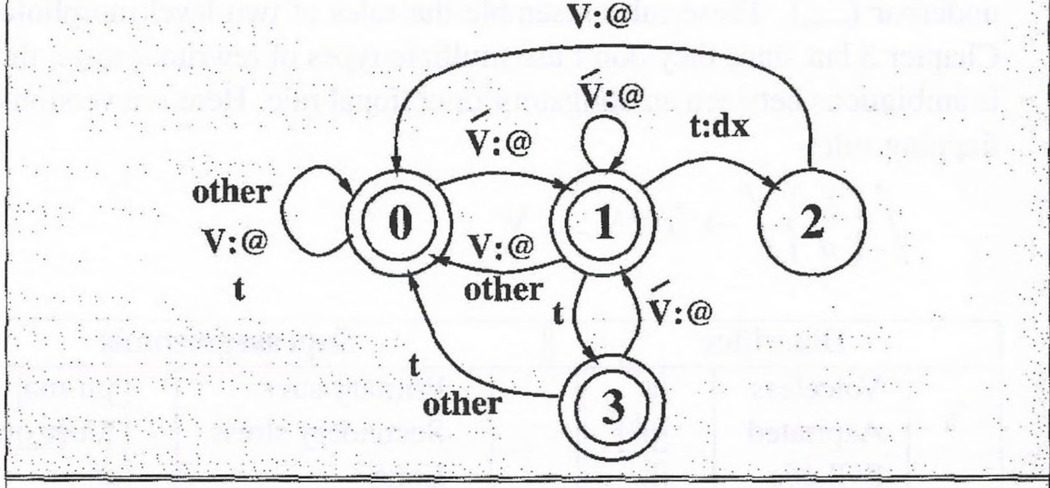

Figure 4.10 Transducer for English Flapping: ARPAbet "dx" indicates a flap, and the "other" symbol means "any feasible pair not used elsewhere in the transducer". "@" means "any symbol not used elsewhere on any arc".

The transducer in Figure 4.10 accepts any string in which flaps occur in the correct places (after a stressed vowel, before an unstressed vowel), and rejects strings in which flapping doesn't occur, or in which flapping occurs in the wrong environment. Of course the factors that flapping are actually a good deal more complicated, as we will see in Section 5.7.

In a traditional phonological system, many different phonological rules apply between the lexical form and the surface form. Sometimes these rules interact; the output from one rule affects the input to another rule. One way to implement rule-interaction in a transducer system is to run transducers in a *cascade*. Consider, for example, the rules that are needed to deal with the phonological behavior of the English noun plural suffix *-s*. This suffix is pronounced [ɨz] after the phones [s], [ʃ], [z], or [ʒ] (so *peaches* is pronounced [pitʃɨz], and *faxes* is pronounced [fæksɨz]), [z] after voiced sounds (*pigs* is pronounced [pɪgz]), and [s] after unvoiced sounds (*cats* is pronounced [kæts]). We model this variation by writing phonological rules for the realization of the morpheme in different contexts. We first need to choose one of these three forms (s, z, and ɨz) as the "lexical" pronunciation of the suffix; we chose z only because it turns out to simplify rule writing. Next we write two phonological rules. One, similar to the E-insertion spelling rule of page 77, inserts a [ɨ] after a morpheme-final sibilant and before the plural morpheme [z]. The other makes sure that the -*s* suffix is properly realized as [s] after unvoiced consonants.

$$\varepsilon \rightarrow \text{ɨ} \,/\, [\text{+sibilant}]\ \hat{}\ \_\_\ \text{z}\ \# \qquad (4.4)$$

$$\text{z} \rightarrow \text{s} \,/\, [\text{-voice}]\ \hat{}\ \_\_\ \# \qquad (4.5)$$

These two rules must be *ordered*; rule (4.4) must apply before (4.5). This is because the environment of (4.4) includes z, and the rule (4.5) changes z. Consider running both rules on the lexical form *fox* concatenated with the plural -*s*:

Lexical form:	faks^z
(4.4) applies:	faks^ɨz
(4.5) doesn't apply:	faks^ɨz

If the devoicing rule (4.5) was ordered first, we would get the wrong result (what would this incorrect result be?). This situation, in which one rule destroys the environment for another, is called **bleeding**:[4]

Lexical form:	faks^z
(4.5) applies:	faks^s
(4.4) doesn't apply:	faks^s

As was suggested in Chapter 3, each of these rules can be represented by a transducer. Since the rules are ordered, the transducers would also need to be ordered. For example if they are placed in a **cascade**, the output of the first transducer would feed the input of the second transducer.

Many rules can be cascaded together this way. As Chapter 3 discussed, running a cascade, particularly one with many levels, can be unwieldy, and so transducer cascades are usually replaced with a single more complex transducer by **composing** the individual transducers.

Koskenniemi's method of **two-level morphology** that was sketchily introduced in Chapter 3 is another way to solve the problem of rule ordering. Koskenniemi (1983) observed that most phonological rules in a grammar are independent of one another; that feeding and bleeding relations between

[4] If we had chosen to represent the lexical pronunciation of -*s* as [s] rather than [z], we would have written the rule inversely to voice the -*s* after voiced sounds, but the rules would still need to be ordered; the ordering would simply flip.

rules are not the norm.[5] Since this is the case, Koskenniemi proposed that phonological rules be run in parallel rather than in series. The cases where there is rule interaction (feeding or bleeding) we deal with by slightly modifying some rules. Koskenniemi's two-level rules can be thought of as a way of expressing **declarative constraints** on the well-formedness of the lexical-surface mapping.

Two-level rules also differ from traditional phonological rules by explicitly coding when they are obligatory or optional, by using four differing **rule operators**; the $\Leftrightarrow$ rule corresponds to traditional **obligatory** phonological rules, while the $\Rightarrow$ rule implements **optional rules**:

Rule type	Interpretation
a : b $\Leftarrow$ c ___ d	*a* is **always** realized as *b* in the context *c* ___ *d*
a : b $\Rightarrow$ c ___ d	*a* may be realized as *b* **only** in the context *c* ___ *d*
a : b $\Leftrightarrow$ c ___ d	*a* must be realized as *b* in context *c* ___ *d* and nowhere else
a : b /$\Leftarrow$ c ___ d	*a* is **never** realized as *b* in the context *c* ___ *d*

The most important intuition of the two-level rules, and the mechanism that lets them avoiding feeding and bleeding, is their ability to represent constraints on *two levels*. This is based on the use of the colon (":"), which was touched in very briefly in Chapter 3. The symbol *a:b* means a lexical *a* that maps to a surface *b*. Thus *a:b* $\Leftrightarrow$ *:c* ___ means *a* is realized as *b* after a **surface** c. By contrast *a:b* $\Leftrightarrow$ *c:* ___ means that *a* is realized as *b* after a **lexical** c. As discussed in Chapter 3, the symbol *c* with no colon is equivalent to *c:c* that means a lexical *c* which maps to a surface *c*.

Figure 4.11 shows an intuition for how the two-level approach avoids ordering for the ɨ-insertion and z-devoicing rules. The idea is that the z-devoicing rule maps a *lexical* z-insertion to a *surface* s and the ɨ rule refers to the *lexical* z:

The two-level rules that model this constraint are shown in (4.6) and (4.7):

$$\varepsilon : \text{ɨ} \ \Leftrightarrow\ [\text{+sibilant}]\text{: } \hat{}\ \_\_\_\ \text{z: } \# \qquad (4.6)$$

$$\text{z : s} \ \Leftrightarrow\ [\text{-voice}]\text{: } \hat{}\ \_\_\_\ \# \qquad (4.7)$$

As Chapter 3 discussed, there are compilation algorithms for creating automata from rules. Kaplan and Kay (1994) give the general derivation of these algorithms, and Antworth (1990) gives one that is specific to two-level rules. The automata corresponding to the two rules are shown in Figure 4.12

[5] Feeding is a situation in which one rules creates the environment for another rule and so must be run beforehand.

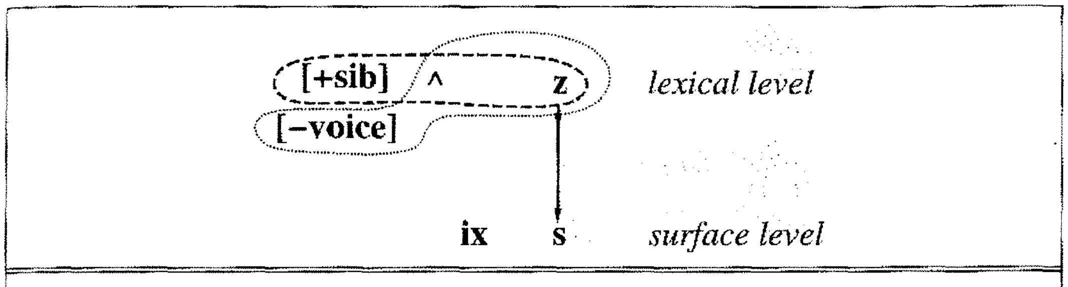

Figure 4.11 The constraints for the ɨ-insertion and z-devoicing rules both refer to a *lexical* z, not a *surface* s.

and Figure 4.13. Figure 4.12 is based on Figure 3.14 of Chapter 3; see page 78 for a reminder of how this automaton works. Note in Figure 4.12 that the plural morpheme is represented by z:, indicating that the constraint is expressed about an lexical rather than surface z.

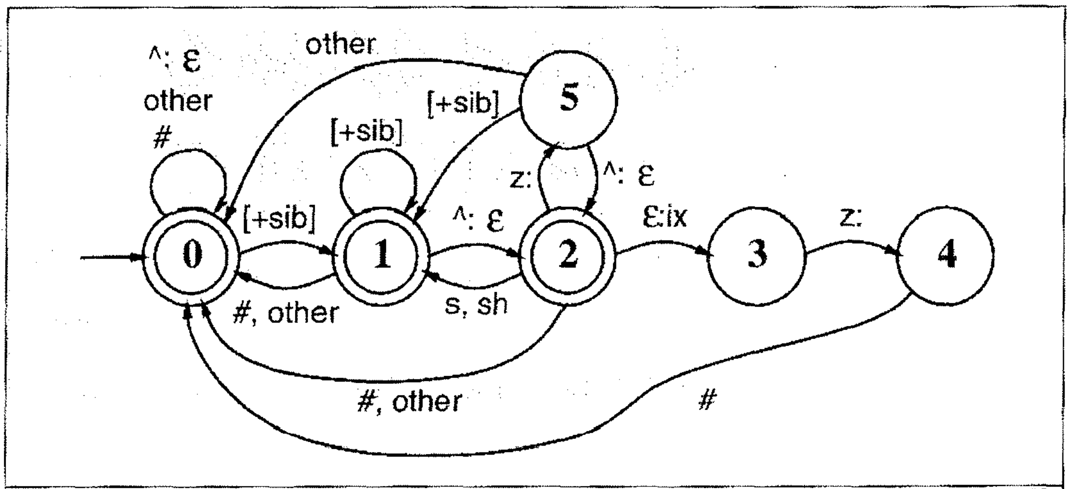

Figure 4.12 The transducer for the ɨ-insertion rule 4.4. The rule can be read *whenever a morpheme ends in a sibilant, and the following morpheme is z, insert* [ɨ].

Figure 4.14 shows the two automata run in parallel on the input [fɑks^z] (the figure uses the ARPAbet notation [f aa k s ^ z]). Note that both the automata assuming the default mapping ^:ε to remove the morpheme boundary, and that both automata end in an accepting state.

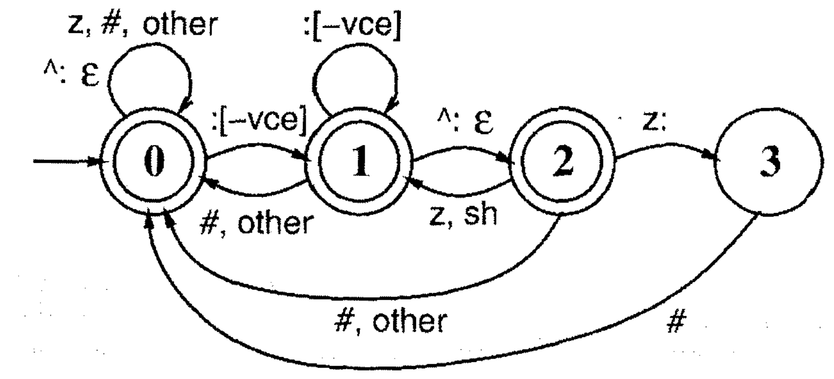

Figure 4.13 The transducer for the z-devoicing rule 4.5. This rule might be summarized *Devoice the morpheme* z *if it follows a morpheme-final voiceless consonant.*

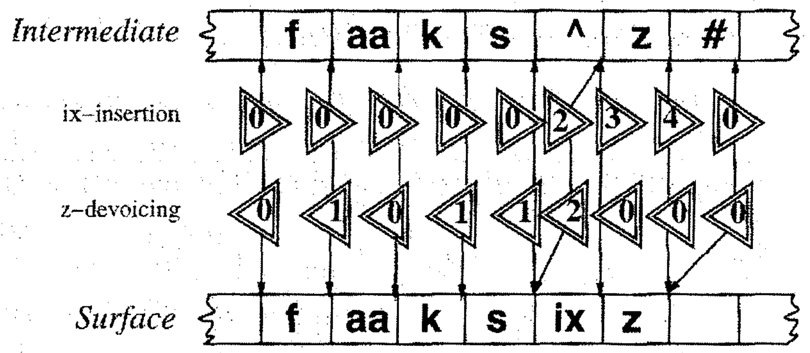

Figure 4.14 The transducer for the i-insertion rule 4.4 and the z-devoicing rule 4.5 run in parallel.

4.4 Advanced Issues in Computational Phonology

Harmony

Rules like flapping, i-insertion, and z-devoicing are relatively simple as phonological rules go. In this section we turn to the use of the two-level or finite-state model of phonology to model more sophisticated phenomena; this section will be easier to follow if the reader has some knowledge of phonology. The Yawelmani dialect of Yokuts is a Native American language spoken in California with a complex phonological system. In particular, there are three phonological rules related to the realization of vowels that had to be ordered in traditional phonology and whose interaction thus demonstrates a complicated use of finite-state phonology. These rules were first drawn up in the traditional Chomsky and Halle (1968) format by Kisseberth (1969) following the field work of Newman (1944).

First, Yokuts (like many other languages including for example Turkish and Hungarian) has a phonological phenomenon called **vowel harmony**. Vowel harmony is a process in which a vowel changes its form to look like a neighboring vowel. In Yokuts, a suffix vowel changes its form to agree in backness and roundness with the preceding stem vowel. That is, a front vowel like /i/ will appear as a backvowel [u] if the stem vowel is /u/ (examples are taken from Cole and Kisseberth (1995):[6]

VOWEL HARMONY

Lexical		Surface	Gloss
dub+hin	→	dubhun	"tangles, non-future"
xil+hin	→	xilhin	"leads by the hand, non-future"
bok'+al	→	bok'ol	"might eat"
xat'+al	→	xat'al	"might find"

This Harmony rule has another constraint: it only applies if the suffix vowel and the stem vowel are of the same height. Thus /u/ and /i/ are both high, while /o/ and /a/ are both low.

The second relevant rule, Lowering, causes long high vowels to become low; thus /u:/ becomes [o:] in the first example below:

Lexical		Surface	Gloss
?u:t'+it	→	?o:t'ut	"steal, passive aorist"
mi:k'+it	→	me:k'+it	"swallow, passive aorist"

The third rule, Shortening, shortens long vowels if they occur in closed syllables:

Lexical		Surface
s:ap+hin	→	saphin
sudu:k+hin	→	sudokhun

The Yokuts rules must be ordered, just as the i-insertion and z-devoicing rules had to be ordered. Harmony must be ordered before Lowering because the /u:/ in the lexical form /?u:t'+it/ causes the /i/ to become [u] before it lowers in the surface form [?o:t'ut]. Lowering must be ordered before Shortening because the /u:/ in /sudu:k+hin/ lowers to [o]; if it was ordered after shortening it would appear on the surface as [u].

Goldsmith (1993) and Lakoff (1993) independently observed that the Yokuts data could be modeled by something like a transducer; Karttunen

[6] For purposes of simplifying the explanation, this account ignores some parts of the system such as vowel underspecification (Archangeli, 1984).

(1998) extended the argument, showing that the Goldsmith and Lakoff constraints could be represented either as a cascade of three rules in series, or in the two-level formalism as three rules in parallel; Figure 4.15 shows the two architectures. Just as in the two-level examples presented earlier, the rules work by referring sometimes to the lexical context, sometimes to the surface context; writing the rules is left as Exercise 4.10 for the reader.

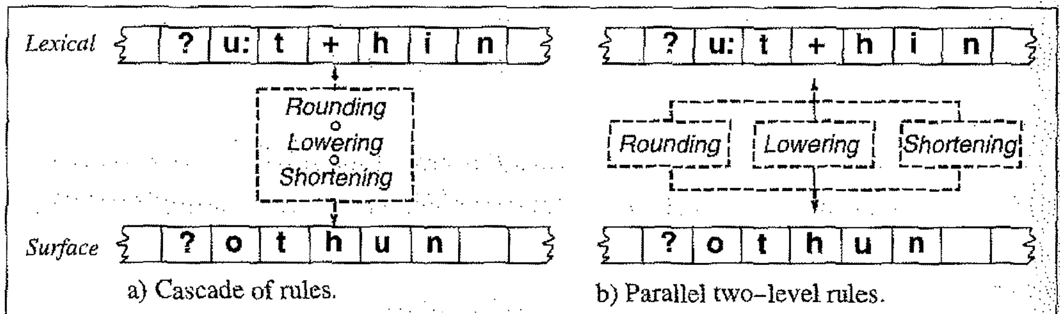

Figure 4.15 Combining the rounding, lowering, and shortening rules for Yawelmani Yokuts.

Templatic Morphology

Finite-state models of phonology/morphology have also been proposed for the templatic (non-concatenative) morphology (discussed on page 60) common in Semitic languages like Arabic, Hebrew, and Syriac. McCarthy (1981) proposed that this kind of morphology could be modeled by using different levels of representation that Goldsmith (1976) had called **tiers**. Kay (1987) proposed a computational model of these tiers via a special transducer which reads four tapes instead of two, as in Figure 4.16.

TIERS

The tricky part here is designing a machine which aligns the various strings on the tapes in the correct way; Kay proposed that the binyan tape could act as a sort of guide for alignment. Kay's intuition has led to a number of more fully worked out finite-state models of Semitic morphology such as Beesley's (1996) model for Arabic and Kiraz's (1997) model for Syriac.

The more recent work of Kornai (1991) and Bird and Ellison (1994) showed how one-tape automata (i.e. finite-state automata rather than four-tape or even two-tape transducers) could be used to model templatic morphology and other kinds of phenomena that are handleed with the tier-based **autosegmental** representations of Goldsmith (1976).

AUTOSEGMENTAL

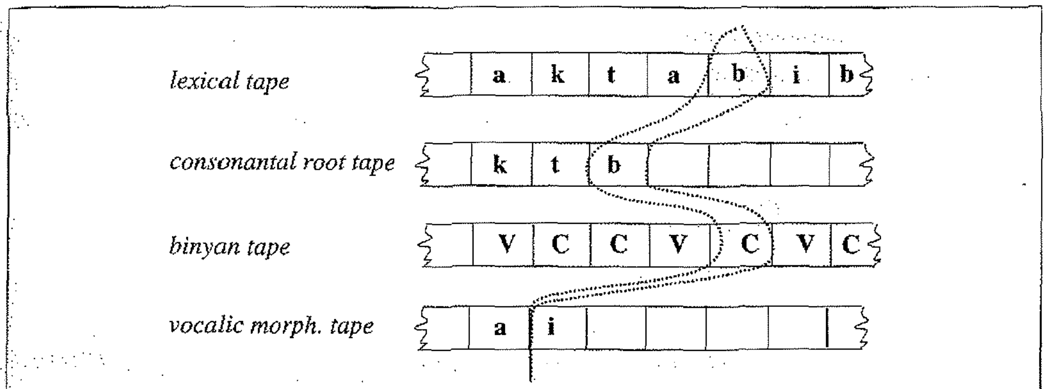

Figure 4.16 A finite-state model of templatic ("non-concatenative") morphology. Modified from Kay (1987) and Sproat (1993).

Optimality Theory

In a traditional phonological derivation, we are given an underlying lexical form and a surface form. The phonological system then consists of one component: a sequence of rules which map the underlying form to the surface form. **Optimality Theory (OT)** (Prince and Smolensky, 1993) offers an alternative way of viewing phonological derivation, based on two functions (GEN and EVAL) and a set of ranked violable constraints (CON). Given an underlying form, the GEN function produces all imaginable surface forms, even those which couldn't possibly be a legal surface form for the input. The EVAL function then applies each constraint in CON to these surface forms in order of constraint rank. The surface form which best meets the constraints is chosen.

OPTIMALITY THEORY

OT

A constraint in OT represents a wellformedness constraint on the surface form, such as a phonotactic constraint on what segments can follow each other, or a constraint on what syllable structures are allowed. A constraint can also check how **faithful** the surface form is to the underlying form.

FAITHFUL

Let's turn to our favorite complicated language, Yawelmani, for an example.[7] In addition to the interesting vowel harmony phenomena discussed above, Yawelmani has a phonotactic constraints that rules out sequences of consonants. In particular three consonants in a row (CCC) are not allowed to occur in a surface word. Sometimes, however, a word contains two consecutive morphemes such that the first one ends in two consonants and the second one starts with one consonant (or vice versa). What does the lan-

[7] The following explication of OT via the Yawelmani example draws heavily from Archangeli (1997) and a lecture by Jennifer Cole at the 1999 LSA Linguistic Institute.

guage do to solve this problem? It turns out that Yawelmani either deletes one of the consonants or inserts a vowel in between.

For example, if a stem ends in a C, and its suffix starts with CC, the first C of the suffix is deleted ("+" here means a morpheme boundary):

C-deletion $C \rightarrow \varepsilon / C + \_\_ C$ (4.8)

Here is an example where the CCVC "passive consequent adjunctive" morpheme hne:l (actually the underlying form is /hnil/) drops the initial C if the previous morpheme ends in two consonants (and an example where it doesn't, for comparison):

underlying morphemes	gloss
diyel-ne:l-aw	"guard - passive consequent adjunctive - locative"
cawa-hne:l-aw	"shout - passive consequent adjunctive - locative"

If a stem ends in CC and the suffix starts with C, the language instead inserts a vowel to break up the first two consonants:

V-insertion $\varepsilon \rightarrow V / C \_\_ C + C$ (4.9)

Here are some examples in which an i is inserted into the roots ʔilk- "sing" and the roots logw- "pulverize" only when they are followed by a C-initial suffix like -hin, "past", not a V-initial suffix like -en, "future":

surface form	gloss
ʔilik-hin	"sang"
ʔilken	"will sing"
logiwhin	"pulverized"
logwen	"will pulverize"

Kisseberth (1970) suggested that it was not a coincidence that Yawelmani had these particular two rules (and for that matter other related deletion rules that we haven't presented). He noticed that these rules were functionally related; in particular, they all are ways of avoiding three consonants in a row. Another way of stating this generalization is to talk about syllable structure. Yawelmani syllables are only allowed to be of the form CVC or CV (where C means a consonant and V means a vowel). We say that languages like Yawelmani don't allow **complex onsets** or **complex codas**. From the point of view of syllabification, then, these insertions and deletions all happen so as to allow Yawelmani words to be properly syllabified. Since CVCC syllables aren't allowed on the surface, CVCC roots must be **resyllabified** when they appear on the surface. For example, here are the syllabifications

COMPLEX ONSET
COMPLEX CODA
RESYLLABIFIED

of the Yawelmani words we have discussed and some others; note, for example, that the surface syllabification of the CVCC syllables moves the final consonant to the beginning of the next syllable:

underlying morphemes	surface syllabification	gloss
ʔilk-en	ʔil.ken	"will sing"
logw-en	log.wen	"will pulverize"
logw-hin	lo.giw.hin	"will pulverize"
xat-en	xa.ten	"will eat"
diyel-hnil-aw	di.yel.neː.law	"ask - pass. cons. adjunct. - locative"

Here's where Optimality Theory comes in. The basic idea in Optimality Theory is that the language has various constraints on things like syllable structure. These constraints generally apply to the surface form. One such constraint, *COMPLEX, says "No complex onsets or codas". Another class of constraints requires the surface form to be identical to (faithful to) the underlying form. Thus FAITHV says "Don't delete or insert vowels" and FAITHC says "Don't delete or insert consonants". Given an underlying form, the GEN function produces all possible surface forms (i.e., every possible insertion and deletion of segments with every possible syllabification) and they are ranked by the EVAL function using these constraints. Figure 4.17 shows the architecture.

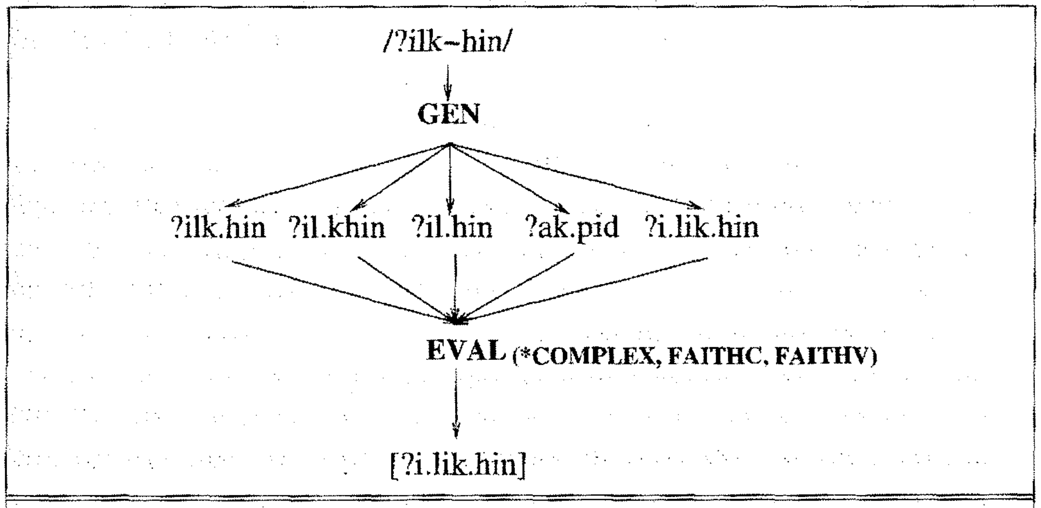

Figure 4.17 The architecture of a derivation in Optimality Theory (after Archangeli (1997)).

The EVAL function works by applying each constraint in ranked order; the optimal candidate is one which either violates no constraints, or violates less of them than all the other candidates. This evaluation is usually shown on a **tableau** (plural **tableaux**). The top left-hand cell shows the input, the constraints are listed in order of rank across the top row, and the possible outputs along the left-most column. Although there are an infinite number of candidates, it is traditional to show only the ones which are 'close'; in the tableau below we have shown the output ?ak.pid just to make it clear that even very different surface forms are to be included. If a form violates a constraint, the relevant cell contains *; a !* indicates the fatal violation which causes a candidate to be eliminated. Cells for constraints which are irrelevant (since a higher-level constraint is already violated) are shaded.

TABLEAU

/ʔilk-hin/	*COMPLEX	FAITHC	FAITHV
ʔilk.hin	*!	▨	▨
ʔil.khin	*!	▨	▨
ʔil.hin		*!	▨
☞ ʔi.lik.hin			*
ʔak.pid		*!	▨

One appeal of Optimality Theoretic derivations is that the constraints are presumed to be cross-linguistic generalizations. That is all languages are presumed to have some version of faithfulness, some preference for simple syllables, and so on. Languages differ in how they rank the constraints; thus English, presumably, ranks FAITHC higher than *COMPLEX. (How do we know this?)

Can a derivation in Optimality Theory be implemented by finite-state transducers? Frank and Satta (1999), following the foundational work of Ellison (1994), showed that (1) if GEN is a regular relation (for example assuming the input doesn't contain context-free trees of some sort), and (2) if the number of allowed violations of any constraint has some finite bound, then an OT derivation can be computed by finite-state means. This second constraint is relevant because of a property of OT that we haven't mentioned: if two candidates violate exactly the same number of constraints, the winning candidate is the one which has the smallest number of violations of the relevant constraint.

One way to implement OT as a finite-state system was worked out by Karttunen (1998), following the above-mentioned work and that of Hammond (1997). In Karttunen's model, GEN is implemented as a finite-state transducer which is given an underlying form and produces a set of candidate forms. For example for the syllabification example above, GEN would generate all strings that are variants of the input with consonant deletions or vowel insertions, and their syllabifications.

Each constraint is implemented as a filter transducer that lets pass only strings which meet the constraint. For legal strings, the transducer thus acts as the identity mapping. For example, *COMPLEX would be implemented via a transducer that mapped any input string to itself, unless the input string had two consonants in the onset or coda, in which case it would be mapped to null.

The constraints can then be placed in a cascade, in which higher-ranked constraints are simply run first, as suggested in Figure 4.18.

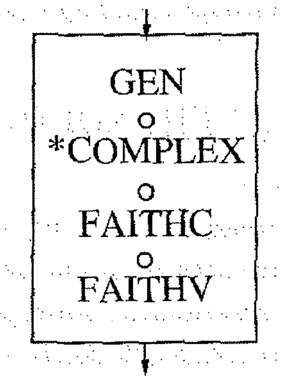

Figure 4.18 Version #1 ("merciless cascade") of Karttunen's finite-state cascade implementation of OT.

There is one crucial flaw with the cascade model in Figure 4.18. Recall that the constraints-transducers filter out any candidate which violates a constraint. But in many derivations, include the proper derivation of ?i.lik.hin, even the optimal form still violates a constraint. The cascade in Figure 4.17 would incorrectly filter it out, leaving no surface form at all! Frank and Satta (1999) and Hammond (1997) both point out that it is essential to only enforce a constraint if it does not reduce the candidate set to zero. Karttunen (1998) formalizes this intuition with the **lenient composition** operator. Lenient composition is a combination of regular composition and an operation called **priority union**. The basic idea is that if any candidates meet the constraint these candidates will be passed through the filter as usual. If no output meets the constraint, lenient composition retains *all* of the candidates. Figure 4.19 shows the general idea; the interested reader should see Karttunen (1998) for the details. Also see Tesar (1995, 1996), Fosler (1996), and Eisner (1997) for discussions of other computational issues in OT.

LENIENT COMPOSITION

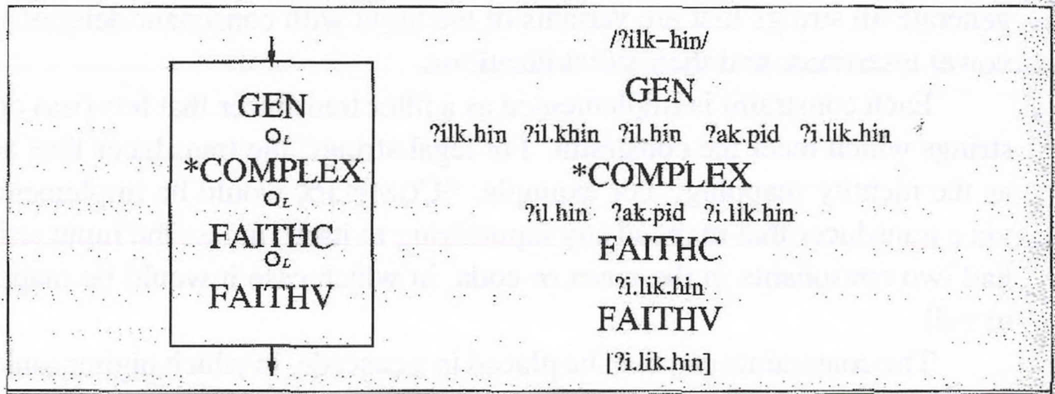

Figure 4.19 Version #2 ("lenient cascade") of Karttunen's finite-state cascade implementation of OT, showing a visualization of the candidate populations that would be passed through each FST constraint.

4.5 Machine Learning of Phonological Rules

MACHINE LEARNING

The task of a **machine learning** system is to automatically induce a model for some domain, given some data from the domain and, sometimes, other information as well. Thus a system to learn phonological rules would be given at least a set of (surface forms of) words to induce from. A **supervised** algorithm is one which is given the correct answers for some of this data, using these answers to induce a model which can generalize to new data it hasn't seen before. An **unsupervised** algorithm does this purely from the data. While unsupervised algorithms don't get to see the correct labels for the classifications, they can be given hints about the nature of the rules or models they should be forming. For example, the knowledge that the models will be in the form of automata is itself a kind of hint. Such hints are called a **learning bias**.

SUPERVISED

UNSUPERVISED

LEARNING BIAS

This section gives a very brief overview of some models of unsupervised machine learning of phonological rules; more details about machine learning algorithms will be presented throughout the book.

Ellison (1992) showed that concepts like the consonant and vowel distinction, the syllable structure of a language, and harmony relationships could be learned by a system based on choosing the model from the set of potential models which is the simplest. Simplicity can be measured by choosing the model with the minimum coding length, or the highest probability (we will define these terms in detail in Chapter 6). Daelemans et al. (1994) used the Instance-Based Generalization algorithm (Aha et al., 1991) to learn stress rule for Dutch; the algorithm is a supervised one which is given a number of words together with their stress patterns, and which induces generalizations about the mapping from the sequences of light and heavy syllable type in the word (light syllables have no coda consonant; heavy syllables have one) to the stress pattern. Tesar and Smolensky (1993) show that a system which is given Optimality Theory constraints but not their ranking can learn the ranking from data via a simple greedy algorithm.

Johnson (1984) gives one of the first computational algorithms for phonological rule induction. His algorithm works for rules of the form

(4.10) $a \rightarrow b/C$

where C is the feature matrix of the segments around a. Johnson's algorithm sets up a system of constraint equations which C must satisfy, by considering both the positive contexts, i.e., all the contexts C_i in which a b occurs on the surface, as well as all the negative contexts C_j in which an a occurs on the surface. Touretzky et al. (1990) extended Johnson's insight by using the *version spaces* algorithm of Mitchell (1981) to induce phonological rules in their *Many Maps* architecture, which is similar to two-level phonology. Like Johnson's, their system looks at the underlying and surface realizations of single segments. For each segment, the system uses the version space algorithm to search for the proper statement of the context. The model also has a separate algorithm which handles harmonic effects by looking for multiple segmental changes in the same word, and is more general than Johnson's in dealing with epenthesis and deletion rules.

The algorithm of Gildea and Jurafsky (1996) was designed to induce transducers representing two-level rules of the type we have discussed earlier. Like the algorithm of Touretzky et al. (1990), Gildea and Jurafsky's algorithm was given sets of pairings of underlying and surface forms. The algorithm was based on the OSTIA (Oncina et al., 1993) algorithm, which is a general learning algorithm for a subtype of finite-state transducers called **subsequential transducers**. By itself, the OSTIA algorithm was too general to learn phonological transducers, even given a large corpus of underlying-form/surface-form pairs. Gildea and Jurafsky then augmented the domain-independent OSTIA system with three kinds of learning biases which are specific to natural language phonology; the main two are **Faithfulness** (underlying segments tend to be realized similarly on the surface), and **Community** (similar segments behave similarly). The resulting system was able to learn transducers for flapping in American English, or German consonant devoicing.

Finally, many learning algorithms for phonology are probabilistic. For example Riley (1991) and Withgott and Chen (1993) proposed a decision-tree approach to segmental mapping. A decision tree is induced for each segment, classifying possible realizations of the segment in terms of contextual factors such as stress and the surrounding segments. Decision trees and probabilistic algorithms in general will be defined in Chapters 5 and 6.

4.6 Mapping Text to Phones for TTS

Dearest creature in Creation
Studying English pronunciation
I will teach you in my verse
Sounds like corpse, corps, horse and worse.
It will keep you, Susy, busy,
Make your head with heat grow dizzy
...
River, rival; tomb, bomb, comb;
Doll and roll, and some and home.
Stranger does not rime with anger
Neither does devour with clangour.
...

G.N. Trenite (1870-1946) *The Chaos*, reprinted in Witten (1982).

Now that we have learned the basic inventory of phones in English and seen how to model phonological rules, we are ready to study the problem of mapping from an orthographic or text word to its pronunciation.

Pronunciation Dictionaries

An important component of this mapping is a **pronunciation dictionary**. These dictionaries are actually used in both ASR and TTS systems, although because of the different needs of these two areas the contents of the dictionaries are somewhat different.

The simplest pronunciation dictionaries just have a list of words and their pronunciations:

Word	Pronunciation	Word	Pronunciation
cat	[kæt]	goose	[gus]
cats	[kæts]	geese	[gis]
pig	[pɪg]	hedgehog	[ˈhɛdʒ.hɔg]
pigs	[pɪgz]	hedgehogs	[ˈhɛdʒ.hɔgz]
fox	[fɑx]		
foxes	[ˈfɑk.sɪz]		

Three large, commonly-used, on-line pronunciation dictionaries in this format are PRONLEX, CMUdict, and CELEX. These are used for speech recognition and can also be adapted for use in speech synthesis. The PRONLEX dictionary (LDC, 1995) was designed for speech recognition applications and contains pronunciations for 90,694 wordforms. It covers all the words used in many years of the Wall Street Journal, as well as the Switchboard Corpus. The CMU Pronouncing Dictionary was also developed for ASR purposes and has pronunciations for about 100,000 wordforms. The CELEX dictionary (Celex, 1993) includes all the words in the Oxford Advanced Learner's Dictionary (1974) (41,000 lemmata) and the Longman Dictionary of Contemporary English (1978) (53,000 lemmata), in total it has pronunciations for 160,595 wordforms. Its pronunciations are British while the other two are American. Each dictionary uses a different phone set; the CMU and PRONLEX phonesets are derived from the ARPAbet, while the CELEX dictionary is derived from the IPA. All three represent three levels of stress: primary stress, secondary stress, and no stress. Figure 4.20 shows the pronunciation of the word *armadillo* in all three dictionaries.

Dictionary	Pronunciation	IPA Version
Pronlex	+arm.xd'Il.o	[ˌɑrməˈdɪloʊ]
CMU	AA2 R M AH0 D IH1 L OW0	[ˌɑrmʌ̲ˈdɪlo̲ʊ̲]
CELEX	"#-m@-'dI-15	[ˌɑː.mə.ˈdɪ.ləʊ]

Figure 4.20 The pronunciation of the word *armadillo* in three dictionaries. Rather than explain special symbols, we have given an IPA equivalent for each pronunciation. The CMU dictionary represents unstressed vowels ([ə], [ɨ], etc.) by giving a 0 stress level to the vowel. We represented this by underlining in the IPA form. Note the r-dropping and use of the [əʊ] rather than [oʊ] vowel in the British CELEX pronunciation.

Often two distinct words are spelled the same (they are **homographs**) but pronounced differently. For example the verb *wind* ("You need to wind this up more neatly") is pronounced [waɪnd] while the noun *wind* ("blow,

blow, thou winter wind") is pronounced [wɪnd]. This is essential for TTS applications (since in a given context the system needs to say one or the other) but for some reason is usually ignored in current speech recognition systems. Printed pronunciation dictionaries give distinct pronunciations for each part-of-speech; CELEX does as well. Since they were designed for ASR, Pronlex and CMU, although they give two pronunciations for the form *wind*, don't specify which one is used for which part-of-speech.

Dictionaries often don't include many proper names. This is a serious problem for many applications; Liberman and Church (1992) report that 21% of the word tokens in their 33-million-word 1988 AP newswire corpus were names. Furthermore, they report that a list obtained in 1987 from the Donnelly marketing organization contains 1.5 million names (covering 72 million households in the United States). But only about 1000 of the 52477 lemmas in CELEX (which is based on traditional dictionaries) are proper names. By contrast Pronlex includes 20,000 names; this is still only a small fraction of the 1.5 million. Very few dictionaries give pronunciations for entries like *Dr.*, which as Liberman and Church (1992) point out can be "doctor" or "drive", or *2/3*, which can be "two thirds" or "February third" or "two slash three".

No dictionaries currently have good models for the pronunciation of function words (*and*, *I*, *a*, *the*, *of*, etc.). This is because the variation in these words due to phonetic context is so great. Usually the dictionaries include some simple baseform (such as [ði] for *the*) and use other algorithms to derive the variation due to context; Chapter 5 will treat the issue of modeling contextual pronunciation variation for words of this sort.

One significant difference between TTS and ASR dictionaries is that TTS dictionaries do not have to represent dialectal variation; thus where a very accurate ASR dictionary needs to represent both pronunciations of *either* and *tomato*, a TTS dictionary can choose one.

Beyond Dictionary Lookup: Text Analysis

Mapping from text to phones relies on the kind of pronunciation dictionaries we talked about in the last section. As we suggested before, one way to map text-to-phones would be to look up each word in a pronunciation dictionary and read the string of phones out of the dictionary. This method would work fine for any word that we can put in the dictionary in advance. But as we saw in Chapter 3, it's not possible to represent every word in English (or any other language) in advance. Both speech synthesis and speech recognition systems need to be able to guess at the pronunciation of words that are not in their dictionary. This section will first examine the kinds of words that are likely to be missing in a pronunciation dictionary, and then show how the finite-state transducers of Chapter 3 can be used to model the basic task of text-to-phones. Chapter 5 will introduce variation in pronunciation and introduce probabilistic techniques for modeling it.

Three of the most important cases where we cannot rely on a word dictionary involve **names**, **morphological productivity**, and **numbers**. As a brief example, we arbitrarily selected a brief (561 word) movie review that appeared in the July 17, 1998 issue of the New York Times. The review, of Vincent Gallo's "Buffalo '66", was written by Janet Maslin. Here's the beginning of the article:

> In Vincent Gallo's "Buffalo '66," Billy Brown (Gallo) steals a blond kewpie doll named Layla (Christina Ricci) out of her tap dancing class and browbeats her into masquerading as his wife at a dinner with his parents. Billy hectors, cajoles and tries to bribe Layla. ("You can eat all the food you want. Just make me look good.") He threatens both that he will kill her and that he won't be her best friend. He bullies her outrageously but with such crazy brio and jittery persistence that Layla falls for him. Gallo's film, a deadpan original mixing pathos with bravado, works on its audience in much the same way.

We then took two large commonly-used on-line pronunciation dictionaries; the PRONLEX dictionary, that contains pronunciations for 90,694 wordforms and includes coverage of many years of the Wall Street Journal, as well as the Switchboard Corpus, and the larger CELEX dictionary, which has pronunciations for 160,595 wordforms. The combined dictionaries have approximately 194,000 pronunciations. Of the 561 words in the movie review, 16 (3%) did not have pronunciations in these two dictionaries (not counting two hyphenated words, *baby-blue* and *hollow-eyed*). Here they are:

Names		**Inflected Names**	**Numbers**	**Other**
Aki	Gazzara	Gallo's	'66	c'mere
Anjelica	Kaurismaki			indie
Arquette	Kusturica			kewpie
Buscemi	Layla			sexpot
Gallo	Rosanna			

Some of these missing words can be found by increasing the dictionary size (for example Wells's (1990) definitive (but not on-line) pronunciation dictionary of English does have *sexpot* and *kewpie*). But the rest need to generated on-line.

Names are a large problem for pronunciation dictionaries. It is difficult or impossible to list in advance all proper names in English; furthermore they may come from any language, and may have variable spellings. Most potential applications for TTS or ASR involve names; for example names are essentially in telephony applications (directory assistance, call routing). Corporate names are important in many applications and are created constantly (*CoComp, Intel, Cisco*). Medical speech applications (such as transcriptions of doctor-patient interviews) require pronunciations of names of pharmaceuticals; there are some off-line medical pronunciation dictionaries but they are known to be extremely inaccurate (Markey and Ward, 1997). Recall the figure of 1.5 million names mentioned above, and Liberman and Church's (1992) finding that 21% of the word tokens in their 33 million word 1988 AP newswire corpus were names.

Morphology is a particular problem for many languages other than English. For languages with very productive morphology it is computationally infeasible to represent every possible word; recall this Turkish example:

(4.11) uygarlaştıramadıklarımızdanmışsınızcasına

uygar	*+laş*	*+tır*	*+ama*	*+dık*	*+lar*	*+ımız*
civilized	+BEC	+CAUS	+NEGABLE	+PPART	+PL	+P1PL

+dan	*+mış*	*+sınız*	*+casına*
+ABL	+PAST	+2PL	+AsIf

"(behaving) as if you are among those whom we could not civilize/cause to become civilized"

Even a language as similar to English as German has greater ability to create words; Sproat et al. (1998) note the spontaneously created German example *Unerfindlichkeitsunterstellung* ("allegation of incomprehensibility").

But even in English, morphologically simple though it is, morphological knowledge is necessary for pronunciation modeling. For example names and acronyms are often inflected (*Gallo's*, *IBM's*, *DATs*, *Syntex's*) as are new words (*faxes*, *indies*). Furthermore, we can't just add *s* to the pronunciation of the uninflected forms, because as the last section showed, the possessive *-'s* and plural *-s* suffix in English are pronounced differently in different contexts; *Syntex's* is pronounced [sɪntɛksɨz], *faxes* is pronounced [fæksɨz], *IBM's* is pronounced [aɪbijɛmz], and *DATs* is pronounced [dæts].

Finally, pronouncing numbers is a particularly difficult problem. The *'66* in *Buffalo '66* is pronounced [sɪkstisɪks] not [sɪkssɪks]. The most natural way to pronounce the phone number "947-2020" is probably "nine"-"four"-"seven"-"twenty"-"twenty" rather than "nine"-"four"-"seven"-"two"-"zero"-"two"-"zero". Liberman and Church (1992) note that there are five main ways to pronounce a string of digits (although others are possible):

- **Serial:** Each digit is pronounced separately—*8765* is "eight seven six five".
- **Combined:** The digit string is pronounced as a single integer, with all position labels read out—"eight thousand seven hundred sixty five".
- **Paired:** Each pair of digits is pronounced as an integer; if there is an odd number of digits the first one is pronounced by itself—"eighty-seven sixty-five".
- **Hundreds:** Strings of four digits can be pronounced as counts of hundreds—"eighty-seven hundred (and) sixty-five".
- **Trailing Unit:** Strings that end in zeros are pronounced serially until the last nonzero digit, which is pronounced followed by the appropriate unit—*8765000* is "eight seven six five thousand".

Pronunciation of numbers and these five methods are discussed further in Exercises 4.5 and 4.6.

An FST-based Pronunciation Lexicon

Early work in pronunciation modeling for text-to-speech systems (such as the seminal MITalk system Allen et al. (1987)) relied heavily on **letter-to-sound** rules. Each rule specified how a letter or combination of letters was mapped to phones; here is a fragment of such a rule-base from Witten (1982):

LETTER-TO-SOUND

Fragment	Pronunciation
-p-	[p]
-ph-	[f]
-phe-	[fi]
-phes-	[fiz]
-place-	[pleɪs]
-placi-	[pleɪsi]
-plement-	[plɪmɛnt]

Such systems consisted of a long list of such rules and a very small dictionary of exceptions (often function words such as *a, are, as, both, do, does,* etc.). More recent systems have completely inverted the algorithm, relying on very large dictionaries, with letter-to-sound rules only used for the small number of words that are neither in the dictionary nor are morphological variants of words in the dictionary. How can these large dictionaries be represented in a way that allows for morphological productivity? Luckily, these morphological issues in pronunciation (adding inflectional suffixes, slight pronunciation changes at the juncture of two morphemes, etc.) are identical to the morphological issues in spelling that we saw in Chapter 3. Indeed, (Sproat, 1998b) and colleagues have worked out the use of transducers for text-to-speech. We might break down their transducer approach into five components:

1. an FST to represent the pronunciation of individual words and morphemes in the lexicon
2. FSAs to represent the possible sequencing of morphemes
3. individual FSTs for each pronunciation rule (for example expressing the pronunciation of *-s* in different contexts)
4. heuristics and letter-to-sound (LTS) rules/transducers used to model the pronunciations of names and acronyms
5. default letter-to-sound rules/transducers for any other unknown words

We will limit our discussion here to the first four components; those interested in letter-to-sound rules should see (Allen et al., 1987). These first components will turn out to be simple extensions of the FST components we saw in Chapter 3 and on page 110. The first is the representation of the lexical base form of each word; recall that base form means the uninflected form of the word. The previous base forms were stored in orthographic representation; we will need to augment each of them with the correct lexical phonological representation. Figure 4.21 shows the original and the updated lexical entries:

The second part of our FST system is the finite-state machinery to model morphology. We will give only one example: the nominal plural suffix *-s*. Figure 4.22 in Chapter 3 shows the automaton for English plurals, updated to handle pronunciation as well. The only change was the addition of the [s] pronunciation for the suffix, and ε pronunciations for all the morphological features.

We can compose the inflection FSA in Figure 4.22 with a transducer implementing the baseform lexicon in Figure 4.21 to produce an inflectionally-enriched lexicon that has singular and plural nouns. The resulting mini-lexicon is shown in Figure 4.23.

The lexicon shown in Figure 4.23 has two levels, an underlying or "lexical" level and an intermediate level. The only thing that remains is to add

Orthographic Lexicon	**Lexicon**
Regular Nouns	
cat	c|k a|æ t|t
fox	f|f o|ɑ x|ks
dog	d|d o|ɑ g|g
Irregular Singular Nouns	
goose	g|g oo|u s|s e|ε
Irregular Plural Nouns	
g o:e o:e s e	g|g oo|u:ee|i s|s e|ε

Figure 4.21 FST-based lexicon, extending the lexicon in the table on page 74 in Chapter 3. Each symbol in the lexicon is now a pair of symbols separated by "|", one representing the "orthographic" lexical entry and one the "phonological" lexical entry. The irregular plural *geese* also pre-specifies the contents of the intermediate tape ":ee|i".

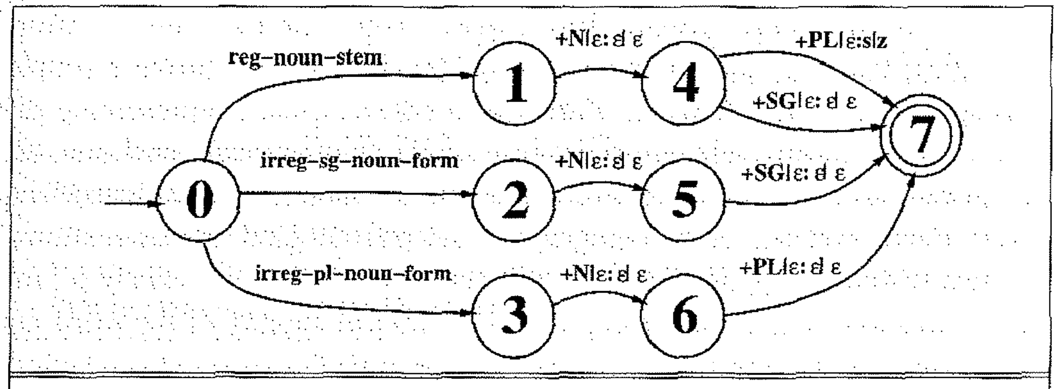

Figure 4.22 FST for the nominal singular and plural inflection. The automaton adds the morphological features [+N], [+PL], and [+SG] at the lexical level where relevant and also adds the plural suffix *s*|z (at the intermediate level). We will discuss below why we represent the pronunciation of *-s* as z rather than s.

transducers which apply spelling rules and pronunciation rules to map the intermediate level into the surface level. These include the various spelling rules discussed on page 77 and the pronunciation rules starting on page 105.

The lexicon and these phonological rules and the orthographic rules from Chapter 3 can now be used to map between a lexical representation (containing both orthographic and phonological strings) and a surface representation (containing both orthographic and phonological strings). As we saw in Chapter 3, this mapping can be run from surface to lexical form, or from lexical to surface form; Figure 4.24 shows the architecture. Recall that

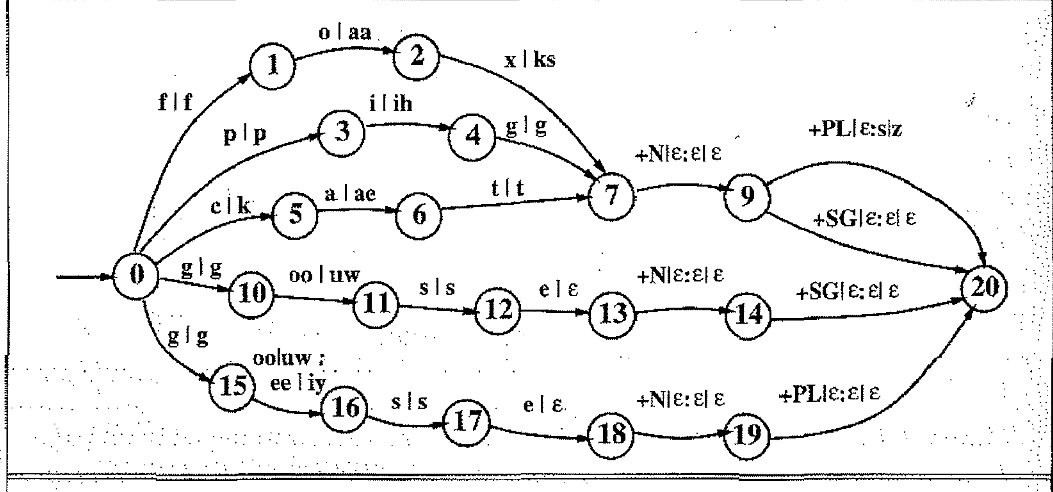

Figure 4.23 Mini-lexicon composing a transducer from the baseform lexicon of Figure 4.21 with the inflectional transducer of Figure 4.22.

the lexicon FST maps between the "lexical" level, with its stems and morphological features, and an "intermediate" level which represents a simple concatenation of morphemes. Then a host of FSTs, each representing either a single spelling rule constraint or a single phonological constraint, all run in parallel so as to map between this intermediate level and the surface level. Each level has both orthographic and phonological representations. For text-to-speech applications in which the input is a lexical form (e.g., for text generation, where the system knows the lexical identity of the word, its part-of-speech, its inflection, etc.), the cascade of FSTs can map from lexical form to surface pronunciation. For text-to-speech applications in which the input is a surface spelling (e.g., for "reading text out loud" applications), the cascade of FSTs can map from surface orthographic form to surface pronunciation via the underlying lexical form.

Finally let us say a few words about names and acronyms. Acronyms can be spelled with or without periods (*I.R.S.* or *IRS*). Acronyms with periods are usually pronounced by spelling them out ([aɪɑrɛs]). Acronyms that usually appear without periods (AIDS, ANSI, ASCAP) may either be spelled out or pronounced as a word; so AIDS is usually pronounced the same as the third-person form of the verb *aid*. Liberman and Church (1992) suggest keeping a small dictionary of the acronyms that are pronounced as words, and spelling out the rest. Their method for dealing with names begins with a dictionary of the pronunciations of 50,000 names, and then applies a small number of affix-stripping rules (akin to the Porter Stemmer of Chapter 3), rhyming heuristics, and letter-to-sound rules to increase the coverage.

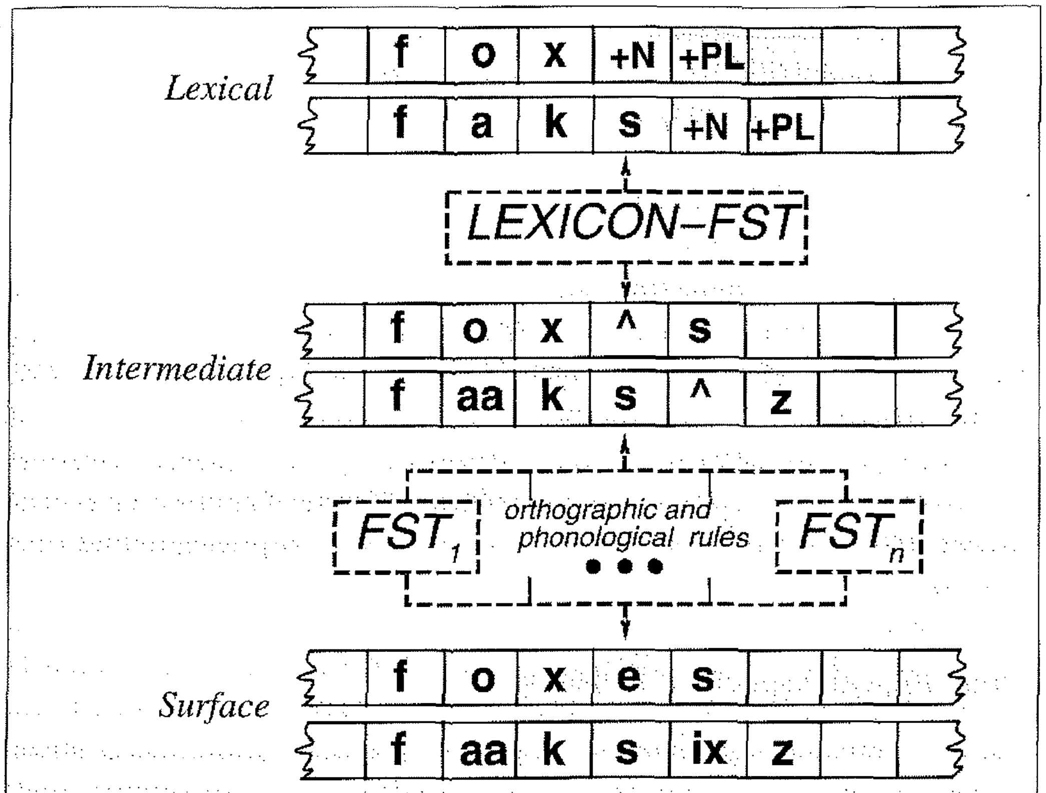

Figure 4.24 Mapping between the lexicon and surface form for orthography and phonology simultaneously. The system can be used to map from a lexical entry to its surface pronunciation or from surface orthography to surface pronunciation via the lexical entry.

Liberman and Church (1992) took the most frequent quarter million words in the Donnelly list. They found that the 50,000 word dictionary covered 59% of these 250,000 name tokens. Adding stress-neutral suffixes like *-s*, *-ville*, and *-son* (*Walters = Walter + s, Abelson = Abel + son, Lucasville = Lucas + ville*) increased the coverage to 84%. Adding name-name compounds (*Abdulhussein, Baumgaertner*) and rhyming heuristics increased the coverage to 89%. The rhyming heuristics used letter-to-sound rules for the beginning of the word and then found a rhyming word to help pronounce the end; so Plotsky was pronounced by using the LTS rule for *Pl-* and guessing *-otsky* from *Trotsky*. They then added a number of more complicated morphological rules (prefixes like *O'Brien*), stress-changing suffixes (*Adamovich*), suffix-exchanges (*Bierstadt = Bierbaum - baum + stadt*) and used a system of letter-to-sound rules for the remainder. This system was not implemented as an FST; Exercise 4.11 will address some of the issues in turning such a set of rules into an FST. Readers interested in further details about names, acronyms and other unknown words should consult sources such as Liberman and Church (1992), Vitale (1991), and Allen et al. (1987).

4.7 Prosody in TTS

The orthography to phone transduction process just described produces the main component for the input to the part of a TTS system which actually generates the speech. Another important part of the input is a specification of the **prosody**. The term **prosody** is generally used to refer to aspects of a sentence's pronunciation which aren't described by the sequence of phones derived from the lexicon. Prosody operates on longer linguistic units than phones, and hence is sometimes called the study of **suprasegmental** phenomena.

PROSODY

SUPRASEGMENTAL

Phonological Aspects of Prosody

There are three main phonological aspects to prosody: **prominence, structure** and **tune**.

PROMINENCE

STRUCTURE

TUNE

As page 102 discussed, prominence is a broad term used to cover **stress** and **accent**. Prominence is a property of syllables, and is often described in a relative manner, by saying one syllable is more prominent than another. Pronunciation lexicons mark lexical stress; for example *table* has its stress on the first syllable, while *machine* has its stress on the second. Function words like *there*, *the* or *a* are usually unaccented altogether. When words are joined together, their accentual patterns combine and form a larger accent pattern for the whole utterance. There are some regularities in how accents combine. For example adjective-noun combinations like *new truck* are likely to have accent on the right word (*new *truck*), while noun-noun compounds like **tree surgeon* are likely to have accent on the left. In generally, however, there are many exceptions to these rules, and so accent prediction is quite complex. For example the noun-noun compound **apple cake* has the accent on the first word while the noun-noun compound *apple *pie* or *city *hall* both have the accent on the second word (Liberman and Sproat, 1992; Sproat, 1994, 1998a). Furthermore, rhythm plays a role in keeping the accented syllables spread apart a bit; thus *city *hall* and **parking lot* combine as **city hall *parking lot* (Liberman and Prince, 1977). Finally, the location of accent is very strongly affected by the discourse factors we will describe in Chapters 18 and 19; in particular new or focused words or phrases often receive accent.

STRESS

ACCENT

Sentences have prosodic structure in the sense that some words seem to group naturally together and some words seem to have a noticeable break or disjuncture between them. Often prosodic structure is described in terms of **prosodic phrasing**, meaning that an utterance has a prosodic phrase structure in a similar way to it having a syntactic phrase structure. For example, in the sentence *I wanted to go to London, but could only get tickets for France* there seems to be two main prosodic phrases, their boundary occurring at the comma. Commonly used terms for these larger prosodic units include **intonational phrase** or **IP** (Beckman and Pierrehumbert, 1986), **intonation unit** (Du Bois et al., 1983), and **tone unit** (Crystal, 1969). Furthermore, in the first phrase, there seems to be another set of lesser prosodic phrase boundaries (often called **intermediate phrases**) that split up the words as follows *I wanted | to go | to London.* The exact definitions of prosodic phrases and subphrases and their relation to syntactic phrases like clauses and noun phrases and semantic units have been and still are the topic of much debate (Chomsky and Halle, 1968; Langendoen, 1975; Streeter, 1978; Hirschberg and Pierrehumbert, 1986; Selkirk, 1986; Nespor and Vogel, 1986; Croft, 1995; Ladd, 1996; Ford and Thompson, 1996; Ford et al., 1996). Despite these complications, algorithms have been proposed which attempt to automatically break an input text sentence into intonational phrases. For example Wang and Hirschberg (1992), Ostendorf and Veilleux (1994), Taylor and Black (1998), and others have built statistical models (incorporating probabilistic predictors such as the CART-style decision trees to be defined in Chapter 5) for predicting intonational phrase boundaries based on such features as the parts of speech of the surrounding words, the length of the utterance in words and seconds, the distance of the potential boundary from the beginning or ending of the utterance, and whether the surrounding words are accented.

PROSODIC PHRASING

INTONATIONAL PHRASE

IP

INTERMEDIATE PHRASE

Two utterances with the same prominence and phrasing patterns can still differ prosodically by having different **tunes**. Tune refers to the intonational melody of an utterance. Consider the utterance *oh, really*. Without varying the phrasing or stress, it is still possible to have many variants of this by varying the intonational tune. For example, we might have an excited version *oh, really!* (in the context of a reply to a statement that you've just won the lottery); a sceptical version *oh, really?*—in the context of not being sure that the speaker is being honest; to an angry *oh, really!* indicating displeasure. Intonational tunes can be broken into component parts, the most important of which is the **pitch accent**. Pitch accents occur on stressed syllables and form a characteristic pattern in the F0 contour (as explained below).

PITCH ACCENT

Depending on the type of pattern, different effects (such as those just outlined above) can be produced. A popular model of pitch accent classification is the Pierrehumbert or ToBI model (Pierrehumbert, 1980; Silverman et al., 1992), which says there are five pitch accents in English, which are made from combining two simple tones (high **H**, and low **L**) in various ways. A **H+L** pattern forms a fall, while a **L+H** pattern forms a rise. An asterisk (*) is also used to indicate which tone falls on the stressed syllable. This gives an inventory of **H***, **L***, **L+H***, **L*+H**, **H+L*** (a sixth pitch accent **H*+L** which was present in early versions of the model was later abandoned). Our three examples of *oh, really* might be marked with the accents **L+H*, L*+H and L*** respectively. In addition to pitch accents, this model also has two phrase accents **L-** and **H-** and two boundary tones **L%** and **H%**, which are used at the ends of phrases to control whether the intonational tune rises or falls.

Other intonational modals differ from ToBI by not using discrete phonemic classes for intonation accents. For example the Tilt (Taylor, 2000) and Fujisaki models (Fujisaki and Ohno, 1997) use continuous parameters rather than discrete categories to model pitch accents. These researchers argue that while the discrete models are often easier to visualize and work with, continuous models may be more robust and more accurate for computational purposes.

Phonetic or Acoustic Aspects of Prosody

The three phonological factors interact and are realized by a number of different phonetic or acoustic phenomena. Prominent syllables are generally louder and longer that non-prominent syllables. Prosodic phrase boundaries are often accompanied by pauses, by lengthening of the syllable just before the boundary, and sometimes lowering of pitch at the boundary. Intonational tune is manifested in the fundamental frequency (F0) contour.

Prosody in Speech Synthesis

A major task for a TTS system is to generate appropriate linguistic representations of prosody, and from them generate appropriate acoustic patterns which will be manifested in the output speech waveform. The output of a TTS system with such a prosodic component is a sequence of phones, each of which has a duration and an F0 (pitch) value. The duration of each phone is dependent on the phonetic context (see Chapter 7). The F0 value is influenced by the factors discussed above, including the lexical stress, the accented or focused element in the sentence, and the intonational tune of the utterance (for example a final rise for questions). Figure 4.25 shows some sample TTS output from the FESTIVAL (Black et al., 1999) speech synthesis system for the sentence *Do you really want to see all of it?*. This output, together with the F0 values shown in Figure 4.26 would be the input to the **waveform synthesis** component described in Chapter 7. The durations here are computed by a CART-style decision tree (Riley, 1992).

				H*												L*		L- H%			
do		you		really				want				to		see		all		of		it	
d	uw	y	uw	r	ih	l	iy	w	aa	n	t	t	ax	s	iy	ao	l	ah	v	ih	t
110	110	50	50	75	64	57	82	57	50	72	41	43	47	54	130	76	90	44	62	46	220

Figure 4.25 Output of the FESTIVAL (Black et al., 1999) generator for the sentence *Do you really want to see all of it?* The exact intonation contour is shown in Figure 4.26. Thanks to Paul Taylor for this figure.

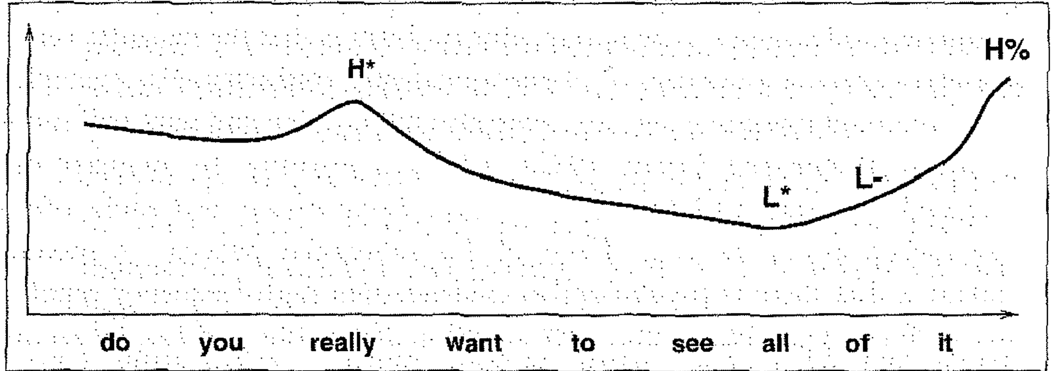

Figure 4.26 The F0 contour for the sample sentence generated by the FESTIVAL synthesis system in Figure 4.25, thanks to Paul Taylor.

As was suggested above, determining the proper prosodic pattern for a sentence is difficult, as real-world knowledge and semantic information is needed to know which syllables to accent, and which tune to apply. This sort of information is difficult to extract from the text and hence prosody modules often aim to produce a "neutral declarative" version of the input text, which assume the sentence should be spoken in a default way with no reference to discourse history or real-world events. This is one of the main reasons why intonation in TTS often sounds "wooden".

4.8 Human Processing of Phonology and Morphology

Chapter 3 suggested that productive morphology plays a psychologically real role in the human lexicon. But we stopped short of a detailed model of how the morphology might be represented. Now that we have studied phonological structure and phonological learning, we return to the psychological question of the representation of morphological/phonological knowledge.

One view of human morphological or phonological processing might be that it distinguishes productive, regular morphology from irregular or exceptional morphology. Under this view, the regular past tense morpheme *-ed*, for example, could be mentally represented as a rule which would be applied to verbs like *walk* to produce *walked*. Irregular past tense verbs like *broke*, *sang*, and *brought*, on the other hand, would simply be stored as part of a lexical representation, and the rule wouldn't apply to these. Thus this proposal strongly distinguishes representation via *rules* from representation via lexical *listing*.

This proposal seems sensible, and is indeed identical to the transducer-based models we have presented in these last two chapters. Unfortunately, this simple model seems to be wrong. One problem is that the irregular verbs

SUBREGULARITY

themselves show a good deal of phonological **subregularity**. For example, the ɪ/æ alternation relating *ring* and *rang* also relates *sing* and *sang* and *swim* and *swam* (Bybee and Slobin, 1982). Children learning the language often extend this pattern to incorrectly produce *bring-brang*, and adults often make speech errors showing effects of this subregular pattern. A second problem is that there is psychological evidence that high-frequency regular inflected forms (*needed, covered*) are stored in the lexicon just like the stems *cover* and *need* (Losiewicz, 1992). Finally, word and morpheme frequency in general seems to play an important role in human processing.

Arguments like these led to "data-driven" models of morphological learning and representation, which essentially store all the inflected forms they have seen. These models generalize to new forms by a kind of analogy; regular morphology is just like subregular morphology but acquires rule-like trappings simply because it occurs more often. Such models include the

CONNECTIONIST
PARALLEL DISTRIBUTED PROCESSING

computational **connectionist** or **Parallel Distributed Processing** model of Rumelhart and McClelland (1986) and subsequent improvements (Plunkett and Marchman, 1991; MacWhinney and Leinbach, 1991) and the similar **network** model of Bybee (1985, 1995). In these models, the behavior of regular morphemes like *-ed* **emerges** from its frequent interaction with other forms. Proponents of the rule-based view of morphology such as Pinker and Prince (1988), Marcus et al. (1995), and others, have criticized the connectionist models and proposed a compromise **dual processing** model, in which regular forms like *-ed* are represent as symbolic rules, but subregular examples (*broke, brought*) are represented by connectionist-style pattern associators. This debate between the connectionist and dual processing models has deep implications for mental representation of all kinds of regular rule-based behavior and is one of the most interesting open questions in human language processing. Chapter 7 will briefly discuss connectionist models of human speech processing; readers who are further interested in connectionist models should consult the references above and textbooks like Anderson (1995).

4.9 SUMMARY

This chapter has introduced many of the important notions we need to understand spoken language processing. The main points are as follows:

- We can represent the pronunciation of words in terms of units called **phones**. The standard system for representing phones is the **International Phonetic Alphabet** or **IPA**. An alternative English-only transcription system that uses ASCII letters is the **ARPAbet**.
- Phones can be described by how they are produced **articulatorily** by the vocal organs; consonants are defined in terms of their **place** and **manner** of articulation and **voicing**, vowels by their **height** and **backness**.
- A **phoneme** is a generalization or abstraction over different phonetic realizations. **Allophonic rules** express how a phoneme is realized in a given context.
- **Transducers** can be used to model phonological rules just as they were used in Chapter 3 to model spelling rules. **Two-level morphology** is a theory of morphology/phonology which models phonological rules as finite-state **well-formedness constraints** on the mapping between lexical and surface form.
- **Pronunciation dictionaries** are used for both text-to-speech and automatic speech recognition. They give the pronunciation of words as strings of phones, sometimes including syllabification and stress. Most on-line pronunciation dictionaries have on the order of 100,000 words but still lack many names, acronyms, and inflected forms.

- The **text-analysis** component of a text-to-speech system maps from orthography to strings of phones. This is usually done with a large dictionary augmented with a system (such as a transducer) for handling productive morphology, pronunciation changes, names, numbers, and acronyms.

Bibliographical and Historical Notes

The major insights of articulatory phonetics date to the linguists of 800–150 B.C. India. They invented the concepts of place and manner of articulation, worked out the glottal mechanism of voicing, and understood the concept of assimilation. European science did not catch up with the Indian phoneticians until over 2000 years later, in the late 19th century. The Greeks did have some rudimentary phonetic knowledge; by the time of Plato's *Theaetetus* and *Cratylus*, for example, they distinguished vowels from consonants, and stop consonants from continuants. The Stoics developed the idea of the syllable and were aware of phonotactic constraints on possible words. An unknown Icelandic scholar of the twelfth century exploited the concept of the phoneme, proposed a phonemic writing system for Icelandic, including diacritics for length and nasality. But his text remained unpublished until 1818 and even then was largely unknown outside Scandinavia (Robins, 1967). The modern era of phonetics is usually said to have begun with Sweet, who proposed what is essentially the phoneme in his *Handbook of Phonetics* (1877). He also devised an alphabet for transcription and distinguished between *broad* and *narrow* transcription, proposing many ideas that were eventually incorporated into the IPA. Sweet was considered the best practicing phonetician of his time; he made the first scientific recordings of languages for phonetic purposes, and advanced the start of the art of articulatory description. He was also infamously difficult to get along with, a trait that is well captured in the stage character that George Bernard Shaw modeled after him: Henry Higgins. The phoneme was first named by the Polish scholar Baudouin de Courtenay, who published his theories in 1894.

The idea that phonological rules could be modeled as regular relations dates to Johnson (1972), who showed that any phonological system that didn't allow rules to apply to their own output (i.e., systems that did not have recursive rules) could be modeled with regular relations (or finite-state transducers). Virtually all phonological rules that had been formulated at the time had this property (except some rules with integral-valued features, like early stress and tone rules). Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Roland Kaplan and Martin Kay; see Chapter 3 for the rest of the history of two-level morphology. Karttunen (1993) gives a tutorial introduction to two-level morphology that includes more of the advanced details than we were able to present here.

Readers interested in phonology should consult (Goldsmith, 1995) as a reference on phonological theory in general and Archangeli and Langendoen (1997) on Optimality Theory.

Two classic text-to-speech synthesis systems are described in Allen et al. (1987) (the *MITalk* system) and Sproat (1998b) (the Bell Labs system). The pronunciation problem in text-to-speech synthesis is an ongoing research area; much of the current research focuses on prosody. Interested readers should consult the proceedings of the main speech engineering conferences: *ICSLP* (the International Conference on Spoken Language Processing), *IEEE ICASSP* (the International Conference on Acoustics, Speech, and Signal Processing), and *EUROSPEECH*.

Students with further interest in transcription and articulatory phonetics should consult an introductory phonetics textbook such as Ladefoged (1993). Pullum and Ladusaw (1996) is a comprehensive guide to each of the symbols and diacritics of the IPA. Many phonetics papers of computational interest are to be found in the *Journal of the Acoustical Society of America (JASA)*, *Computer Speech and Language*, and *Speech Communication*.

Exercises

4.1 Find the mistakes in the IPA transcriptions of the following words:

a. "three" [ðri]
b. "sing" [sɪng]
c. "eyes" [aɪs]
d. "study" [stʊdi]
e. "though" [θoʊ]

f. "planning" [plɑnɪŋ]
g. "slight" [slit]

4.2 Translate the pronunciations of the following color words from the IPA into the ARPAbet (and make a note if you think you pronounce them differently than this!):

a. [rɛd]
b. [blu]
c. [grin]
d. [ˈjɛloʊ]
e. [blæk]
f. [waɪt]
g. [ˈɔrɪndʒ]
h. [ˈpɝpl̩]
i. [pjus]
j. [toʊp]

4.3 Ira Gershwin's lyric for *Let's Call the Whole Thing Off* talks about two pronunciations of the word "either" (in addition to the tomato and potato example given at the beginning of the chapter. Transcribe Ira Gershwin's two pronunciations of "either" in IPA and in the ARPAbet.

4.4 Transcribe the following words in both the ARPAbet and the IPA:

a. dark
b. suit
c. greasy
d. wash
e. water

4.5 Write an FST which correctly pronounces strings of dollar amounts like *$45*, *$320*, and *$4100*. If there are multiple ways to pronounce a number you may pick your favorite way.

4.6 Write an FST which correctly pronounces seven-digit phone numbers like *555-1212*, *555-1300*, and so on. You should use a combination of the **paired** and **trailing unit** methods of pronunciation for the last four digits.

4.7 Build an automaton for rule (4.5).

4.8 One difference between one dialect of Canadian English and most dialects of American English is called **Canadian raising**. Bromberger and Halle (1989) note that some Canadian dialects of English raise /aɪ/ to [ʌɪ] and /aʊ/ to [ʌʊ] in stressed position before a voiceless consonant. A simplified version of the rule dealing only with /aɪ/ can be stated as:

CANADIAN RAISING

$$/\text{aɪ}/ \rightarrow [\text{ʌɪ}] \;/\; \_\_ \begin{bmatrix} C \\ -voice \end{bmatrix} \qquad (4.12)$$

This rule has an interesting interaction with the flapping rule. In some Canadian dialects the word *rider* and *writer* are pronounced differently: *rider* is pronounced [raɪɾɚ] while *writer* is pronounced [rʌɪɾɚ]. Write a two-level rule and an automaton for both the raising rule and the flapping rule which correctly models this distinction. You may make simplifying assumptions as needed.

4.9 Write the lexical entry for the pronunciation of the English past tense (preterite) suffix *-d*, and the two level-rules that express the difference in its pronunciation depending on the previous context. Don't worry about the spelling rules. (Hint: make sure you correctly handle the pronunciation of the past tenses of the words *add*, *pat*, *bake*, and *bag*.)

4.10 Write two-level rules for the Yawelmani Yokuts phenomena of Harmony, Shortening, and Lowering introduced on page 111. Make sure your rules are capable of running in parallel.

4.11 Find 10 stress-neutral name suffixes (look in a phone book) and sketch an FST which would model the pronunciation of names with or without suffixes.

5 PROBABILISTIC MODELS OF PRONUNCIATION AND SPELLING

ALGERNON: *But my own sweet Cecily, I have never written you any letters.*
CECILY: *You need hardly remind me of that, Ernest. I remember only too well that I was forced to write your letters for you. I wrote always three times a week, and sometimes oftener.*
ALGERNON: *Oh, do let me read them, Cecily?*
CECILY: *Oh, I couldn't possibly. They would make you far too conceited. The three you wrote me after I had broken off the engagement are so beautiful, and so badly spelled, that even now I can hardly read them without crying a little.*

Oscar Wilde, *The Importance of being Ernest*

Like Oscar Wilde's fabulous Cecily, a lot of people were thinking about spelling during the last turn of the century. Gilbert and Sullivan provide many examples. *The Gondoliers*' Giuseppe, for example, worries that his private secretary is "shaky in his spelling" while *Iolanthe*'s Phyllis can "spell every word that she uses". Thorstein Veblen's explanation (in his 1899 classic *The Theory of the Leisure Class*) was that a main purpose of the "archaic, cumbrous, and ineffective" English spelling system was to be difficult enough to provide a test of membership in the leisure class. Whatever the social role of spelling, we can certainly agree that many more of us are like Cecily than like Phyllis. Estimates for the frequency of spelling errors in human typed text vary from 0.05% of the words in carefully edited newswire text to 38% in difficult applications like telephone directory lookup (Kukich, 1992).

In this chapter we discuss the problem of detecting and correcting spelling errors and the very related problem of modeling pronunciation variation for automatic speech recognition and text-to-speech systems. On the surface, the problems of finding spelling errors in text and modeling the variable pronunciation of words in spoken language don't seem to have much in common. But the problems turn out to be isomorphic in an important way: they can both be viewed as problems of *probabilistic transduction*. For speech recognition, given a string of symbols representing the pronunciation of a word in context, we need to figure out the string of symbols representing the lexical or dictionary pronunciation, so we can look the word up in the dictionary. But any given surface pronunciation is ambiguous; it might correspond to different possible words. For example the ARPAbet pronunciation [er] could correspond to reduced forms of the words *her*, *were*, *are*, *their*, or *your*. This ambiguity problem is heightened by **pronunciation variation**; for example the word *the* is sometimes pronounced THEE and sometimes THUH; the word *because* sometimes appears as *because*, sometimes as *'cause*. Some aspects of this variation are systematic; Section 5.7 will survey the important kinds of variation in pronunciation that are important for speech recognition and text-to-speech, and present some preliminary rules describing this variation. High-quality speech synthesis algorithms need to know when to use particular pronunciation variants. Solving both speech tasks requires extending the transduction between surface phones and lexical phones discussed in Chapter 4 with probabilistic variation.

Similarly, given the sequence of letters corresponding to a mis-spelled word, we need to produce an ordered list of possible correct words. For example the sequence *acress* might be a mis-spelling of *actress*, or of *cress*, or of *acres*. We transduce from the "surface" form *acress* to the various possible "lexical" forms, assigning each with a probability; we then select the most probable correct word.

In this chapter we first introduce the problems of detecting and correcting spelling errors, and also summarize typical human spelling error patterns. We then introduce the essential probabilistic architecture that we will use to solve both spelling and pronunciation problems: the **Bayes Rule** and the **noisy channel model**. The Bayes rule and its application to the noisy channel model will play a role in many problems throughout the book, particularly in speech recognition (Chapter 7), part-of-speech tagging (Chapter 8), and probabilistic parsing (Chapter 12).

The Bayes Rule and the noisy channel model provide the probabilistic framework for these problems. But actually solving them requires an algorithm. This chapter introduces an essential algorithm called the **dynamic**

programming algorithm, and various instantiations including the **Viterbi** algorithm, the **minimum edit distance** algorithm, and the **forward** algorithm. We will also see the use of a probabilistic version of the finite-state automaton called the **weighted automaton**.

5.1 DEALING WITH SPELLING ERRORS

The detection and correction of spelling errors is an integral part of modern word-processors. The very same algorithms are also important in applications in which even the individual letters aren't guaranteed to be accurately identified: **optical character recognition** (**OCR**) and **on-line handwriting recognition**. **Optical character recognition** is the term used for automatic recognition of machine or hand-printed characters. An optical scanner converts a machine or hand-printed page into a bitmap which is then passed to an OCR algorithm.

OCR

On-line handwriting recognition is the recognition of human printed or cursive handwriting as the user is writing. Unlike OCR analysis of handwriting, algorithms for on-line handwriting recognition can take advantage of dynamic information about the input such as the number and order of the strokes, and the speed and direction of each stroke. On-line handwriting recognition is important where keyboards are inappropriate, such as in small computing environments (palm-pilot applications, etc.) or in scripts like Chinese that have large numbers of written symbols, making keyboards cumbersome.

In this chapter we will focus on detection and correction of spelling errors, mainly in typed text, but the algorithms will apply also to OCR and handwriting applications. OCR systems have even higher error rates than human typists, although they tend to make different errors than typists. For example OCR systems often misread "D" as "O" or "ri" as "n", producing 'mis-spelled' words like *dension* for *derision*, or *POQ Bach* for *PDQ Bach*. The reader with further interest in handwriting recognition should consult sources such as Tappert et al. (1990), Hu et al. (1996), and Casey and Lecolinet (1996).

Kukich (1992), in her survey article on spelling correction, breaks the field down into three increasingly broader problems:

1. **non-word error detection:** detecting spelling errors that result in non-words (like *graffe* for *giraffe*)

2. **isolated-word error correction:** correcting spelling errors that result in non-words, for example correcting *graffe* to *giraffe*, but looking only at the word in isolation.

3. **context-dependent error detection and correction:** using the context to help detect and correct spelling errors even if they accidentally result in an actual word of English (**real-word errors**). This can happen from typographical errors (insertion, deletion, transposition) which accidently produce a real word (e.g., *there* for *three*), or because the writer substituted the wrong spelling of a homophone or near-homophone (e.g., *dessert* for *desert*, or *piece* for *peace*).

REAL-WORD ERRORS

The next section will discuss the kinds of spelling-error patterns that occur in typed text and OCR and handwriting-recognition input.

5.2 Spelling Error Patterns

The number and nature of spelling errors in human typed text differs from those caused by pattern-recognition devices like OCR and handwriting recognizers. Grudin (1983) found spelling error rates of between 1 and 3% in human typewritten text (this includes both non-word errors and real-word errors). This error rate goes down significantly for copy-edited text. The rate of spelling errors in handwritten text itself is similar; word error rates of between 1.5 and 2.5% have been reported (Kukich, 1992).

The errors of OCR and on-line hand-writing systems vary. Yaeger et al. (1998) propose, based on studies that they warn are inconclusive, that the on-line printed character recognition on Apple Computer's NEWTON MESSAGEPAD had a word accuracy rate of 97–98%, that is, an error rate of 2–3%, but with a high variance (depending on the training of the writer, etc.). It is not clear whether the failure of the NEWTON was because this error rate was optimistic or because a 2–3% error rate is unacceptable. More recent devices, like 3Com's Palm Pilot, often use a special input script (like the Palm Pilot's "Graffiti") instead of allowing arbitrary handwriting. OCR error rates also vary widely depending on the quality of the input; (Lopresti and Zhou, 1997) suggest that OCR letter-error rates typically range from 0.2% for clean, first-generation copy to 20% or worse for multigeneration photocopies and faxes.

In an early study, Damerau (1964) found that 80% of all misspelled words (non-word errors) in a sample of human keypunched text were caused by **single-error misspellings**: a single one of the following errors:[1]

- **insertion**: mistyping *the* as *ther*
- **deletion**: mistyping *the* as *th*
- **substitution**: mistyping *the* as *thw*
- **transposition**: mistyping *the* as *hte*

INSERTION
DELETION
SUBSTITUTION
TRANSPOSITION

Because of this study, much following research has focused on the correction of single-error misspellings. Indeed, the first algorithm we will present later in this chapter relies on the large proportion of single-error misspellings.

Kukich (1992) breaks down human typing errors into two classes. **Typographic errors** (for example misspelling *spell* as *speel*), are generally related to the keyboard. **Cognitive errors** (for example misspelling *separate* as *seperate*) are caused by writers who don't know how to spell the word. Grudin (1983) found that the keyboard was the strongest influence on the errors produced; typographic errors constituted the majority of all error types. For example consider substitution errors, which were the most common error type for novice typists, and the second most common error type for expert typists. Grudin found that immediately adjacent keys in the same row accounted for 59% of the novice substitutions and 31% of the error substitutions (e.g., *smsll* for *small*). Adding in errors in the same column and **homologous** errors (hitting the corresponding key on the opposite side of the keyboard with the other hand), a total of 83% of the novice substitutions and 51% of the expert substitutions could be considered keyboard-based errors. Cognitive errors included phonetic errors (substituting a phonetically equivalent sequence of letters (*seperate* for *separate*) and homonym errors (substituting *piece* for *peace*). Homonym errors will be discussed in Chapter 7 when we discuss real-word error correction.

While typing errors are usually characterized as substitutions, insertions, deletions, or transpositions, OCR errors are usually grouped into five classes: substitutions, multisubstitutions, space deletions or insertions, and

[1] In another corpus, Peterson (1986) found that single-error misspellings accounted for an even higher percentage of all misspelled words (93–95%). The difference between the 80% and the higher figure may be due to the fact that Damerau's text included errors caused in transcription to punched card forms, errors in keypunching, and errors caused by paper tape equipment (!) in addition to purely human misspellings.

failures. Lopresti and Zhou (1997) give the following example of common OCR errors:

Correct:
The quick brown fox jumps over the lazy dog.
Recognized:
'lhe q˜ ick brown foxjurnps ovcr tb l azy dog.

Substitutions ($e \rightarrow c$) are generally caused by visual similarity (rather than keyboard distance), as are multisubstitutions ($T \rightarrow$ *'l*, $m \rightarrow$ *rn*, *he* $\rightarrow$ *b*). Multisubstitutions are also often called **framing errors**. Failures (represented by the tilde character '˜'; $u \rightarrow$ ˜) are cases where the OCR algorithm does not select any letter with sufficient accuracy.

5.3 Detecting Non-Word Errors

Detecting non-word errors in text, whether typed by humans or scanned, is most commonly done by the use of a dictionary. For example, the word *foxjurnps* in the OCR example above would not occur in a dictionary. Some early research (Peterson, 1986) had suggested that such spelling dictionaries would need to be kept small, because large dictionaries contain very rare words that resemble misspellings of other words. For example *wont* is a legitimate but rare word but is a common misspelling of *won't*. Similarly, *veery* (a kind of thrush) might also be a misspelling of *very*. Based on a simple model of single-error misspellings, Peterson showed that it was possible that 10% of such misspellings might be "hidden" by real words in a 50,000 word dictionary, but that 15% of single-error misspellings might be "hidden" in a 350,000-word dictionary. In practice, Damerau and Mays (1989) found that this was not the case; while some misspellings were hidden by real words in a larger dictionary, in practice the larger dictionary proved more help than harm.

Because of the need to represent productive inflection (the *-s* and *ed* suffixes) and derivation, dictionaries for spelling error detection usually include models of morphology, just as the dictionaries for text-to-speech we saw in Chapters 3 and 4. Early spelling error detectors simply allowed any word to have any suffix – thus Unix SPELL accepts bizarre prefixed words like *misclam* and *antiundoggingly* and suffixed words based on *the* like *thehood* and *theness*. Modern spelling error detectors use more linguistically-motivated morphological representations (see Chapter 3).

5.4 Probabilistic Models

This section introduces probabilistic models of pronunciation and spelling variation. These models, particularly the **Bayesian inference** or **noisy channel** model, will be applied throughout this book to many different problems.

We claimed earlier that the problem of ASR pronunciation modeling, and the problem of spelling correction for typing or for OCR, can be modeled as problems of mapping from one string of symbols to another. For speech recognition, given a string of symbols representing the pronunciation of a word in context, we need to figure out the string of symbols representing the lexical or dictionary pronunciation, so we can look the word up in the dictionary. Similarly, given the incorrect sequence of letters in a mis-spelled word, we need to figure out the correct sequence of letters in the correctly spelled word.

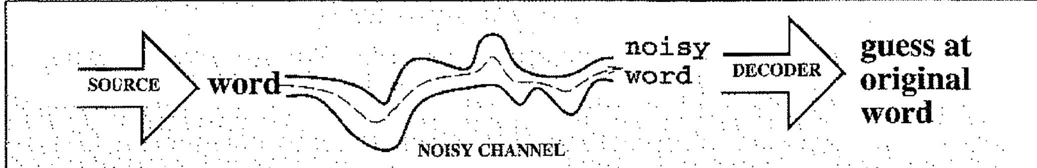

Figure 5.1 The noisy channel model.

NOISY CHANNEL

The intuition of the **noisy channel** model (see Figure 5.1) is to treat the surface form (the "reduced"" pronunciation or misspelled word) as an instance of the lexical form (the "lexical" pronunciation or correctly-spelled word) which has been passed through a noisy communication channel. This channel introduces "noise" which makes it hard to recognize the "true" word. Our goal is then to build a model of the channel so that we can figure out how it modified this "true" word and hence recover it. For the complete speech recognition tasks, there are many sources of "noise"; variation in pronunciation, variation in the realization of phones, acoustic variation due to the channel (microphones, telephone networks, etc.). Since this chapter focuses on pronunciation, what we mean by "noise" here is the variation in pronunciation that masks the lexical or "canonical" pronunciation; the other sources of noise in a speech recognition system will be discussed in Chapter 7. For spelling error detection, what we mean by noise is the spelling errors which mask the correct spelling of the word. The metaphor of the noisy channel comes from the application of the model to speech recognition in the IBM labs in the 1970s (Jelinek, 1976). But the algorithm itself is a special case

BAYESIAN

of **Bayesian inference** and as such has been known since the work of Bayes (1763). Bayesian inference or Bayesian classification was applied successfully to language problems as early as the late 1950s, including the OCR work of Bledsoe in 1959, and the seminal work of Mosteller and Wallace (1964) on applying Bayesian inference to determine the authorship of the Federalist papers.

In Bayesian classification, as in any classification task, we are given some observation and our job is to determine which of a set of classes it belongs to. For speech recognition, imagine for the moment that the observation is the string of phones which make up a word as we hear it. For spelling error detection, the observation might be the string of letters that constitute a possibly-misspelled word. In both cases, we want to classify the observations into words; thus in the speech case, no matter which of the many possible ways the word *about* is pronounced (see Chapter 4) we want to classify it as *about*. In the spelling case, no matter how the word *separate* is misspelled, we'd like to recognize it as *separate*.

Let's begin with the pronunciation example. We are given a string of phones (say [ni]). We want to know which word corresponds to this string of phones. The Bayesian interpretation of this task starts by considering all possible classes—in this case, all possible words. Out of this universe of words, we want to chose the word which is most probable given the observation we have ([ni]). In other words, we want, out of all words in the vocabulary V the single word such that $P(\text{word}|\text{observation})$ is highest. We use $\hat{\mathbf{w}}$ to mean "our estimate of the correct w", and we'll use **O** to mean "the observation sequence [ni]" (we call it a sequence because we think of each letter as an individual observation). Then the equation for picking the best word given is:

V

$\hat{\mathrm{w}}$

O

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w|O) \tag{5.1}$$

The function $\operatorname{argmax}_x f(x)$ means "the x such that $f(x)$ is maximized". While (5.1) is guaranteed to give us the optimal word w, it is not clear how to make the equation operational; that is, for a given word w and observation sequence O we don't know how to directly compute $P(w|O)$. The intuition of Bayesian classification is to use Bayes' rule to transform (5.1) into a product of two probabilities, each of which turns out to be easier to compute than $P(w|O)$. Bayes' rule is presented in (5.2); it gives us a way to break down $P(x|O)$ into three other probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{5.2}$$

We can see this by substituting (5.2) into (5.1) to get (5.3):

$$\hat{w} = \operatorname*{argmax}_{w \in V} \frac{P(O|w)P(w)}{P(O)} \tag{5.3}$$

The probabilities on the right-hand side of (5.3) are for the most part easier to compute than the probability $P(w|O)$ that we were originally trying to maximize in (5.1). For example, $P(w)$, the probability of the word itself, we can estimate by the frequency of the word. And we will see below that $P(O|w)$ turns out to be easy to estimate as well. But $P(O)$, the probability of the observation sequence, turns out to be harder to estimate. Luckily, we can ignore $P(O)$. Why? Since we are maximizing over all words, we will be computing $\frac{P(O|w)P(w)}{P(O)}$ for each word. But $P(O)$ doesn't change for each word; we are always asking about the most likely word string for the same observation O, which must have the same probability $P(O)$. Thus:

$$\hat{w} = \operatorname*{argmax}_{w \in V} \frac{P(O|w)P(w)}{P(O)} = \operatorname*{argmax}_{w \in V} P(O|w)\,P(w) \tag{5.4}$$

To summarize, the most probable word w given some observation O can be computing by taking the product of two probabilities for each word, and choosing the word for which this product is greatest. These two terms have names; $P(w)$ is called the **Prior probability**, and $P(O|w)$ is called the **likelihood**.

PRIOR

LIKELIHOOD

Key Concept #3. $$\hat{w} = \operatorname*{argmax}_{w \in V} \overbrace{P(O|w)}^{\text{likelihood}} \; \overbrace{P(w)}^{\text{prior}} \tag{5.5}$$

In the next sections we will show how to compute these two probabilities for the probabilities of pronunciation and spelling.

5.5 Applying the Bayesian Method to Spelling

There are many algorithms for spelling correction; we will focus on the Bayesian (or noisy channel) algorithm because of its generality. Chapter 6 will show how this algorithm can be extended to model real-word spelling errors; this section will focus on non-word spelling errors. The noisy channel approach to spelling correction was first suggested by Kernighan et al. (1990); their program, `correct`, takes words rejected by the Unix `spell` program, generates a list of potential correct words, rank them according to Equation (5.5), and picks the highest-ranked one.

Let's walk through the algorithm as it applies to Kernighan et al.'s (1990) example misspelling *acress*. The algorithm has two stages: *proposing candidate corrections* and *scoring the candidates*.

In order to propose candidate corrections Kernighan et al. make the simplifying assumption that the correct word will differ from the misspelling by a single insertion, deletion, substitution, or transposition. As Damerau's (1964) results show, even though this assumption causes the algorithm to miss some corrections, it should handle most spelling errors in human typed text. The list of candidate words is generated from the typo by applying any single transformation which results in a word in a large on-line dictionary. Applying all possible transformations to *acress* yields the list of candidate words in Figure 5.2.

		Transformation			
		Correct	Error	Position	
Error	Correction	Letter	Letter	(Letter #)	Type
acress	actress	t	–	2	deletion
acress	cress	–	a	0	insertion
acress	caress	ca	ac	0	transposition
acress	access	c	r	2	substitution
acress	across	o	e	3	substitution
acress	acres	–	2	5	insertion
acress	acres	–	2	4	insertion

Figure 5.2 Candidate corrections for the misspelling *acress*, together with the transformations that would have produced the error (after Kernighan et al. (1990)). "–" represents a null letter.

The second stage of the algorithm scores each correction by Equation 5.4. Let t represent the typo (the misspelled word), and let c range over the set C of candidate corrections. The most likely correction is then:

$$\hat{c} = \operatorname*{argmax}_{c \in C} \overbrace{P(t|c)}^{\text{likelihood}} \; \overbrace{P(c)}^{\text{prior}} \tag{5.6}$$

As in Equation (5.4) we have omitted the denominator in Equation (5.6) since the typo t, and hence its probability $P(t)$, is constant for all c. The prior probability of each correction $P(c)$ can be estimated by counting how often NORMALIZING the word c occurs in some corpus, and then **normalizing** these counts by the total count of all words.[2] So the probability of a particular correction word c is computed by dividing the count of c by the number N of words in the corpus. Zero counts can cause problems, and so we will add .5 to all the counts. This is called "smoothing", and will be discussed in Chapter 6; note that in Equation (5.7) we can't just divide by the total number of words N since we added .5 to the counts of all the words, so we add .5 for each of the V words in the vocabulary).

$$P(c) = \frac{C(c)+0.5}{N+0.5V} \tag{5.7}$$

Chapter 6 will talk more about the role of corpora in computing prior probabilities; for now let's use the corpus of Kernighan et al. (1990), which is the 1988 AP newswire corpus of 44 million words. Thus N is 44 million. Since in this corpus the word *actress* occurs 1343 times, the word *acres* 2879 times, and so on, the resulting prior probabilities are as follows:

c	freq(c)	p(c)
actress	1343	.0000315
cress	0	.000000014
caress	4	.0000001
access	2280	.000058
across	8436	.00019
acres	2879	.000065

Computing the likelihood term $p(t|c)$ exactly is an unsolved (unsolveable?) research problem; the exact probability that a word will be mistyped depends on who the typist was, how familiar they were with the keyboard they were using, whether one hand happened to be more tired than the other, etc. Luckily, while $p(t|c)$ cannot be computed exactly, it can be *estimated* pretty well, because the most important factors predicting an insertion, deletion, transposition are simple local factors like the identity of the correct letter itself, how the letter was misspelled, and the surrounding context. For example, the letters *m* and *n* are often substituted for each other; this is partly a fact about their identity (these two letters are pronounced similarly and they are next to each other on the keyboard), and partly a fact about context (because they are pronounced similarly, they occur in similar contexts).

One simple way to estimate these probabilities is the one that Kernighan et al. (1990) used. They ignored most of the possible influences on the probability of an error and just estimated e.g. $p(acress|across)$ using

[2] Normalizing means dividing by some total count so that the resulting probabilities fall legally between 0 and 1.

CONFUSION MATRIX

the number of times that e was substituted for o in some large corpus of errors. This is represented by a **confusion matrix**, a square 26×26 table which represents the number of times one letter was incorrectly used instead of another. For example, the cell labeled $[o,e]$ in a substitution confusion matrix would give the count of times that e was substituted for o. The cell labeled $[t,s]$ in an insertion confusion matrix would give the count of times that t was inserted after s. A confusion matrix can be computed by hand-coding a collection of spelling errors with the correct spelling and then counting the number of times different errors occurred (this has been done by Grudin (1983)). Kernighan et al. (1990) used four confusion matrices, one for each type of single-error:

- $\text{del}[x,y]$ contains the number of times in the training set that the characters xy in the correct word were typed as x.
- $\text{ins}[x,y]$ contains the number of times in the training set that the character x in the correct word was typed as xy.
- $\text{sub}[x,y]$ the number of times that x was typed as y.
- $\text{trans}[x,y]$ the number of times that xy was typed as yx.

Note that they chose to condition their insertion and deletion probabilities on the previous character; they could also have chosen to condition on the following character. Using these matrices, they estimated $p(t|c)$ as follows (where c_p is the pth character of the word c):

$$P(t|c) = \begin{cases} \dfrac{\text{del}[c_{p-1},c_p]}{\text{count}[c_{p-1}c_p]}, \text{ if deletion} \\ \dfrac{\text{ins}[c_{p-1},t_p]}{\text{count}[c_{p-1}]}, \text{ if insertion} \\ \dfrac{\text{sub}[t_p,c_p]}{\text{count}[c_p]}, \text{ if substitution} \\ \dfrac{\text{trans}[c_p,c_{p+1}]}{\text{count}[c_pc_{p+1}]}, \text{ if transposition} \end{cases} \tag{5.8}$$

Figure 5.3 shows the final probabilities for each of the potential corrections; the prior (from Equation (5.7)) is multiplied by the likelihood (computed using Equation (5.8) and the confusion matrices). The final column shows the "normalized percentage".

This implementation of the Bayesian algorithm predicts *acres* as the correct word (at a total normalized percentage of 45%), and *actress* as the second most likely word. Unfortunately, the algorithm was wrong here: The writer's intention becomes clear from the context: *. . . was called a "stellar and versatile* **acress** *whose combination of sass and glamour has defined her. . .* ". The surrounding words make it clear that *actress* and not *acres* was

c	freq(c)	p(c)	p(t|c)	p(t|c)p(c)	%
actress	1343	.0000315	.000117	3.69×10^{-9}	**37%**
cress	0	.000000014	.00000144	2.02×10^{-14}	**0%**
caress	4	.0000001	.00000164	1.64×10^{-13}	**0%**
access	2280	.000058	.000000209	1.21×10^{-11}	**0%**
across	8436	.00019	.0000093	1.77×10^{-9}	**18%**
acres	2879	.000065	.0000321	2.09×10^{-9}	**21%**
acres	2879	.000065	.0000342	2.22×10^{-9}	**23%**

Figure 5.3 Computation of the ranking for each candidate correction. Note that the highest ranked word is not *actress* but *acres* (the two lines at the bottom of the table), since *acres* can be generated in two ways. The *del*[], *ins*[], *sub*[], and *trans*[] confusion matrices are given in full in Kernighan et al. (1990).

the intended word; Chapter 6 will show how to augment the computation of the prior probability to use the surrounding words.

The algorithm as we have described it requires hand-annotated data to train the confusion matrices. An alternative approach used by Kernighan et al. (1990) is to compute the matrices by iteratively using this very spelling error correction algorithm itself. The iterative algorithm first initializes the matrices with equal values; thus any character is equally likely to be deleted, equally likely to be substituted for any other character, etc. Next the spelling error correction algorithm is run on a set of spelling errors. Given the set of typos paired with their corrections, the confusion matrices can now be recomputed, the spelling algorithm run again, and so on. This clever method turns out to be an instance of the important **EM** algorithm (Dempster et al., 1977) that we will discuss in Chapter 7 and Appendix D. Kernighan et al. (1990)'s algorithm was evaluated by taking some spelling errors that had two potential corrections, and asking three human judges to pick the best correction. Their program agreed with the majority vote of the human judges 87% of the time.

5.6 Minimum Edit Distance

The previous section showed that the Bayesian algorithm, as implemented with confusion matrices, was able to rank candidate corrections. But Kernighan et al. (1990) relied on the simplifying assumption that each word had only a single spelling error. Suppose we wanted a more powerful algorithm

DISTANCE

which could handle the case of multiple errors? We could think of such an algorithm as a general solution to the problem of **string distance.** The "string distance" is some metric of how alike two strings are to each other. The Bayesian method can be viewed as a way of applying such an algorithm to the spelling error correction problem; we pick the candidate word which is "closest" to the error in the sense of having the highest probability given the error.

MINIMUM EDIT DISTANCE

One of the most popular classes of algorithms for finding string distance are those that use some version of the **minimum edit distance** algorithm, named by Wagner and Fischer (1974) but independently discovered by many people; see the History section. The minimum edit distance between two strings is the minimum number of editing operations (insertion, deletion, substitution) needed to transform one string into another. For example the gap between intention and execution is five operations, which can be represented in three ways; as a **trace**, an **alignment**, or a **operation list** as show in Figure 5.4.

ALIGNMENT

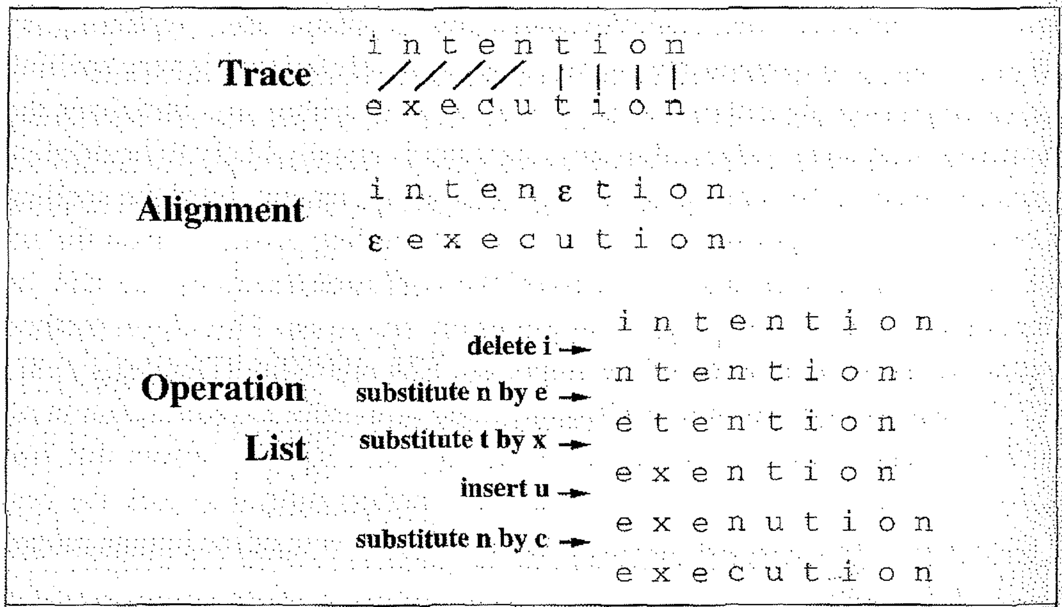

Figure 5.4 Three methods for representing differences between sequences (after Kruskal (1983))

We can also assign a particular cost or weight to each of these operations. The **Levenshtein** distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966). Thus the Levenshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternate version of his metric in which each insertion or deletion has a cost of one, and substitutions are not allowed (equivalent to allowing substitution, but giving each substitution a cost of 2, since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8. We can also weight operations by more complex functions, for example by using the confusion matrices discussed above to assign a probability to each operation. In this case instead of talking about the "minimum edit distance" between two strings, we are talking about the "maximum probability **alignment**" of one string with another. If we do this, an augmented minimum edit distance algorithm which multiplies the probabilities of each transformation can be used to estimate the Bayesian likelihood of a multiple-error typo given a candidate correction.

DYNAMIC PROGRAMMING

The minimum edit distance is computed by **dynamic programming**. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. This class of algorithms includes the most commonly-used algorithms in speech and language processing, among them the **minimum edit distance** algorithm for spelling error correction the **Viterbi** algorithm and the **forward** algorithm which are used both in speech recognition and in machine translation, and the **CYK** and **Earley** algorithm used in parsing. We will introduce the minimum-edit-distance, Viterbi, and forward algorithms in this chapter and Chapter 7, the Earley algorithm in Chapter 10, and the CYK algorithm in Chapter 12.

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. For example, consider the sequence or "path" of transformed words that comprise the minimum edit distance between the strings *intention* and *execution*. Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal operation-list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention* then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

Dynamic programming algorithms for sequence comparison work by creating a distance matrix with one column for each symbol in the target sequence and one row for each symbol in the source sequence (i.e., target along the bottom, source along the side). For minimum edit distance, this matrix is the *edit-distance* matrix. Each cell *edit-distance*[*i*,*j*] contains the distance between the first i characters of the target and the first j characters of the source. Each cell can be computed as a simple function of the surrounding cells; thus starting from the beginning of the matrix it is possible to fill in every entry. The value in each cell is computing by taking the minimum of the three possible paths through the matrix which arrive there:

$$P(t|c) = \min \begin{cases} distance[i-1, j] + ins\text{-}cost(target_i) \\ distance[i-1, j-1] + subst\text{-}cost(source_j, target_i) \\ distance[i, j-1] + del\text{-}cost(source_j)) \end{cases} \quad (5.9)$$

The algorithm itself is summarized in Figure 5.5, while Figure 5.6 shows the results of applying the algorithm to the distance between *intention* and *execution* assuming the version of Levenshtein distance in which insertions and deletions each have a cost of 1 and substitutions have a cost of 2.

```
function MIN-EDIT-DISTANCE(target, source) returns min-distance

  n ← LENGTH(target)
  m ← LENGTH(source)
  Create a distance matrix distance[n+1,m+1]
  distance[0,0] ← 0
  for each column i from 0 to n do
    for each row j from 0 to m do
      distance[i,j] ← MIN( distance[i−1,j] + ins-cost(target_i),
                           distance[i−1,j−1] + subst-cost(source_j, target_i),
                           distance[i,j−1] + del-cost(source_j))
```

Figure 5.5 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms.

5.7 English Pronunciation Variation

When any of the fugitives of Ephraim said: 'Let me go over,' the men of Gilead said unto him: 'Art thou an Ephraimite?' If he said: 'Nay'; then said they unto him: 'Say now Shibboleth'; and he said 'Sibboleth'; for he could not frame to pronounce it right; then they laid hold on him, and slew him at the fords of the Jordan.

Judges 12:5-6

n	9	10	11	10	11	12	11	10	9	**8**
o	8	9	10	9	10	11	10	9	**8**	9
i	7	8	9	8	9	10	9	**8**	9	10
t	6	7	8	7	8	9	**8**	9	10	11
n	5	6	7	6	7	**8**	9	10	11	12
e	4	5	6	**5**	**6**	7	8	9	10	11
t	3	4	**5**	6	7	8	9	10	11	12
n	2	3	**4**	5	6	7	8	8	10	11
i	1	**2**	3	4	5	6	7	8	9	10
#	**0**	1	2	3	4	5	6	7	8	9
	#	e	x	e	c	u	t	i	o	n

Figure 5.6 Computation of minimum edit distance between *intention* and *execution* via algorithm of Figure 5.5, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. Substitution of a character for itself has a cost of 0.

This passage from Judges is a rather gory reminder of the political importance of pronunciation variation. Even in our (hopefully less political) computational applications of pronunciation, it is important to correctly model how pronunciations can vary. We have already seen that a phoneme can be realized as different allophones in different phonetic environments. We have also shown how to write rules and transducers to model these changes for speech synthesis. Unfortunately, these models significantly simplified the nature of pronunciation variation. In particular, pronunciation variation is caused by many factors in addition to the phonetic environment. This section summarizes some of these kinds of variation; the following section will introduce the probabilistic tools for modeling it.

Pronunciation variation is extremely widespread. Figure 5.7 shows the most common pronunciations of the words *because* and *about* from the hand-transcribed Switchboard corpus of American English telephone conversations. Note the wide variation in pronunciation for these two words when spoken as part of a continuous stream of speech.

LEXICAL VARIATION

ALLOPHONIC VARIATION

What causes this variation? There are two broad classes of pronunciation variation: **lexical variation** and **allophonic variation**. We can think of lexical variation as a difference in what segments are used to represent the word in the lexicon, while allophonic variation is a difference in how the individual segments change their value in different contexts. In Figure 5.7, most of the variation in pronunciation is allophonic; that is, due to the influ-

because			about		
IPA	**ARPAbet**	**%**	**IPA**	**ARPAbet**	**%**
[bikʌz]	[b iy k ah z]	27%	[əbaʊ]	[ax b aw]	32%
[bɨkʌz]	[b ix k ah z]	14%	[əbaʊt]	[ax b aw t]	16%
[kʌz]	[k ah z]	7%	[baʊ]	[b aw]	9%
[kəz]	[k ax z]	5%	[ʌbaʊ]	[ix b aw]	8%
[bɨkəz]	[b ix k ax z]	4%	[ɨbaʊt]	[ix b aw t]	5%
[bɪkʌz]	[b ih k ah z]	3%	[ɨbæ]	[ix b ae]	4%
[bəkʌz]	[b ax k ah z]	3%	[əbæɾ]	[ax b ae dx]	3%
[kʊz]	[k uh z]	2%	[baʊɾ]	[b aw dx]	3%
[ks]	[k s]	2%	[bæ]	[b ae]	3%
[kɨz]	[k ix z]	2%	[baʊt]	[b aw t]	3%
[kɪz]	[k ih z]	2%	[əbaʊɾ]	[ax b aw dx]	3%
[bikʌʒ]	[b iy k ah zh]	2%	[əbæ]	[ax b ae]	3%
[bikʌs]	[b iy k ah s]	2%	[bɑ]	[b aa]	3%
[bikʌ]	[b iy k ah]	2%	[bæɾ]	[b ae dx]	3%
[bikɑz]	[b iy k aa z]	2%	[ɨbaʊɾ]	[ix b aw dx]	2%
[əz]	[ax z]	2%	[ɨbɑt]	[ix b aa t]	2%

Figure 5.7 The 16 most common pronunciations of *because* and *about* from the hand-transcribed Switchboard corpus of American English conversational telephone speech (Godfrey et al., 1992; Greenberg et al., 1996).

ence of the surrounding sounds, syllable structure, and so forth. But the fact that the word *because* can be pronounced either as monosyllabic *'cause* or bisyllabic *because* is probably a lexical fact, having to do perhaps with the level of informality of speech.

SOCIOLINGUISTIC

DIALECT VARIATION

An important source of lexical variation (although it can also affect allophonic variation) is **sociolinguistic** variation. Sociolinguistic variation is due to extralinguistic factors such as the social identity or background of the speaker. One kind of sociolinguistic variation is **dialect variation**. Speakers of some deep-southern dialects of American English use a monophthong or near-monophthong [a] or [aɛ] instead of a diphthong in some words with the vowel [aɪ]. In these dialects *rice* is pronounced [raːs]. African-American Vernacular English (AAVE) has many of the same vowel differences from General American as does Southern American English, and also has individual words with specific pronunciations such as [bɪdnɪs] for *business* and [æks] for *ask*. For older speakers or those not from the American West or Midwest, the words *caught* and *cot* have different vowels ([kɔt] and [kɑt] respectively). Young American speakers or those from the West pronounce the two words *cot* and *caught* the same; the vowels [ɔ] and [ɑ] are usually not distinguished in these dialects. For some speakers from New York City like the first author's parents, the words *Mary* ([meɪri]), *marry* ([mæri]), and *merry* ([mɛri]) are all pronounced differently, while other New York City speakers like the second author pronounce *Mary*, and *merry* identically, but differently than *marry*. Most American speakers pronounce all three of these words identically as ([mɛri]). Students who are interested in dialects of English should consult Wells (1982), the most comprehensive study of dialects of English around the world.

REGISTER
STYLE

Other sociolinguistic differences are due to **register** or **style** rather than dialect. In a pronunciation difference that is due to style, the same speaker might pronounce the same word differently depending on who they were talking to or what the social situation is; this is probably the case when choosing between *because* and *'cause* above. One of the most well-studied examples of style-variation is the suffix *-ing* (as in *something*), which can be pronounced [ɪŋ] or /ɪn/ (this is often written *somethin'*). Most speakers use both forms; as Labov (1966) shows, they use [ɪŋ] when they are being more formal, and [ɪn] when more casual. In fact whether a speaker will use [ɪŋ] or [ɪn] in a given situation varies markedly according to the social context, the gender of the speaker, the gender of the other speaker, and so on. Wald and Shopen (1981) found that men are more likely to use the non-standard form [ɪn] than women, that both men and women are more likely to use more of the standard form [ɪŋ] when the addressee is a women, and that men (but not women) tend to switch to [ɪn] when they are talking with friends.

Where lexical variation happens at the lexical level, allophonic variation happens at the surface form and reflects phonetic and articulatory factors.[3] For example, most of the variation in the word *about* in Figure 5.7 was caused by changes in one of the two vowels or by changes to the final [t]. Some of this variation is due to the allophonic rules we have already discussed for the realization of the phoneme /t/. For example the pronunciation of *about* as [əbaʊɾ]/[ax b aw dx]) has a flap at the end because the next word was the word *it*, which begins with a vowel; the sequence *about it* was pronounced [əbaʊɾɨ]/[ax b aw dx ix]). Similarly, note that final [t] is often deleted; (*about* as [baʊ]/[b aw]). Considering these cases as "deleted" is actually a simplification; many of these "deleted" cases of [t] are actually

[3] For some purposes we distinguish between allophonic variation and what are called "optional phonological rules"; for the purposes of this textbook we will lump these both together as "allophonic variation".

realized as a slight change to the vowel quality called **glottalization** which are not represented in these transcriptions.

When we discussed these rules earlier, we implied that they were deterministic; given an environment, a rule always applies. This is by no means the case. Each of these allophonic rules is dependent on a complicated set of factors that must be interpreted probabilistically. In the rest of this section we summarize more of these rules and talk about the influencing factors. Many of these rules model **coarticulation**, which is a change in a segment due to the movement of the articulators in neighboring segments. Most allophonic rules relating English phoneme to their allophones can be grouped into a small number of types: assimilation, dissimilation, deletion, flapping, vowel reduction, and epenthesis.

COARTICULATION

ASSIMILATION

Assimilation is the change in a segment to make it more like a neighboring segment. The dentalization of [t] to ([t̪]) before the dental consonant [θ] is an example of assimilation. Another common type of assimilation in English and cross-linguistically is **palatalization**. Palatalization occurs when the constriction for a segment occurs closer to the palate than it normally would, because the following segment is palatal or alveolo-palatal. In the most common cases, /s/ becomes [ʃ], /z/ becomes [ʒ], /t/ becomes [tʃ and /d/ becomes dʒ]. We saw one case of palatalization in Figure 5.7 in the pronunciation of *because* as [bikʌʒ] (ARPAbet [b iy k ah zh]). Here the final segment of *because*, a lexical /z/, is realized as [ʒ], because the following word was *you've*. So the sequence *because you've* was pronounced [bikʌʒuv]. A simple version of a palatalization rule might be expressed as follows; Figure 5.8 shows examples from the Switchboard corpus.

PALATALIZATION

$$\left\{\begin{matrix}[s]\\ [z]\\ [t]\\ [d]\end{matrix}\right\} \Rightarrow \left\{\begin{matrix}[\text{ʃ}]\\ [\text{ʒ}]\\ [\text{tʃ}]\\ [\text{dʒ}]\end{matrix}\right\} / \_\_ \{\, y \,\} \qquad (5.10)$$

Note in Figure 5.8 that whether a [t] is palatalized depends on lexical factors like word frequency ([t] is more likely to be palatalized in frequent words and phrases).

DELETION

Deletion is quite common in English speech. We saw examples of deletion of final /t/ above, in the words *about* and *it*. /t/ and /d/ are often deleted before consonants, or when they are part of a sequence of two or three consonants; Figure 5.9 shows some examples.

$$\left\{\begin{matrix} t \\ d \end{matrix}\right\} \Rightarrow \emptyset \,/\, \text{V} \_\_ \text{C} \qquad (5.11)$$

The many factors that influence the deletion of /t/ and /d/ have been extensively studied. For example /d/ is more likely to be deleted than /t/.

Phrase	IPA Lexical	IPA Reduced	ARPAbet Reduced
set your	[sɛtjɔr]	[sɛtʃɚ]	[s eh ch er]
not yet	[nɑtjɛt]	[nɑtʃɛt]	[n aa ch eh t]
last year	[læstjir]	[læstʃir]	[l ae s ch iy r]
what you	[wʌtju]	[wətʃu]	[w ax ch uw]
this year	[ðɪsjir]	[ðɪʃir]	[dh ih sh iy r]
because you've	[bikʌzjuv]	[bikʌʒuv]	[b iy k ah zh uw v]
did you	[dɪdju]	[dɪdʒyʌ]	[d ih jh y ah]

Figure 5.8 Examples of palatalization from the Switchboard corpus; the lemma *you* (including *your*, *you've*, and *you'd*) was by far the most common cause of palatalization, followed by *year(s)* (especially in the phrases *this year* and *last year*).

Phrase	IPA Lexical	IPA Reduced	ARPAbet Reduced
find him	[faɪndhɪm]	[faɪnɨm]	[f ay n ix m]
around this	[əraʊndðɪs]	[ɨraʊnɪs]	[ix r aw n ih s]
mind boggling	[maɪnbɔgl̩ɪŋ]	[maɪnbɔgl̩ɪŋ]	[m ay n b ao g el ih ng]
most places	[moʊstpleɪsɨz]	[moʊspleɪsɨz]	[m ow s p l ey s ix z]
draft the	[dræftði]	[dræfði]	[d r ae f dh iy]
left me	[lɛftmi]	[lɛfmi]	[l eh f m iy]

Figure 5.9 Examples of /t/ and /d/ deletion from Switchboard. Some of these examples may have glottalization instead of being completely deleted.

Both are more likely to be deleted before a consonant (Labov, 1972). The final /t/ and /d/ in the words *and* and *just* are particularly likely to be deleted (Labov, 1975; Neu, 1980). Wolfram (1969) found that deletion is more likely in faster or more casual speech, and that younger people and males are more likely to delete. Deletion is more likely when the two words surrounding the segment act as a sort of phrasal unit, either occurring together frequently (Bybee, 1996), having a high **mutual information** or **trigram predictability** (Gregory et al., 1999), or being tightly connected for other reasons (Zwicky, 1972). Fasold (1972), Labov (1972), and many others have shown that deletion is less likely if the word-final /t/ or /d/ is the past tense ending. For example in Switchboard, deletion is more likely in the word *around* (73% /d/-deletion) than in the word *turned* (30% /d/-deletion) even though the two words have similar frequencies.

The **flapping** rule is significantly more complicated than we suggested in Chapter 4, as a number of scholars have pointed out (see especially Rhodes (1992)). The preceding vowel is highly likely to be stressed, although this is not necessary (for example there is commonly a flap in the word *thermometer* [θɚˈmɑmɨɾɚ]). The following vowel is highly likely to be unstressed, although again this is not necessary. /t/ is much more likely to flap than /d/. There are complicated interactions with syllable, foot, and word boundaries. Flapping is more likely to happen when the speaker is speaking more quickly, and is more likely to happen at the end of a word when it forms a collocation (high mutual information) with the following word (Gregory et al., 1999). Flapping is less likely to happen when a speaker **hyperarticulates**, i.e. uses a particularly clear form of speech, which often happens when users are talking to computer speech recognition systems (Oviatt et al., 1998). There is a nasal flap [ɾ̃] whose tongue movements resemble the oral flap but in which the velum is lowered. Finally, flapping doesn't always happen, even when the environment is appropriate; thus the flapping rule, or transducer, needs to be probabilistic, as we will see below.

HYPERARTICULATES

We have saved for last one of the most important phonological processes: **vowel reduction**, in which many vowels in unstressed syllables are realized as **reduced vowels**, the most common of which is **schwa** ([ə]). Stressed syllables are those in which more air is pushed out of the lungs; stressed syllables are longer, louder, and usually higher in pitch than unstressed syllables. Vowels in unstressed syllables in English often don't have their full form; the articulatory gesture isn't as complete as for a full vowel. As a result the shape of the mouth is somewhat neutral; the tongue is neither particularly high nor particularly low. For example the second vowels in *parakeet* is schwa: [pærəkit].

REDUCED VOWELS

SCHWA

While schwa is the most common reduced vowel, it is not the only one, at least not in some dialects. Bolinger (1981) proposed three reduced vowels: a reduced mid vowel [ə], a reduced front vowel [ɨ], and a reduced rounded vowel [ɵ]. But the majority of computational pronunciation lexicons or computational models of phonology systems limit themselves to one reduced vowel ([ə]) (for example PRONLEX and CELEX) or at most two ([ə] =ARPABET [ax] and [ɨ] = ARPAbet [ix]). Miller (1998) was able to train a neural net to automatically categorize a vowel as [ə] or [ɨ] based only on the phonetic context, which suggests that for speech recognition and text-to-speech purposes, one reduced vowel is probably adequate. Indeed, Wells (1982, p. 167–168) notes that [ə] and [ɨ] are falling together in many dialects of English including General American and Irish, among others, a phenomenon he calls **weak vowel merger**.

A final note: not all unstressed vowels are reduced; any vowel, and diphthongs in particular can retain their full quality even in unstressed position. For example the vowel [eɪ] (ARPAbet [ey]) can appear in stressed position as in the word *eight*) ['eɪt] or unstressed position as in the word *always* ['ɔ.weɪz]. Whether a vowel is reduced depends on many factors. For example the word *the* can be pronounced with a full vowel ði or reduced vowel ðə. It is more likely to be pronounced with the reduced vowel ðə in fast speech, in more casual situations, and when the following word begins with a consonant. It is more likely to be pronounced with the full vowel ði when the following word begins with a vowel or when the speaker is having "planning problems"; speakers are more likely to use a full vowel than a reduced one if they don't know what they are going to say next (Fox Tree and Clark, 1997). See Keating et al. (1994) and Jurafsky et al. (1998) for more details on factors effecting vowel reduction in the TIMIT and Switchboard corpora. Other factors influencing reduction include the frequency of the word, whether this is the final vowel in a phrase, and even the idiosyncracies of individual speakers.

5.8 The Bayesian Method for Pronunciation

HEAD KNIGHT OF NI:	Ni!
KNIGHTS OF NI:	Ni! Ni! Ni! Ni! Ni!
ARTHUR:	Who are you?
HEAD KNIGHT:	We are the Knights Who Say... 'Ni'!
RANDOM:	Ni!
ARTHUR:	No! Not the Knights Who Say 'Ni'!
HEAD KNIGHT:	The same!
BEDEVERE:	Who are they?
HEAD KNIGHT:	We are the keepers of the sacred words: 'Ni', 'Peng', and 'Neee–wom'!

Graham Chapman, John Cleese, Eric Idle, Terry Gilliam, Terry Jones, and Michael Palin, *Monty Python and the Holy Grail* 1975.

The Bayesian algorithm that we used to pick the optimal correction for a spelling error can be used to solve what is often called the **pronunciation** subproblem in speech recognition. In this task, we are given a series of phones and our job is to compute the most probable word which generated them. For this chapter, we will simplify the problem in an important way by assuming the correct string of phones. A real speech recognizer relies on probabilistic estimators for each phone, so it is never sure about the identity of any phone. We will relax this assumption in Chapter 7; for now, let's look at the simpler problem.

We'll also begin with another simplification by assuming that we already know where the word boundaries are. Later in the chapter, we'll show that we can simultaneously find word boundaries ("segment") and model pronunciation variation.

Consider the particular problem of interpreting the sequence of phones [ni], when it occurs after the word *I* at the beginning of a sentence. Stop and see if you can think of any words which are likely to have been pronounced [ni] before you read on. The word "Ni" is not allowed.

You probably thought of the word *knee*. This word is in fact pronounced [ni]. But an investigation of the Switchboard corpus produces a total of 7 words which can be pronounced [ni]! The seven words are *the*, *neat*, *need*, *new*, *knee*, *to*, and *you*.

How can the word *the* be pronounced [ni]? The explanation for this pronunciation (and all the others except the one for *knee*) lies in the contextually-induced pronunciation variation we discussed in Chapter 4. For example, we saw that [t] and [d] were often deleted word finally, especially before coronals; thus the pronunciation of *neat* as [ni] happened before the word *little* (*neat little* → [niləl̩]). The pronunciation of *the* as [ni] is caused by the regressive assimilation process also discussed in Chapter 4. Recall that in nasal assimilation, phones before or after nasals take on nasal manner of articulation. Thus [θ] can be realized as [n]. The many cases of *the* pronounced as [ni] in Switchboard occurred after words like *in*, *on*, and *been* (so *in the* → [ɪnni]). The pronunciation of *new* as [ni] occurred most frequently in the word *New York*; the vowel [u] has fronted to [i] before a [y].

The pronunciation of *to* as [ni] occurred after the work *talking* (*talking to you* → [tɔkɪniyu]); here the [u] is palatalized by the following [y] and the [n] is functioning jointly as the final sound of *talking* and the initial sound of *to*. Because this phone is part of two separate words we will not try to model this particular mapping; for the rest of this section let's consider only the following five words as candidate lexical forms for [ni]: *knee*, *the*, *neat*, *need*, *new*.

We saw in the previous section that the Bayesian spelling error correction algorithm had two components: candidate generation, and candidate scoring. Speech recognizers often use an alternative architecture, trading off speech for storage. In this architecture, each pronunciation is expanded in advance with all possible variants, which are then pre-stored with their scores. Thus there is no need for candidate generation; the word [ni] is simply stored with the list of words that can generate it. Let's assume this method and see how the prior and likelihood are computed for each word.

We will be choosing the word whose product of prior and likelihood is the highest, according to Equation (5.12), where *y* represents the sequence of phones (in this case [ni] and *w* represents the candidate word [*the*, *new*, etc.]). The most likely word is then:

$$\hat{w} = \underset{w \in W}{\operatorname{argmax}} \overbrace{P(y|w)}^{\text{likelihood}} \overbrace{P(w)}^{\text{prior}} \tag{5.12}$$

We could choose to generate the likelihoods $p(y|w)$ by using a set of confusion matrices as we did for spelling error correction. But it turns out that confusion matrices don't do as well for pronunciation as for spelling. While misspelling tends to change the form of a word only slightly, the changes in pronunciation between a lexical and surface form are much greater. Confusion matrices only work well for single-errors, which, as we saw above, are common in misspelling. Furthermore, recall from Chapter 4 that pronunciation variation is strongly affected by the surrounding phones, lexical frequency, and stress and other prosodic factors. Thus probabilistic models of pronunciation variation include a lot more factors than a simple confusion matrix can include.

PROBABILISTIC RULES

One simple way to generate pronunciation likelihoods is via **probabilistic rules**. Probabilistic rules were first proposed for pronunciation by (Labov, 1969) (who called them **variable rules**). The idea is to take the rules of pronunciation variation we saw in Chapter 4 and associate them with probabilities. We can then run these probabilistic rules over the lexicon and generate different possible surface forms each with its own probability. For example, consider a simple version of a nasal assimilation rule which explains why *the* can be pronounced [ni]; a word-initial [ð] becomes [n] if the preceding word ended in [n] or sometimes [m]:

$$[.15]\ ð \Rightarrow n\ /\ [+nasal]\ \#\_\_ \tag{5.13}$$

The [.15] to the left of the rule is the probability; this can be computed from a large-enough labeled corpus such as the transcribed portion of Switchboard. Let *ncount* be the number of times lexical [ð] is realized word-initially by surface [n] when the previous word ends in a nasal (91 in the Switchboard corpus). Let *envcount* be the total number of times lexical [ð] occurs (whatever its surface realization) when the previous word ends in a nasal (617 in the Switchboard corpus). The resulting probability is:

$$\begin{aligned} P(\eth \rightarrow n \,/\, [+nasal]\ \#\_\_) &= \frac{ncount}{envcount} \\ &= \frac{91}{617} \\ &= .15 \end{aligned}$$

We can build similar probabilistic versions of the assimilation and deletion rules which account for the [ni] pronunciation of the other words. Figure 5.10 shows sample rules and the probabilities trained on the Switchboard pronunciation database.

Word	**Rule Name**	**Rule**	**P**
the	nasal assimilation	$\eth \Rightarrow$ n / $[+nasal]$ # ___	[.15]
neat	final t deletion	$t \Rightarrow \emptyset \,/\, V$ ___ #	[.52]
need	final d deletion	$d \Rightarrow \emptyset \,/\, V$ ___ #	[.11]
new	u fronting	$u \Rightarrow i$ / ___ # [y]	[.36]

Figure 5.10 Simple rules of pronunciation variation due to context in continuous speech accounting for the pronunciation of each of these words as [ni].

We now need to compute the prior probability $P(w)$ for each word. For spelling correction we did this by using the relative frequency of the word in a large corpus; a word which occurred 44,000 times in 44 million words receives the probability estimate $\frac{44,000}{44,000,000}$ or .001. For the pronunciation problem, let's take our prior probabilities from a collection of a written and a spoken corpus. The Brown Corpus is a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.) which was assembled at Brown University in 1963–1964 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982). The Switchboard Treebank corpus is a 1.4 million word collection of telephone conversations. Together they let us sample from both the written and spoken genres. The table below shows the probabilities for our five words; each probability is computed from the raw frequencies by normalizing by the number of words in the combined corpus (plus .5 * the number of word types; so the total denominator is 2,486,075 + 30,836):

w	freq(w)	p(w)
knee	61	.000024
the	114,834	.046
neat	338	.00013
need	1417	.00056
new	2625	.001

Now we are almost ready to answer our original question: what is the most likely word given the pronunciation [ni] and given that the previous word was *I* at the beginning of a sentence. Let's start by multiplying together our estimates for $p(w)$ and $p(y|w)$ to get an estimate; we show them sorted from most probable to least probable (*the* has a probability of 0 since the previous phone was not [n], and hence there is no other rule allowing [ð] to be realized as [n]):

Word	**p(y|w)**	**p(w)**	**p(y|w)p(w)**
new	.36	.001	.00036
neat	.52	.00013	.000068
need	.11	.00056	.000062
knee	1.00	.000024	.000024
the	0	.046	0

Our algorithm suggests that *new* is the most likely underlying word. But this is the wrong answer; the string [ni] following the word *I* came in fact from the word *need* in the Switchboard corpus. One way that people are able to correctly solve this task is word-level knowledge; people know that the word string *I need* ... is much more likely than the word string *I new* .... We don't need to abandon our Bayesian model to handle this fact; we just need to modify it so that our model also knows that *I need* is more likely than *I new*. In Chapter 6 we will see that we can do this by using a slightly more intelligent estimate of $p(w)$ called a **bigram** estimate; essentially we consider the probability of *need* following *I* instead of just the individual probability of *need*.

This Bayesian algorithm is in fact part of all modern speech recognizers. Where the algorithms differ strongly is how they detect individual phones in the acoustic signal, and on which search algorithm they use to efficiently compute the Bayesian probabilities to find the proper string of words in connected speech (as we will see in Chapter 7).

Decision Tree Models of Pronunciation Variation

In the previous section we saw how hand-written rules could be augmented with probabilities to model pronunciation variation. Riley (1991) and Withgott and Chen (1993) suggested an alternative to writing rules by hand, which has proved quite useful: automatically inducing lexical-to-surface pronunciations mappings from a labeled corpus with a **decision tree**, particularly with the kind of decision tree called a **Classification and Regression Tree (CART)** (Breiman et al., 1984). A decision tree takes a situation described by a set of features and classifies it into a category and an associated probability. For pronunciation, a decision tree can be trained to take a lexical phone and various contextual features (surrounding phones, stress and syllable structure information, perhaps lexical identity) and select an appropriate surface phone to realize it. We can think of the confusion matrices we used in spelling error correction above as degenerate decision trees; thus the substitution matrix takes a lexical phone and outputs a probability distribution over potential surface phones to be substituted. The advantage of decision trees is that they can be automatically induced from a labeled corpus, and that they are concise: Decision trees pick out only the relevant features and thus suffer less from sparseness than a matrix, which has to condition on every neighboring phone.

DECISION TREE

CART

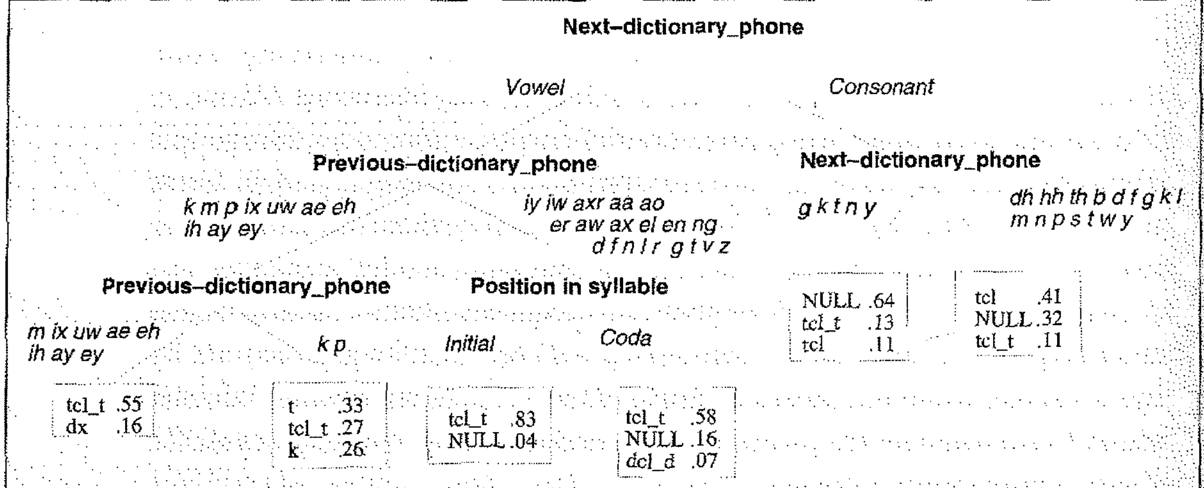
Next–dictionary_phone
Vowel
Consonant
Previous–dictionary_phone
k m p ix uw ae eh ih ay ey
iy iw axr aa ao er aw ax el en ng d f n l r g t v z
Next–dictionary_phone
g k t n y
dh hh th b d f g k l m n p s t w y
Previous–dictionary_phone
Position in syllable
NULL .64 tcl_t .13 tcl .11
tcl .41 NULL .32 tcl_t .11
m ix uw ae eh ih ay ey
k p
Initial
Coda
tcl_t .55 dx .16
t .33 tcl_t .27 k .26
tcl_t .83 NULL .04
tcl_t .58 NULL .16 dcl_d .07

Figure 5.11 Hand-pruned decision tree for the phoneme /t/ induced from the Switchboard corpus (courtesy of Eric Fosler-Lussier). This particular decision tree doesn't model flapping since flaps were already listed in the dictionary. The tree automatically induced the categories *Vowel* and *Consonant*. We have only shown the most likely realizations at each leaf node.

For example, Figure 5.11 shows a decision tree for the pronunciation of the phoneme /t/ induced from the Switchboard corpus. While this tree doesn't including flapping (there is a separate tree for flapping) it does model the fact that /t/ is more likely to be deleted before a consonant than before a vowel. Note, in fact, that the tree automatically induced the classes *Vowel* and *Consonant*. Furthermore note that if /t/ is not deleted before a consonant, it is likely to be unreleased. Finally, notice that /t/ is very unlikely to be deleted in syllable onset position.

Readers with interest in decision tree modeling of pronunciation should consult Riley (1991), Withgott and Chen (1993), and a textbook with an introduction to decision trees such as Russell and Norvig (1995).

5.9 WEIGHTED AUTOMATA

We said earlier that for purposes of efficiency a lexicon is often stored with the most likely kinds of pronunciation variation pre-compiled. The two most common representation for such a lexicon are the **trie** and the **weighted finite-state automaton/transducer** (or **probabilistic FSA/FST**) (Pereira et al., 1994). We will leave the discussion of the trie to Chapter 7, and concentrate here on the weighted automaton.

WEIGHTED

The weighted automaton is a simple augmentation of the finite automaton in which each arc is associated with a probability, indicating how likely that path is to be taken. The probability on all the arcs leaving a node must sum to 1. Figure 5.12 shows two weighted automata for the word *tomato*, adapted from Russell and Norvig (1995). The top automaton shows two possible pronunciations, representing the dialect difference in the second vowel. The bottom one shows more pronunciations (how many?) representing optional reduction or deletion of the first vowel and optional flapping of the final [t].

MARKOV CHAIN

A **Markov chain** is a special case of a weighted automaton in which the input sequence uniquely determines which states the automaton will go through. Because they can't represent inherently ambiguous problems, a Markov chain is only useful for assigning probabilities to unambiguous sequences; thus the N-gram models to be discussed in Chapter 6 are Markov chains since each word is treated as if it was unambiguous. In fact the weighted automata used in speech and language processing can be shown to be equivalent to **Hidden Markov Models (HMMs)**. Why do we introduce weighted automata in this chapter and HMMs in Chapter 7? The

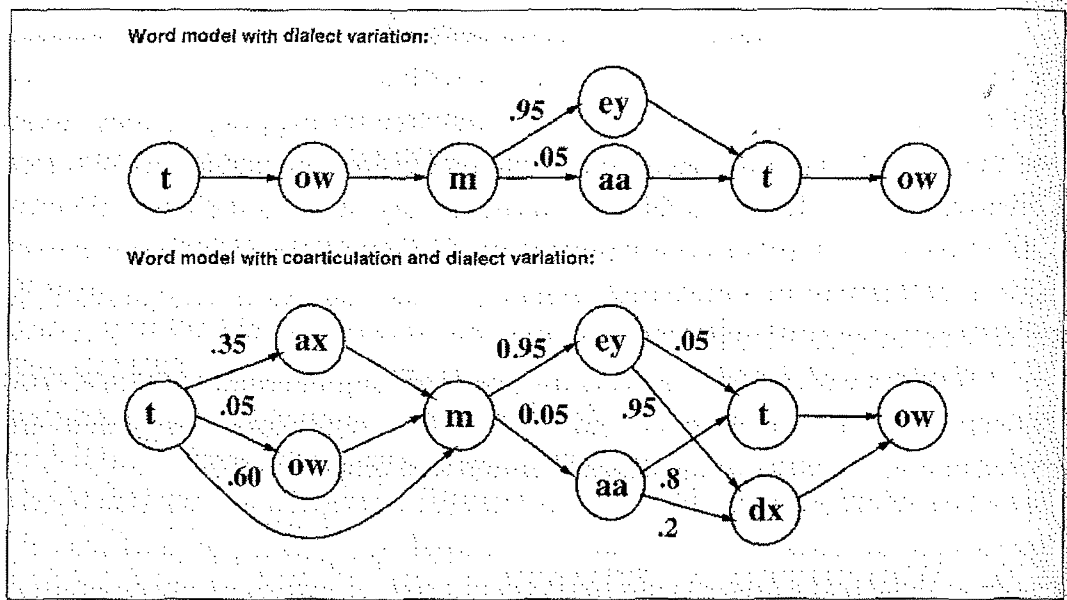
Word model with dialect variation:
t
ow
m
.95
.05
ey
aa
t
ow
Word model with coarticulation and dialect variation:
t
.35
.05
.60
ax
ow
m
0.95
0.05
ey
aa
.05
.95
.8
.2
t
dx
ow

Figure 5.12 You say [t ow m ey t ow] and I say [t ow m aa t ow]. Two pronunciation networks for the word *tomato*, adapted from Russell and Norvig (1995). The top one models sociolinguistic variation (some British or eastern American dialects); the bottom one adds in coarticulatory effects. Note the correlation between allophonic and sociolinguistic variation; the dialect with the vowel [ey] is more likely to flap than the other dialect.

two models offer a different metaphor; it is sometimes easier to think about certain problems as weighted-automata than as HMMs. The weighted automaton metaphor is often applied when the input alphabet maps relatively neatly to the underlying alphabet. For example, in the problem of correcting spelling errors in typewritten input, the input sequence consists of letters and the states of the automaton can correspond to letters. Thus it is natural to think of the problem as transducing from a set of symbols to the same set of symbols with some modifications, and hence weighted automata are naturally used for spelling error correction. In the problem of correcting errors in hand-written input, the input sequence is visual, and the input alphabet is an alphabet of lines and angles and curves. Here instead of transducing from an alphabet to itself, we need to do classification on some input sequence before considering it as a sequence of states. Hidden Markov Models provide a more appropriate metaphor, since they naturally handle separate alphabets for input sequences and state sequences. But since any probabilistic automaton in which the input sequence does not uniquely specify the state sequence can be modeled as an HMM, the difference is one of metaphor rather than explanatory power.

Weighted automata can be created in many ways. One way, first proposed by Cohen (1989) is to start with on-line pronunciation dictionaries and use hand-written rules of the kind we saw above to create different potential surface forms. The probabilities can then be assigned either by counting the number of times each pronunciation occurs in a corpus, or if the corpus is too sparse, by learning probabilities for each rule and multiplying out the rule probabilities for each surface form (Tajchman et al., 1995). Finally these weighted rules, or alternatively the decision trees we discussed in the last section, can be automatically compiled into a weighted finite-state transducer (Sproat and Riley, 1996). Alternatively, for very common words, we can simply find enough examples of the pronunciation in a transcribed corpus to build the model by just combining all the pronunciations into a network (Wooters and Stolcke, 1994).

The networks for *tomato* above were shown merely as illustration and are not from any real system; Figure 5.13 shows an automaton for the word *about* which is trained on actual pronunciations from the Switchboard corpus (we discussed these pronunciations in Chapter 4).

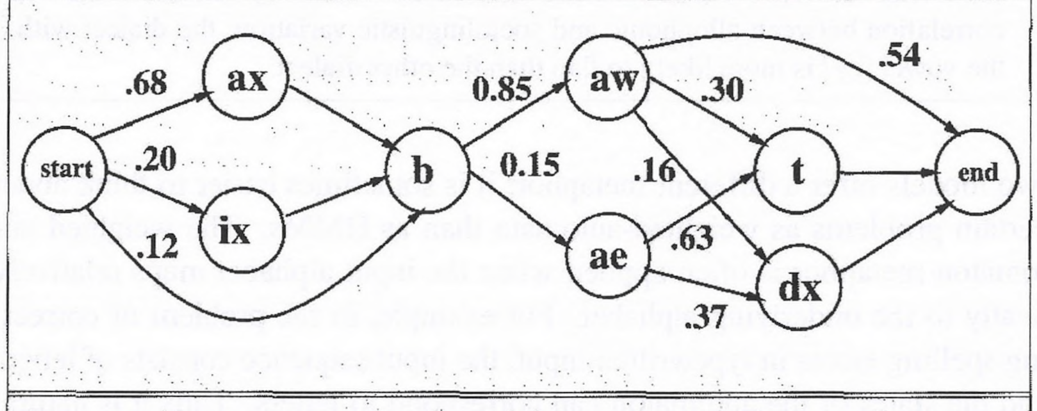

Figure 5.13 A pronunciation network for the word *about*, from the actual pronunciations in the Switchboard corpus.

Computing Likelihoods from Weighted Automata: The Forward Algorithm

One advantage of an automaton-based lexicon is that there are efficient algorithms for generating the probabilities that are needed to implement the Bayesian method of correct-word-identification of Section 5.8. These algorithms apply to weighted automata and also to the **Hidden Markov Models** that we will discuss in Chapter 7. Recall that in our example the Bayesian method is given as input a series of phones [n iy], and must choose between the words *the*, *neat*, *need*, *new*, and *knee*. This was done by computing two probabilities: the prior probability of each word, and the likelihood of the phone string [n iy] given each word. When we discussed this example earlier, we said that for example the likelihood of [n iy] given the word *need* was .11, since we computed a probability of .11 for the *final-d-deletion* rule from our Switchboard corpus. This probability is transparent for *need* since there were only two possible pronunciations ([n iy] and [n iy d]). But for words like *about*, visualizing the different probabilities is more complex. Using a precompiled weighted automata can make it simpler to see all the different probabilities of different paths through the automaton.

There is a very simple algorithm for computing the likelihood of a string of phones given the weighted automaton for a word. This algorithm, the **forward** algorithm, is an essential part of ASR systems, although in this chapter we will only be working with a simple usage of the algorithm. This is because the forward algorithm is particularly useful when there are multiple paths through an automaton which can account for the input; this is not the case in the weighted automata in this chapter, but will be true for the HMMS of Chapter 7. The forward algorithm is also an important step in defining the **Viterbi** algorithm that we will see later in this chapter.

FORWARD

Let's begin by giving a formal definition of a weighted automaton and of the input and output to the likelihood computation problem. A weighted automaton consists of

1. a sequence of states $q = (q_0 q_1 q_2 \dots q_n)$, each corresponding to a phone, and
2. a set of transition probabilities between states, a_{01}, a_{12}, a_{13}, encoding the probability of one phone following another.

We represent the states as nodes, and the transition probabilities as edges between nodes; an edge exists between two nodes if there is a non-zero transition probability between the two nodes.[4] The sequences of symbols

[4] We have used two "special" states (often called **non-emitting states**) as the start and end state; it is also possible to avoid the use of these states. In that case, an automaton must specify two more things:

1. π, an initial probability distribution over states, such that π_i is the probability that the automaton will start in state i. Of course, some states j may have $\pi_j = 0$, meaning that they cannot be initial states.
2. a set of legal accepting states.

that are input to the model (if we are thinking of it as recognizer) or which are produced by the model (if we are thinking of it as a generator) are generally called the **observation sequence**, referred to as $O = (o_1 o_2 o_3 \ldots o_t)$. (Upper-case letters are used for a sequence and lower-case letters for an individual element of a sequence). We will use this terminology when talking about weighted automata and later when talking about HMMs.

OBSERVATION SEQUENCE

Figure 5.14 shows an automaton for the word *need* with a sample observation sequence.

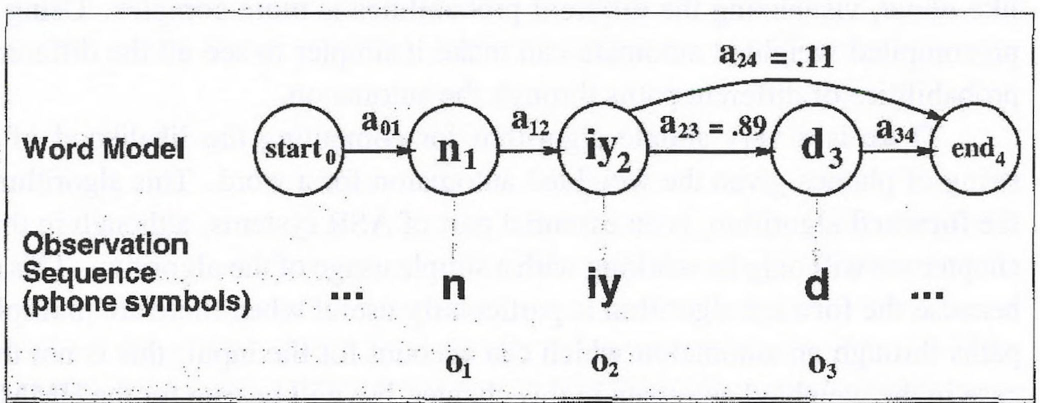

Figure 5.14 A simple weighted automaton or Markov chain pronunciation network for the word *need,* showing the transition probabilities, and a sample observation sequence. The transition probabilities a_{xy} between two states x and y are 1.0 unless otherwise specified.

This task of determining which underlying word might have produced an observation sequence is called the **decoding** problem. Recall that in order to find which of the candidate words was most probable given the observation sequence [n iy], we need to compute the product $P(O|w)P(w)$ for each candidate word (*the, need, neat, knee, new*), i.e. the likelihood of the observation sequence O given the word w times the prior probability of the word.

DECODING

The forward algorithm can be run to perform this computation for each word; we give it an observation sequence and the pronunciation automaton for a word and it will return $P(O|w)P(w)$. Thus one way to solve the decoding problem is to run the forward algorithm separately on each word and choose the word with the highest value. As we saw earlier, the Bayesian method produces the wrong result for pronunciation [n iy] as part of the word sequence *I need* (its first choice is the word *new*, and the second choice is *neat*; *need* is only the third choice). Since the forward algorithm is just a way of implementing the Bayesian approach, it will return the exact same rankings. (We will see in Chapter 6 how to augment the algorithm with **bigram** probabilities which will enable it to make use of the knowledge that the previous word was *I*).

The forward algorithm takes as input a pronunciation network for each candidate word. Because the word *the* only has the pronunciation [n iy] after nasals, and since we are assuming the actual context of this word was after the word *I* (no nasal), we will skip that word and look only at *new*, *neat*, *need*, and *knee*. Note in Figure 5.15 that we have augmented each network with the probability of each word, computed from the frequency that we saw on page 167.

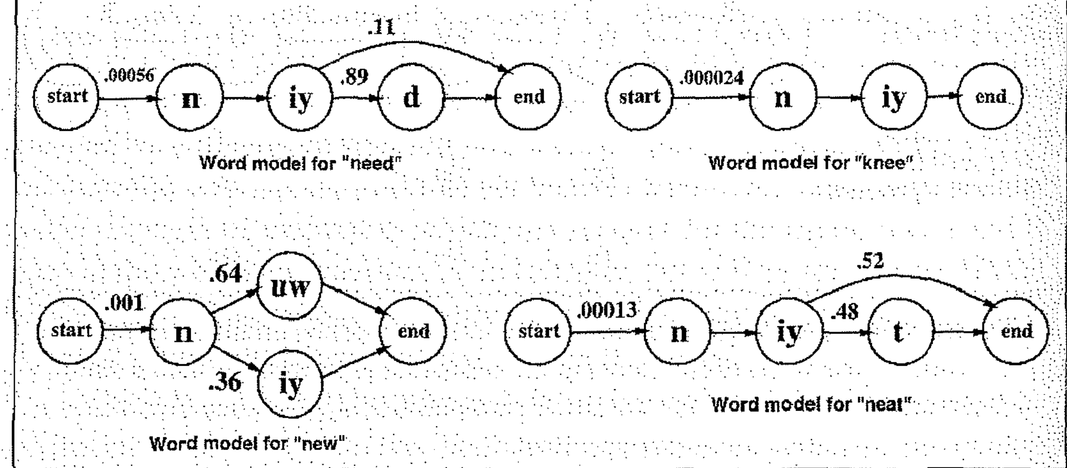

Figure 5.15 Pronunciation networks for the words *need*, *neat*, *new*, and *knee*. All networks are simplified from the actual pronunciations in the Switchboard corpus. Each network has been augmented by the unigram probability of the word (i.e., its normalized frequency from the Switchboard+Brown corpus). Word probabilities are not usually included as part of the pronunciation network for a word; they are added here to simplify the exposition of the forward algorithm.

The forward algorithm is another **dynamic programming** algorithm, and can be thought of as a slight generalization of the minimum edit distance algorithm. Like the minimum edit distance algorithm, it uses a table to store intermediate values as it builds up the probability of the observation sequence. Unlike the minimum edit distance algorithm, the rows are labeled not just by states which always occur in linear order, but implicitly by a *state-graph* which has many ways of getting from one state to another. In the minimum edit distance algorithm, we filled in the matrix by just computing the value of each cell from the three cells around it. With the forward algorithm, on the other hand, a state might be entered by any other state, and so the recurrence relation is somewhat more complicated. Furthermore, the forward algorithm computes the *sum* of the probabilities of all possible paths that could generate the observation sequence, where the minimum edit distance computed the *minimum* such probability.[5] Each cell of the forward algorithm matrix, *forward*$[t, j]$ represents the probability of being in state j after seeing the first t observations, given the automaton λ. Since we have augmented our graphs with the word probability $p(w)$, our example of the forward algorithm here is actually computing this likelihood times $p(w)$. The value of each cell *forward*$[t, j]$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$\textit{forward}[t,j] = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)\; P(w) \tag{5.14}$$

Here $q_t = j$ means "the probability that the tth state in the sequence of states is state j". We compute this probability by summing over the extensions of all the paths that lead to the current cell. An extension of a path from a state i at time $t-1$ is computed by multiplying the following three factors:

1. the **previous path probability** from the previous cell forward$[t-1, i]$,
2. the **transition probability** a_{ij} from previous state i to current state j, and
3. the **observation likelihood** b_{jt} that current state j matches observation symbol t. For the weighted automata that we consider here, b_{jt} is 1 if the observation symbol matches the state, and 0 otherwise. Chapter 7 will consider more complex observation likelihoods.

The algorithm is described in Figure 5.16.

Figure 5.17 shows the forward algorithm applied to the word *need*. The algorithm applies similarly to the other words which can produce the string [n iy], resulting in the probabilities on page 167. In order to compute the most probable underlying word, we run the forward algorithm separately on each of the candidate words, and choose the one with the highest probability. Chapter 7 will give further details of the mathematics of the forward algorithm and introduce the related forward-backward algorithm.

[5] The forward algorithm computes the *sum* because there may be multiple paths through the network which explain a given observation sequence. Chapter 7 will take up this point in more detail.

```
function FORWARD(observations,state-graph) returns forward-probability

  num-states ← NUM-OF-STATES(state-graph)
  num-obs ← length(observations)
  Create probability matrix forward[num-states + 2, num-obs + 2]
  forward[0,0] ← 1.0
  for each time step t from 0 to num-obs do
    for each state s from 0 to num-states do
      for each transition s' from s specified by state-graph
        $forward[s',t+1] \leftarrow forward[s,t] * a[s,s'] * b[s',o_t]$
  return the sum of the probabilities in the final column of forward
```

Figure 5.16 The forward algorithm for computing likelihood of observation sequence given a word model. $a[s,s']$ is the transition probability from current state s to next state s', and $b[s',o_t]$ is the observation likelihood of s' given o_t. For the weighted automata that we consider here, $b[s',o_t]$ is 1 if the observation symbol matches the state, and 0 otherwise.

	end				.00056* .11 = .00062
	d				
need	**iy**			.00056 * 1.0 = .00056	
	n		.00056 * 1.0 = .00056		
	start	1.0			
		#	**n**	**iy**	**#**

Figure 5.17 The forward algorithm applied to the word *need*, computing the probability $P(O|w)P(w)$. While this example doesn't require the full power of the forward algorithm, we will see its use on more complex examples in Chapter 7.

Decoding: The Viterbi Algorithm

The forward algorithm as we presented it seems a bit of an overkill. Since only one path through the pronunciation networks will match the input string, why use such a big matrix and consider so many possible paths? Furthermore, as a decoding method, it seems rather inefficient to run the forward algorithm once for each word (imagine how inefficient this would be if we were computing likelihoods for all possible sentences rather than all possible words!) Part of the reason that the forward algorithm seems like overkill is that we have immensely simplified the pronunciation problem by assuming that our input consists of sequences of unambiguous symbols. We will see in Chapter 7 that when the observation sequence is a set of noisy acoustic values, there are many possibly paths through the automaton, and the forward algorithm will play an important role in summing these paths.

But it is true that having to run it separately on each word makes the forward algorithm a very inefficient decoding method. Luckily, there is a simple variation on the forward algorithm called the **Viterbi** algorithm which allows us to consider all the words simultaneously and still compute the most likely path. The term **Viterbi** is common in speech and language processing, but like the forward algorithm this is really a standard application of the classic **dynamic programming** algorithm, and again looks a lot like the **minimum edit distance** algorithm. The Viterbi algorithm was first applied to speech recognition by Vintsyuk (1968), but has what Kruskal (1983) calls a 'remarkable history of multiple independent discovery and publication'; see the History section at the end of the chapter for more details. The name Viterbi is the one which is most commonly used in speech recognition, although the terms **DP alignment** (for **Dynamic Programming alignment**), **dynamic time warping** and **one-pass decoding** are also commonly used. The term is applied to the decoding algorithm for weighted automata and Hidden Markov Models on a single word and also to its more complex application to continuous speech, as we will see in Chapter 7. In this chapter we will show how the algorithm is used to find the best path through a network composed of single words, as a result choosing the word which is most probable given the observation sequence string of words.

VITERBI

DYNAMIC TIME WARPING

The version of the Viterbi algorithm that we will present takes as input a single weighted automaton and a set of observed phones $o = (o_1 o_2 o_3 \ldots o_t)$ and returns the most probable state sequence $q = (q_1 q_2 q_3 \ldots q_t)$, together with its probability. We can create a single weighted automaton by combining the pronunciation networks for the four words in parallel with a single start and a single end state. Figure 5.18 shows the combined network.

Figure 5.19 shows pseudocode for the Viterbi algorithm. Like the minimum edit distance and forward algorithm, the Viterbi algorithm sets up a probability matrix, with one column for each time index t and one row for each state in the state graph. Also like the forward algorithm, each column has a cell for each state q_i in the single combined automaton for the four words. In fact, the code for the Viterbi algorithm should look exactly like the code for the forward algorithm with two modifications. First, where

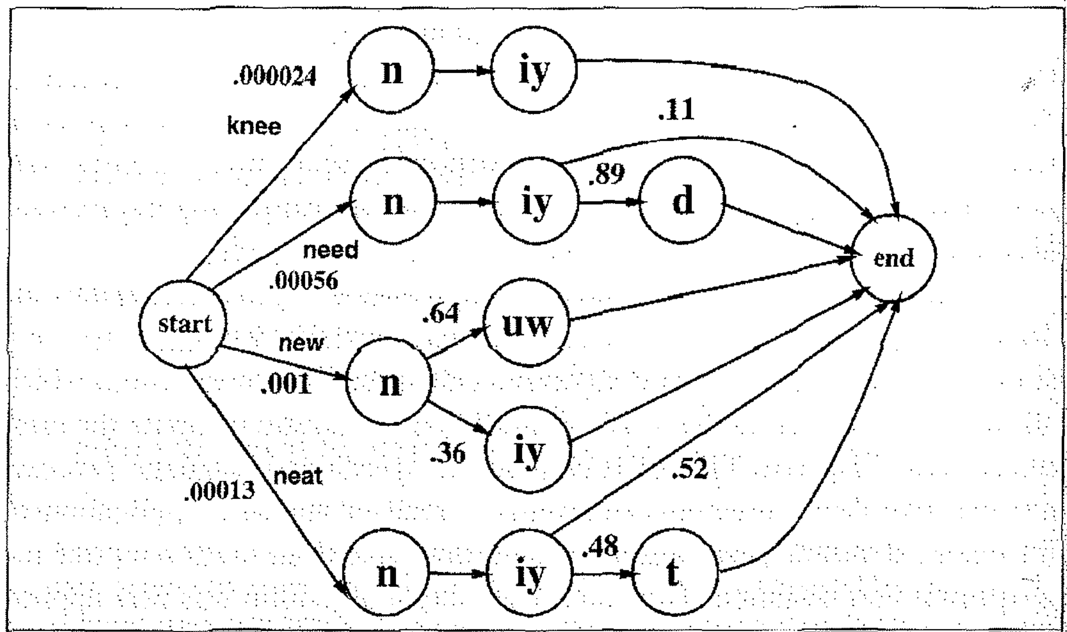

Figure 5.18 The pronunciation networks for the words *need*, *neat*, *new*, and *knee* combined into a single weighted automaton. Again, word probabilities are not usually considered part of the pronunciation network for a word; they are added here to simplify the exposition of the Viterbi algorithm.

the forward algorithm places the *sum* of all previous paths into the current cell, the Viterbi algorithm puts the *max* of the previous paths into the current cell.

The algorithm first creates $N+2$ or four state columns. The first column is an initial pseudo-observation, the second corresponds to the first observation phone [n], the third to [iy] and the fourth to a final pseudo-observation. We begin in the first column by setting the probability of the *start* state to 1.0, and the other probabilities to 0; the reader should find this in Figure 5.20. Cells with probability 0 are simply left blank for readability.

Then we move to the next state; as with the forward algorithm, for every state in column 0, we compute the probability of moving into each state in column 1. The value $viterbi[t, j]$ is computed by taking the maximum over the extensions of all the paths that lead to the current cell. An extension of a path from a state i at time $t-1$ is computed by multiplying the same three factors we used for the forward algorithm:

1. the **previous path probability** from the previous cell $forward[t-1, i]$,
2. the **transition probability** a_{ij} from previous state i to current state j, and
3. the **observation likelihood** b_{jt} that current state j matches observation symbol t. For the weighted automata that we consider here, b_{jt} is 1 if

```
function VITERBI(observations of len T,state-graph) returns best-path
  num-states ← NUM-OF-STATES(state-graph)
  Create a path probability matrix viterbi[num-states+2,T+2]
  viterbi[0,0] ← 1.0
  for each time step t from 0 to T do
    for each state s from 0 to num-states do
      for each transition s' from s specified by state-graph
        new-score ← viterbi[s, t] * a[s,s'] * $b_{s'}(o_t)$
        if ((viterbi[s',t+1] = 0) || (new-score > viterbi[s', t+1]))
          then
              viterbi[s', t+1] ← new-score
              back-pointer[s', t+1] ← s
  Backtrace from highest probability state in the final column of viterbi[] and
return path
```

Figure 5.19 Viterbi algorithm for finding optimal sequence of states in continuous speech recognition, simplified by using phones as inputs. Given an observation sequence of phones and a weighted automaton (state graph), the algorithm returns the path through the automaton which has maximum probability and accepts the observation sequence. $a[s,s']$ is the transition probability from current state s to next state s', and $b[s',o_t]$ is the observation likelihood of s' given o_t. For the weighted automata that we consider here, $b[s',o_t]$ is 1 if the observation symbol matches the state, and 0 otherwise.

the observation symbol matches the state, and 0 otherwise. Chapter 7 will consider more complex observation likelihoods.

In Figure 5.20, in the column for the input *n*, each word starts with [n], and so each has a non-zero probability in the cell for the state *n*. Other cells in that column have zero entries, since their states don't match n. When we proceed to the next column, each cell that matches iy gets updated with the contents of the previous cell times the transition probability to that cell. Thus the value of *viterbi*[2,iy_{new}] for the iy state of the word *new* is the product of the "word" probability of *new* times the probability of *new* being pronounced with the vowel *iy*. Notice that if we look only at this iy column, that the word *need* is currently the "most-probable" word. But when we move to the final column, the word *new* will win out, since *need* has a smaller transition probability to *end* (.11) than *new* does (1.0). We can now follow the backpointers and backtrace to find the path that gave us this final probability of .00036.

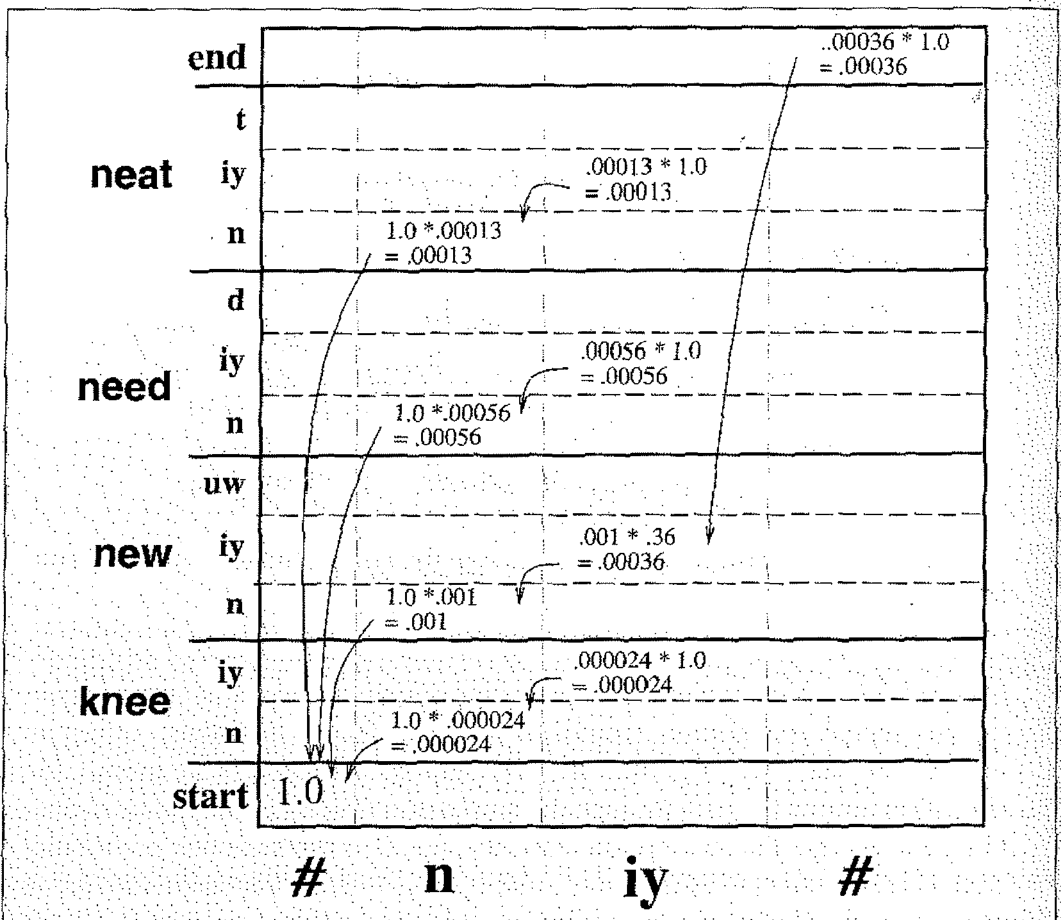

Figure 5.20 The entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. Backtracing from the *end* state, we can reconstruct the state sequence n_{new} iy_{new}, arriving at the best word *new*.

Weighted Automata and Segmentation

SEGMENTATION

Weighted automata and the Viterbi algorithm play an important in various algorithm for **segmentation**. Segmentation is the process of taking an undifferentiated sequence of symbols and "segmenting" it into chunks. For example **sentence segmentation** is the problem of automatically finding the sentence boundaries in a corpus. Similarly **word segmentation** is the problem of finding word-boundaries in a corpus. In written English there is no difficulty in segmenting words from each other because there are orthographic spaces between words. This is not the case in languages like Chinese and Japanese that use a Chinese-derived writing system. Written Chinese does not mark word boundaries. Instead, each Chinese character is written one after the other without spaces. Since each character approximately represents a single morpheme, and since words can be composed of one or more characters, it is often difficult to know where words should be segmented. Proper word-segmentation is necessary for many applications, particularly including parsing and text-to-speech. (How a sentence is broken up into words influences its pronunciation in a number of ways.)

Consider the following example sentence from Sproat et al. (1996):

(5.15) 日文章鱼怎麽说？
"How do you say 'octopus' in Japanese?"

This sentence has two potential segmentations, only one of which is correct. In the plausible segmentation, the first two characters are combined to make the word for 'Japanese language' (日文 rì-wén) (the accents indicate the **tone** of each syllable), and the next two are combined to make the word for 'octopus' (章鱼 zhāng-yú).

(5.16)

日文	章鱼	怎麽	说	?
rì-wén	zhāng-yú	zěn-me	shuō	
Japanese	octopus	how	say	

"How do you say octopus in Japanese?"

(5.17)

日	文章	鱼	怎麽	说	?
rì	wén-zhāng	yú	zěn-me	shuō	
Japan	essay	fish	how	say	

"How do you say Japan essay fish?"

Sproat et al. (1996) give a very simple algorithm which selects the correct segmentation by choosing the one which contains the most-frequent words. In other words, the algorithm multiplies together the probabilities of each word in a potential segmentation and chooses whichever segmentation results in a higher product probability.

The implementation of their algorithm combines a weighted-finite-state transducer representation of a Chinese lexicon with the Viterbi algorithm. This lexicon is a slight augmentation of the FST lexicons we saw in Chapter 4; each word is represented as a series of arcs representing each character in the word, followed by a weighted arc representing the probability of the word. As is commonly true with probabilistic algorithms, they actually use the negative log probability of the word ($-\log(P(w))$. The log probability is mainly useful because the product of many probabilities gets very small, and so using the log probability can help avoid underflow. Using log probabilities also means that we are *adding costs* rather than *multiplying* *probabilities*, and that we are looking for the *minimum cost* solution rather than the *maximum probability* solution.

Consider the example in Figure 5.21. This sample lexicon Figure 5.21(a) consists of only five potential words:

Word	Pronunciation	Meaning	Cost ($-logp(w)$)
日文	rì-wén	'Japanese'	10.63
日	rì	'Japan'	6.51
章鱼	zhāng- yú	'octopus'	13.18
文章	wén-zhāng	'essay'	9.51
鱼	yú	'fish'	10.28

The system represents the input sentence as the unweighted FSA in Figure 5.21(b). In order to compose this input with the lexicon, it needs to be converted into an FST. The algorithm uses a function *Id* which takes an FSA *A* and returns the FST which maps all and only the strings accepted by *A* to themselves. Let $D*$ represent the transitive closure of D, that is, the automaton created by adding a loop from the end of the lexicon back to the beginning. The set of all possible segmentations is $Id(I) \circ D^*$, that is, the input transducer $Id(I)$ composed with the transitive closure of the dictionary *D*, shown in Figure 5.21(c). Then the best segmentation is the lowest-cost segmentation in $Id(I) \circ D^*$, shown in Figure 5.21(d).

Finding the best path shown in Figure 5.21(d) can be done easily with the Viterbi algorithm and is left as an exercise for the reader. Furthermore, this segmentation algorithm, like the spelling error correction algorithm we saw earlier, can also be extended to incorporate the cross-word probabilities (*N*-gram probabilities) that will be introduced in Chapter 6.

Segmentation for Lexicon-Induction

The weighted automata segmentation algorithm that was presented above relies on the weights stored in the lexicon. But how is this lexicon to be learned in the first place? A number of segmentation algorithms address this "prior" problem of segmentation in the absence of a lexicon. For example de Marcken (1996) and Brent and Cartwright (1996) both propose algorithms that take an unsegmented sequence of input phones and use information-theoretic principles to iteratively induce the lexicon by trying different possible segmentations. Both rely on stochastic versions of the **Minimum Description Length** (**MDL**) principle and on phonotactic transition probabilities to choose between alternative models. The description length of a lexicon

MDL

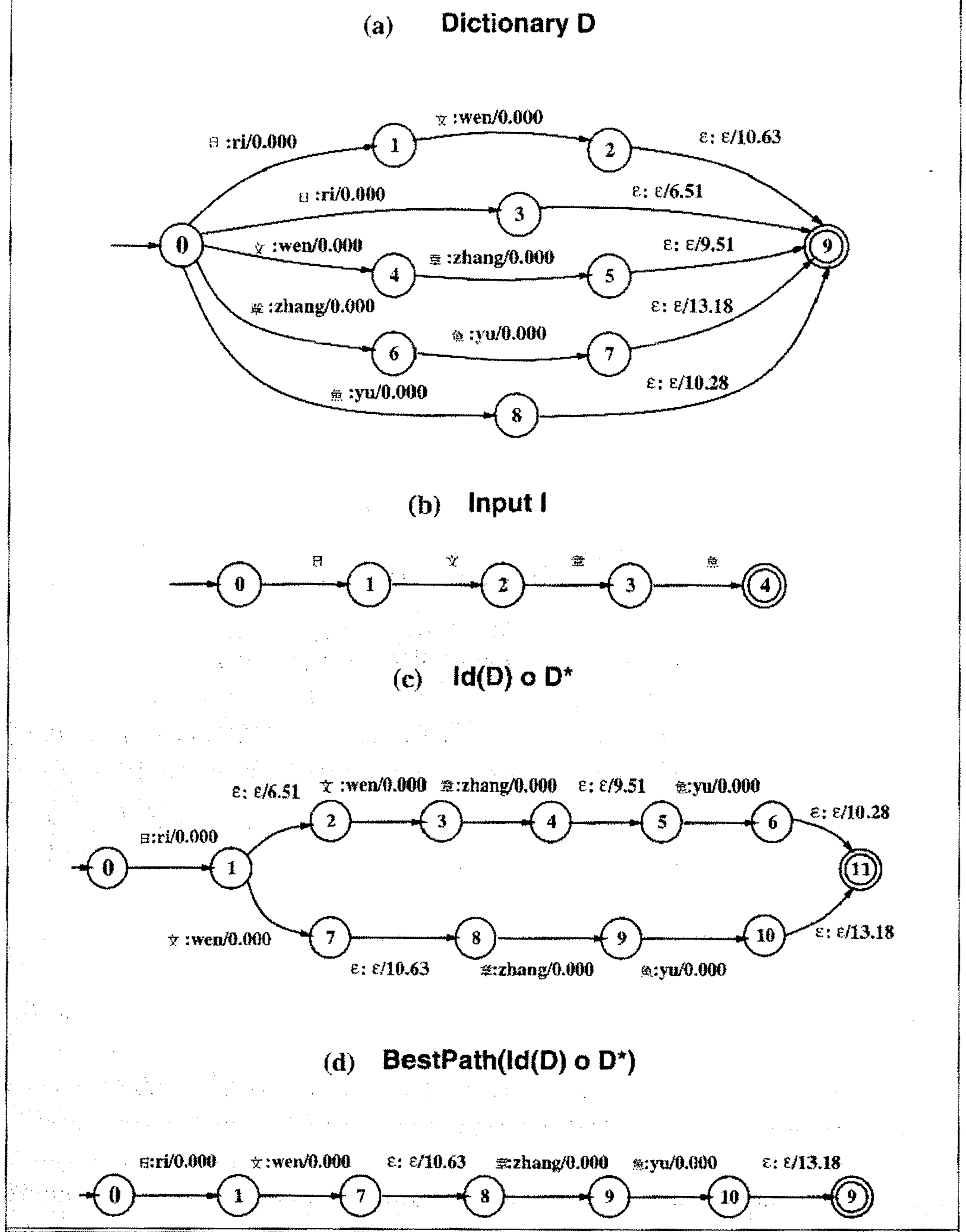

Figure 5.21 The Sproat et al. (1996) algorithm applied to four input words (after Sproat et al. (1996))

or grammar (measured, for example, in the number of symbols in it) is a heuristic measure of the information complexity in the lexicon. By preferring a lexicon with less symbols, MDL is implicitly choosing a simpler and