Speech and Language Processing: An introduction to speech recognition, natural language processing, and computational linguistics. Daniel Jurafsky & James H. Martin. Copyright © 2006, All rights reserved. Draft of February 21, 2007. Do not cite without permission.

3 WORDS & TRANSDUCERS

How can there be any sin in sincere?
Where is the good in goodbye?
Meredith Wilson, *The Music Man*

Ch. 2 introduced the regular expression, showing for example how a single search string could help a web search engine find both *woodchuck* and *woodchucks*. Hunting for singular or plural woodchucks was easy; the plural just tacks an *s* on to the end. But suppose we were looking for another fascinating woodland creatures; let's say a *fox*, and a *fish*, that surly *peccary* and perhaps a Canadian *wild goose*. Hunting for the plurals of these animals takes more than just tacking on an *s*. The plural of *fox* is *foxes*; of *peccary*, *peccaries*; and of *goose*, *geese*. To confuse matters further, fish don't usually change their form when they are plural[1].

It takes two kinds of knowledge to correctly search for singulars and plurals of these forms. **Orthographic rules** tell us that English words ending in *-y* are pluralized by changing the *-y* to *-i-* and adding an *-es*. **Morphological rules** tell us that *fish* has a null plural, and that the plural of *goose* is formed by changing the vowel.

MORPHOLOGICAL PARSING

The problem of recognizing that a word (like *foxes*) breaks down into component morphemes (*fox* and *-es*) and building a structured representation of this fact is called **morphological parsing**.

PARSING

Parsing means taking an input and producing some sort of linguistic structure for it. We will use the term parsing very broadly throughout this book, including many kinds of structures that might be produced; morphological, syntactic, semantic, discourse; in the form of a string, or a tree, or a network. Morphological parsing or stemming applies to many affixes other than plurals; for example we might need to take any English verb form ending in *-ing* (*going*, *talking*, *congratulating*) and parse it into its verbal stem plus the *-ing* morpheme. So given the **surface** or **input form** *going*, we might want to produce the parsed form `VERB-go + GERUND-ing`.

SURFACE

Morphological parsing is important throughout speech and language processing. It plays a crucial role in part-of-speech tagging for morphologically complex languages like Russian or German, as we will see in Ch. 5. It is important for producing the large dictionaries that are necessary for robust spell-checking. We will need it in machine

[1] (see e.g., Seuss (1960))

translation to realize for example that the French words *va* and *aller* should both translate to forms of the English verb *go*.

To solve the morphological parsing problem, why couldn't we just store all the plural forms of English nouns and *-ing* forms of English verbs in a dictionary and do parsing by lookup? The major reason is that *-ing* is a **productive** suffix; by this we mean that it applies to every verb. Similarly *-s* applies to almost every noun. Productive suffixes even apply to new words; thus the new word *fax* can automatically be used in the *-ing* form: *faxing*. Since new words (particularly acronyms and proper nouns) are created every day, the class of nouns in English increases constantly, and we need to be able to add the plural morpheme *-s* to each of these. Additionally, the plural form of these new nouns depends on the spelling/pronunciation of the singular form; for example if the noun ends in *-z* then the plural form is *-es* rather than *-s*. We'll need to encode these rules somewhere.

PRODUCTIVE

Finally, we certainly cannot list all the morphological variants of every word in morphologically complex languages like Turkish, which has words like:

(3.1) uygarlaştıramadıklarımızdanmışsınızcasına
uygar +laş +tır +ama +dık +lar +ımız +dan +mış +sınız +casına
civilized +BEC +CAUS +NABL +PART +PL +P1PL +ABL +PAST +2PL +AsIf
"(behaving) as if you are among those whom we could not civilize"

The various pieces of this word (the **morphemes**) have these meanings:

+BEC	"become"
+CAUS	the causative verb marker ('cause to X')
+NABL	"not able"
+PART	past participle form
+P1PL	1st person pl possessive agreement
+2PL	2nd person pl
+ABL	ablative (from/among) case marker
+AsIf	derivationally forms an adverb from a finite verb

Not all Turkish words look like this; the average Turkish word has about three morphemes. But such long words do exist; indeed Kemal Oflazer, who came up with this example, notes (p.c.) that verbs in Turkish have 40,000 possible forms not counting derivational suffixes. Adding derivational suffixes, such as causatives, allows a theoretically infinite number of words, since causativization can be repeated in a single word (*You cause X to cause Y to . . . do W*). Thus we cannot store all possible Turkish words in advance, and must do morphological parsing dynamically.

In the next section we survey morphological knowledge for English and some other languages. We then introduce the key algorithm for morphological parsing, the **finite-state transducer**. Finite-state transducers are a crucial technology throughout speech and language processing, so we will return to them again in later chapters.

After describing morphological parsing, we will introduce some related algorithms in this chapter. In some applications we don't need to parse a word, but we do need to map from the word to its root or stem. For example in information retrieval and web search (IR), we might want to map from *foxes* to *fox*; but might not need to also know that *foxes* is plural. Just stripping off such word endings is called **stemming** in IR. We will describe a simple stemming algorithm called the **Porter stemmer**.

STEMMING

LEMMATIZATION

For other speech and language processing tasks, we need to know that two words have a similar root, despite their surface differences. For example the words *sang*, *sung*, and *sings* are all forms of the verb *sing*. The word *sing* is sometimes called the common *lemma* of these words, and mapping from all of these to *sing* is called **lemmatization**.[2]

TOKENIZATION

Next, we will introduce another task related to morphological parsing. **Tokenization** or **word segmentation** is the task of separating out (tokenizing) words from running text. In English, words are often separated from each other by blanks (whitespace), but whitespace is not always sufficient; we'll need to notice that *New York* and *rock 'n' roll* are individual words despite the fact that they contain spaces, but for many applications we'll need to separate *I'm* into the two words *I* and *am*.

Finally, we introduce the important **minimum edit distance** algorithm, and show how it can be used in spell-checking.

3.1 Survey of (Mostly) English Morphology

MORPHEMES

Morphology is the study of the way words are built up from smaller meaning-bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language. So for example the word *fox* consists of a single morpheme (the morpheme *fox*) while the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*.

STEMS

AFFIXES

As this example suggests, it is often useful to distinguish two broad classes of morphemes: **stems** and **affixes**. The exact details of the distinction vary from language to language, but intuitively, the stem is the "main" morpheme of the word, supplying the main meaning, while the affixes add "additional" meanings of various kinds.

Affixes are further divided into **prefixes**, **suffixes**, **infixes**, and **circumfixes**. Prefixes precede the stem, suffixes follow the stem, circumfixes do both, and infixes are inserted inside the stem. For example, the word *eats* is composed of a stem *eat* and the suffix *-s*. The word *unbuckle* is composed of a stem *buckle* and the prefix *un-*. English doesn't have any good examples of circumfixes, but many other languages do. In German, for example, the past participle of some verbs is formed by adding *ge-* to the beginning of the stem and *-t* to the end; so the past participle of the verb *sagen* (to say) is *gesagt* (said). Infixes, in which a morpheme is inserted in the middle of a word, occur very commonly for example in the Philipine language Tagalog. For example the affix *um*, which marks the agent of an action, is infixed to the Tagalog stem *hingi* "borrow" to produce *humingi*. There is one infix that occurs in some dialects of English in which the taboo morphemes "f**king" or "bl**dy" or others like them are inserted in the middle of other words ("Man-f**king-hattan", "abso-bl**dy-lutely"[3]) (McCawley, 1978).

A word can have more than one affix. For example, the word *rewrites* has the prefix *re-*, the stem *write*, and the suffix *-s*. The word *unbelievably* has a stem (*believe*)

[2] Lemmatization is actually more complex, since it sometimes involves deciding on which sense of a word is present. We return to this issue in Ch. 19.

[3] Alan Jay Lerner, the lyricist of My Fair Lady, bowdlerized the latter to *abso-bloomin'lutely* in the lyric to "Wouldn't It Be Loverly?" (Lerner, 1978, p. 60).

plus three affixes (*un-*, *-able*, and *-ly*). While English doesn't tend to stack more than four or five affixes, languages like Turkish can have words with nine or ten affixes, as we saw above. Languages that tend to string affixes together like Turkish does are called **agglutinative** languages.

There are many ways to combine morphemes to create words. Four of these methods are common and play important roles in speech and language processing: **inflection**, **derivation**, **compounding**, and **cliticization**.

INFLECTION
DERIVATION
COMPOUNDING
CLITICIZATION

Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. For example, English has the inflectional morpheme *-s* for marking the **plural** on nouns, and the inflectional morpheme *-ed* for marking the past tense on verbs. **Derivation** is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly. For example the verb *computerize* can take the derivational suffix *-ation* to produce the noun *computerization*. **Compounding** is the combination of multiple word stems together. For example the noun *doghouse* is the concatenation of the morpheme *dog* with the morpheme *house*. Finally, **cliticization** is the combination of a word stem with a **clitic**. A clitic is a morpheme that acts syntactically like a word, but is reduced in form and attached (phonologically and sometimes orthographically) to another word. For example the English morpheme *'ve* in the word *I've* is a clitic, as is the French definite article *l'* in the word *l'opera*. In the following sections we give more details on these processes.

CLITIC

3.1.1 Inflectional Morphology

English has a relatively simple inflectional system; only nouns, verbs, and sometimes adjectives can be inflected, and the number of possible inflectional affixes is quite small.

PLURAL
SINGULAR

English nouns have only two kinds of inflection: an affix that marks **plural** and an affix that marks **possessive**. For example, many (but not all) English nouns can either appear in the bare stem or **singular** form, or take a plural suffix. Here are examples of the regular plural suffix *-s* (also spelled *-es*), and irregular plurals:

	Regular Nouns		Irregular Nouns	
Singular	cat	thrush	mouse	ox
Plural	cats	thrushes	mice	oxen

While the regular plural is spelled *-s* after most nouns, it is spelled *-es* after words ending in *-s* (*ibis/ibises*), *-z* (*waltz/waltzes*), *-sh* (*thrush/thrushes*), *-ch* (*finch/finches*), and sometimes *-x* (*box/boxes*). Nouns ending in *-y* preceded by a consonant change the *-y* to *-i* (*butterfly/butterflies*).

The possessive suffix is realized by apostrophe + *-s* for regular singular nouns (*llama's*) and plural nouns not ending in *-s* (*children's*) and often by a lone apostrophe after regular plural nouns (*llamas'*) and some names ending in *-s* or *-z* (*Euripides' comedies*).

REGULAR

English verbal inflection is more complicated than nominal inflection. First, English has three kinds of verbs; **main verbs**, (*eat, sleep, impeach*), **modal verbs** (*can, will, should*), and **primary verbs** (*be, have, do*) (using the terms of Quirk et al., 1985). In this chapter we will mostly be concerned with the main and primary verbs, because it is these that have inflectional endings. Of these verbs a large class are **regular**, that is to say all verbs of this class have the same endings marking the same functions. These regular verbs (e.g. *walk*, or *inspect*) have four morphological forms, as follow:

Morphological Form Classes	Regularly Inflected Verbs			
stem	walk	merge	try	map
-*s* form	walks	merges	tries	maps
-*ing* participle	walking	merging	trying	mapping
Past form or -*ed* participle	walked	merged	tried	mapped

These verbs are called regular because just by knowing the stem we can predict the other forms by adding one of three predictable endings and making some regular spelling changes (and as we will see in Ch. 7, regular pronunciation changes). These regular verbs and forms are significant in the morphology of English first because they cover a majority of the verbs, and second because the regular class is **productive**. As discussed earlier, a productive class is one that automatically includes any new words that enter the language. For example the recently-created verb *fax* (*My mom faxed me the note from cousin Everett*) takes the regular endings -*ed*, -*ing*, -*es*. (Note that the -*s* form is spelled *faxes* rather than *faxs*; we will discuss spelling rules below).

IRREGULAR VERBS

PRETERITE

The **irregular verbs** are those that have some more or less idiosyncratic forms of inflection. Irregular verbs in English often have five different forms, but can have as many as eight (e.g., the verb *be*) or as few as three (e.g. *cut* or *hit*). While constituting a much smaller class of verbs (Quirk et al. (1985) estimate there are only about 250 irregular verbs, not counting auxiliaries), this class includes most of the very frequent verbs of the language.[4] The table below shows some sample irregular forms. Note that an irregular verb can inflect in the past form (also called the **preterite**) by changing its vowel (*eat/ate*), or its vowel and some consonants (*catch/caught*), or with no change at all (*cut/cut*).

Morphological Form Classes	Irregularly Inflected Verbs		
stem	eat	catch	cut
-*s* form	eats	catches	cuts
-*ing* participle	eating	catching	cutting
Past form	ate	caught	cut
-*ed* participle	eaten	caught	cut

The way these forms are used in a sentence will be discussed in Chapters 8–12 but is worth a brief mention here. The -*s* form is used in the "habitual present" form to distinguish the third-person singular ending (*She jogs every Tuesday*) from the other

[4] In general, the more frequent a word form, the more likely it is to have idiosyncratic properties; this is due to a fact about language change; very frequent words tend to preserve their form even if other words around them are changing so as to become more regular.

PROGRESSIVE

GERUND

PERFECT

choices of person and number (*I/you/we/they jog every Tuesday*). The stem form is used in the infinitive form, and also after certain other verbs (*I'd rather walk home, I want to walk home*). The *-ing* participle is used in the **progressive** construction to mark present or ongoing activity (*It is raining*), or when the verb is treated as a noun; this particular kind of nominal use of a verb is called a **gerund** use: *Fishing is fine if you live near water.* The *-ed* participle is used in the **perfect** construction (*He's eaten lunch already*) or the passive construction (*The verdict was overturned yesterday*).

In addition to noting which suffixes can be attached to which stems, we need to capture the fact that a number of regular spelling changes occur at these morpheme boundaries. For example, a single consonant letter is doubled before adding the *-ing* and *-ed* suffixes (*beg/begging/begged*). If the final letter is "c", the doubling is spelled "ck" (*picnic/picnicking/picnicked*). If the base ends in a silent *-e*, it is deleted before adding *-ing* and *-ed* (*merge/merging/merged*). Just as for nouns, the *-s* ending is spelled *-es* after verb stems ending in *-s* (*toss/tosses*) , *-z*, (*waltz/waltzes*) *-sh*, (*wash/washes*) *-ch*, (*catch/catches*) and sometimes *-x* (*tax/taxes*). Also like nouns, verbs ending in *-y* preceded by a consonant change the *-y* to *-i* (*try/tries*).

The English verbal system is much simpler than for example the European Spanish system, which has as many as fifty distinct verb forms for each regular verb. Fig. 3.1 shows just a few of the examples for the verb *amar*, 'to love'. Other languages can have even more forms than this Spanish example.

	Present	Imperfect	Future	Preterite	Present	Conditional	Imperfect	Future
	Indicative	Indicative			Subjnct.		Subjnct.	Subjnct.
1SG	amo	amaba	amaré	amé	ame	amaría	amara	amare
2SG	amas	amabas	amarás	amaste	ames	amarías	amaras	amares
3SG	ama	amaba	amará	amó	ame	amaría	amara	amáreme
1PL	amamos	amábamos	amaremos	amamos	amemos	amaríamos	amáramos	amáremos
2PL	amáis	amabais	amaréis	amasteis	améis	amaríais	amarais	amareis
3PL	aman	amaban	amarán	amaron	amen	amarían	amaran	amaren

Figure 3.1 To love in Spanish. Some of the inflected forms of the verb *amar* in European Spanish. *1SG* stands for "first person singular", 3PL for "third person plural", and so on.

3.1.2 Derivational Morphology

While English inflection is relatively simple compared to other languages, derivation in English is quite complex. Recall that derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly.

NOMINALIZATION

A very common kind of derivation in English is the formation of new nouns, often from verbs or adjectives. This process is called **nominalization**. For example, the suffix *-ation* produces nouns from verbs ending often in the suffix *-ize* (*computerize* → *computerization*). Here are examples of some particularly productive English nominalizing suffixes.

Suffix	Base Verb/Adjective	Derived Noun
-ation	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

Adjectives can also be derived from nouns and verbs. Here are examples of a few suffixes deriving adjectives from nouns or verbs.

Suffix	Base Noun/Verb	Derived Adjective
-al	computation (N)	computational
-able	embrace (V)	embraceable
-less	clue (N)	clueless

Derivation in English is more complex than inflection for a number of reasons. One is that it is generally less productive; even a nominalizing suffix like -*ation*, which can be added to almost any verb ending in -*ize*, cannot be added to absolutely every verb. Thus we can't say **eatation* or **spellation* (we use an asterisk (*) to mark "non-examples" of English). Another is that there are subtle and complex meaning differences among nominalizing suffixes. For example *sincerity* has a subtle difference in meaning from *sincereness*.

3.1.3 Cliticization

Recall that a clitic is a unit whose status lies in between that of an affix and a word. The phonological behavior of clitics is like affixes; they tend to be short and unaccented (we will talk more about phonology in Ch. 8). Their syntactic behavior is more like words, often acting as pronouns, articles, conjunctions, or verbs. Clitics preceding a word are called **proclitics**, while those following are **enclitics**.

PROCLITICS

ENCLITICS

English clitics include these auxiliary verbal forms:

Full Form	Clitic	Full Form	Clitic
am	'm	have	've
are	're	has	's
is	's	had	'd
will	'll	would	'd

Note that the clitics in English are ambiguous; Thus *she's* can mean *she is* or *she has*. Except for a few such ambiguities, however, correctly segmenting off clitics in English is simplified by the presence of the apostrophe. Clitics can be harder to parse in other languages. In Arabic and Hebrew, for example, the definite article (*the*; *Al* in Arabic, *ha* in Hebrew) is cliticized on to the front of nouns. It must be segmented off in order to do part-of-speech tagging, parsing, or other tasks. Other Arabic proclitics include prepositions like *b* 'by/with', and conjunctions like *w* 'and'. Arabic also has *enclitics* marking certain pronouns. For example the word *and by their virtues* has clitics meaning *and*, *by*, and *their*, a stem *virtue*, and a plural affix. Note that since

Arabic is read right to left, these would actually appear ordered from right to left in an Arabic word.

	proclitic	proclitic	stem	affix	enclitic
Arabic	w	b	Hsn	At	hm
Gloss	and	by	virtue	s	their

3.1.4 Non-concatenative Morphology

CONCATENATIVE

The kind of morphology we have discussed so far, in which a word is composed of a string of morphemes concatenated together is often called **concatenative morphology**. A number of languages have extensive **non-concatenative morphology**, in which morphemes are combined in more complex ways. The Tagalog infixation example above is one example of non-concatenative morphology, since two morphemes (*hingi* and *um*) are intermingled.

Another kind of non-concatenative morphology is called **templatic morphology** or **root-and-pattern** morphology. This is very common in Arabic, Hebrew, and other Semitic languages. In Hebrew, for example, a verb is constructed using two components: a root, consisting usually of three consonants (CCC) and carrying the basic meaning, and a template, which gives the ordering of consonants and vowels and specifies more semantic information about the resulting verb, such as the semantic voice (e.g., active, passive, middle). For example the Hebrew tri-consonantal root *lmd*, meaning 'learn' or 'study', can be combined with the active voice CaCaC template to produce the word *lamad*, 'he studied', or the intensive CiCeC template to produce the word *limed*, 'he taught', or the intensive passive template CuCaC to produce the word *lumad*, 'he was taught'. Arabic and Hebrew combine this templatic morphology with concatenative morphology (like the cliticization example shown in the previous section).

3.1.5 Agreement

AGREE

GENDER

We introduced the plural morpheme above, and noted that plural is marked on both nouns and verbs in English. We say that the subject noun and the main verb in English have to **agree** in number, meaning that the two must either be both singular or both plural. There are other kinds of agreement processes. For example nouns, adjectives, and sometimes verbs in many languages are marked for **gender**. A gender is a kind of equivalence class that is used by the language to categorize the nouns; each noun falls into one class. Many languages (for example Romance languages like French, Spanish, or Italian) have 2 genders, which are referred to as masculine and feminine. Other languages (like most Germanic and Slavic languages) have three (masculine, feminine, neuter). Some languages, for example the Bantu languages of Africa, have as many as 20 genders. When the number of classes is very large, we often refer to them as **noun classes** instead of genders.

NOUN CLASSES

Gender is sometimes marked explicitly on a noun; for example Spanish masculine words often end in *-o* and feminine words in *-a*. But in many cases the gender is not marked in the letters or phones of the noun itself. Instead, it is a property of the word that must be stored in a lexicon. We will see an example of this in Fig. 3.2.

3.2 FINITE-STATE MORPHOLOGICAL PARSING

Let’s now proceed to the problem of parsing morphology. Our goal will be to take input forms like those in the first and third columns of Fig. 3.2, produce output forms like those in the second and fourth column.

English		Spanish		
Input	Morphologically Parsed Output	Input	Morphologically Parsed Output	Gloss
cats	cat +N +PL	pavos	pavo +N +Masc +Pl	‘ducks’
cat	cat +N +SG	pavo	pavo +N +Masc +Sg	‘duck’
cities	city +N +Pl	bebo	beber +V +PInd +1P +Sg	‘I drink’
geese	goose +N +Pl	canto	cantar +V +PInd +1P +Sg	‘I sing’
goose	goose +N +Sg	canto	canto +N +Masc +Sg	‘song’
goose	goose +V	puse	poner +V +Perf +1P +Sg	‘I was able’
gooses	goose +V +1P +Sg	vino	venir +V +Perf +3P +Sg	‘he/she came’
merging	merge +V +PresPart	vino	vino +N +Masc +Sg	‘wine’
caught	catch +V +PastPart	lugar	lugar +N +Masc +Sg	‘place’
caught	catch +V +Past			

Figure 3.2 Output of a morphological parse for some English and Spanish words. Spanish output modified from the Xerox XRCE finite-state language tools.

FEATURES

The second column contains the stem of each word as well as assorted morphological **features**. These features specify additional information about the stem. For example the feature +N means that the word is a noun; +Sg means it is singular, +Pl that it is plural. Morphological features will be referred to again in Ch. 5 and in more detail in Ch. 13; for now, consider +Sg to be a primitive unit that means “singular”. Spanish has some features that don’t occur in English; for example the nouns *lugar* and *pavo* are marked +Masc (masculine). Because Spanish nouns agree in gender with adjectives, knowing the gender of a noun will be important for tagging and parsing.

Note that some of the input forms (like *caught*, *goose*, *canto*, or *vino*) will be ambiguous between different morphological parses. For now, we will consider the goal of morphological parsing merely to list all possible parses. We will return to the task of disambiguating among morphological parses in Ch. 5.

In order to build a morphological parser, we’ll need at least the following:

LEXICON

1. **lexicon:** the list of stems and affixes, together with basic information about them (whether a stem is a Noun stem or a Verb stem, etc.).

MORPHOTACTICS

2. **morphotactics:** the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. For example, the fact that the English plural morpheme follows the noun rather than preceding it is a morphotactic fact.
3. **orthographic rules:** these **spelling rules** are used to model the changes that occur in a word, usually when two morphemes combine (e.g., the *y* → *ie* spelling rule discussed above that changes *city* + *-s* to *cities* rather than *ENDNOSPELL*).

The next section will discuss how to represent a simple version of the lexicon just for the sub-problem of morphological recognition, including how to use FSAs to model morphotactic knowledge.

In following sections we will then introduce the finite-state transducer (FST) as a way of modeling morphological features in the lexicon, and addressing morphological parsing. Finally, we show how to use FSTs to model orthographic rules.

3.3 Building a Finite-State Lexicon

A lexicon is a repository for words. The simplest possible lexicon would consist of an explicit list of every word of the language (*every* word, i.e., including abbreviations ("AAA") and proper names ("Jane" or "Beijing")) as follows:

a, AAA, AA, Aachen, aardvark, aardwolf, aba, abaca, aback, . . .

Since it will often be inconvenient or impossible, for the various reasons we discussed above, to list every word in the language, computational lexicons are usually structured with a list of each of the stems and affixes of the language together with a representation of the morphotactics that tells us how they can fit together. There are many ways to model morphotactics; one of the most common is the finite-state automaton. A very simple finite-state model for English nominal inflection might look like Fig. 3.3.

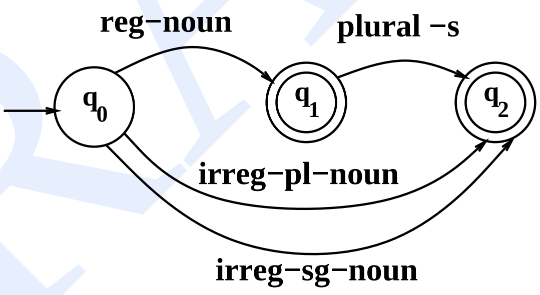

Figure 3.3 A finite-state automaton for English nominal inflection.

The FSA in Fig. 3.3 assumes that the lexicon includes regular nouns (**reg-noun**) that take the regular -*s* plural (e.g., *cat*, *dog*, *fox*, *aardvark*). These are the vast majority of English nouns since for now we will ignore the fact that the plural of words like *fox* have an inserted *e*: *foxes*. The lexicon also includes irregular noun forms that don't take -*s*, both singular **irreg-sg-noun** (*goose*, *mouse*) and plural **irreg-pl-noun** (*geese*, *mice*).

reg-noun	**irreg-pl-noun**	**irreg-sg-noun**	**plural**
fox	geese	goose	-s
cat	sheep	sheep	
aardvark	mice	mouse	

A similar model for English verbal inflection might look like Fig. 3.4.

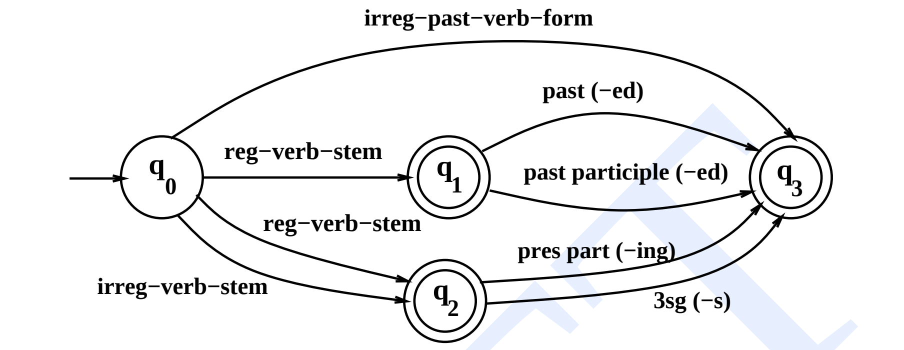

Figure 3.4 A finite-state automaton for English verbal inflection

This lexicon has three stem classes (reg-verb-stem, irreg-verb-stem, and irreg-past-verb-form), plus four more affix classes (-*ed* past, -*ed* participle, -*ing* participle, and third singular -*s*):

reg-verb-stem	**irreg-verb-stem**	**irreg-past-verb**	**past**	**past-part**	**pres-part**	**3sg**
walk fry talk impeach	cut speak sing	caught ate eaten sang	-ed	-ed	-ing	-s

English derivational morphology is significantly more complex than English inflectional morphology, and so automata for modeling English derivation tend to be quite complex. Some models of English derivation, in fact, are based on the more complex context-free grammars of Ch. 11 (Sproat, 1993; Orgun, 1995).

Consider a relatively simpler case of derivation: the morphotactics of English adjectives. Here are some examples from Antworth (1990):

big, bigger, biggest,
happy, happier, happiest, happily
unhappy, unhappier, unhappiest, unhappily
clear, clearer, clearest, clearly, unclear, unclearly

cool, cooler, coolest, coolly
red, redder, reddest
real, unreal, really

An initial hypothesis might be that adjectives can have an optional prefix (*un*-), an obligatory root (*big, cool*, etc.) and an optional suffix (-*er*, -*est*, or -*ly*). This might suggest the the FSA in Fig. 3.5.

Alas, while this FSA will recognize all the adjectives in the table above, it will also recognize ungrammatical forms like *unbig*, *unfast*, *oranger*, or *smally*. We need to set up classes of roots and specify their possible suffixes. Thus **adj-root**$_1$ would include adjectives that can occur with *un*- and -*ly* (*clear*, *happy*, and *real*) while **adj-root**$_2$ will include adjectives that can't (*big*, *small*), and so on.

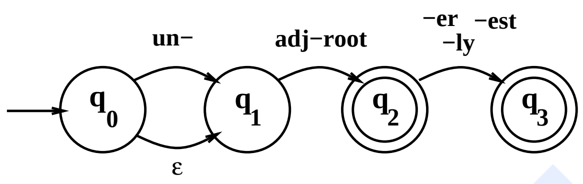

Figure 3.5 An FSA for a fragment of English adjective morphology: Antworth's Proposal #1.

This gives an idea of the complexity to be expected from English derivation. As a further example, we give in Figure 3.6 another fragment of an FSA for English nominal and verbal derivational morphology, based on Sproat (1993), Bauer (1983), and Porter (1980). This FSA models a number of derivational facts, such as the well known generalization that any verb ending in *-ize* can be followed by the nominalizing suffix *-ation* (Bauer, 1983; Sproat, 1993). Thus since there is a word *fossilize*, we can predict the word *fossilization* by following states q_0, q_1, and q_2. Similarly, adjectives ending in *-al* or *-able* at q_5 (*equal*, *formal*, *realizable*) can take the suffix *-ity*, or sometimes the suffix *-ness* to state q_6 (*naturalness*, *casualness*). We leave it as an exercise for the reader (Exercise 3.1) to discover some of the individual exceptions to many of these constraints, and also to give examples of some of the various noun and verb classes.

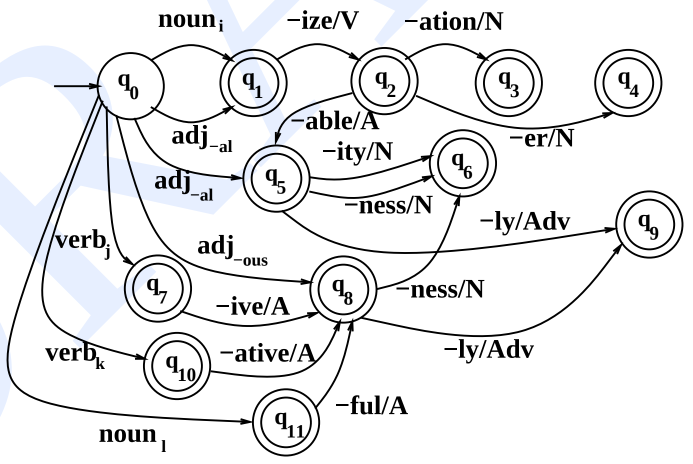

Figure 3.6 An FSA for another fragment of English derivational morphology.

We can now use these FSAs to solve the problem of **morphological recognition**; that is, of determining whether an input string of letters makes up a legitimate English word or not. We do this by taking the morphotactic FSAs, and plugging in each "sub-lexicon" into the FSA. That is, we expand each arc (e.g., the **reg-noun-stem** arc) with all the morphemes that make up the set of **reg-noun-stem**. The resulting FSA can then be defined at the level of the individual letter.

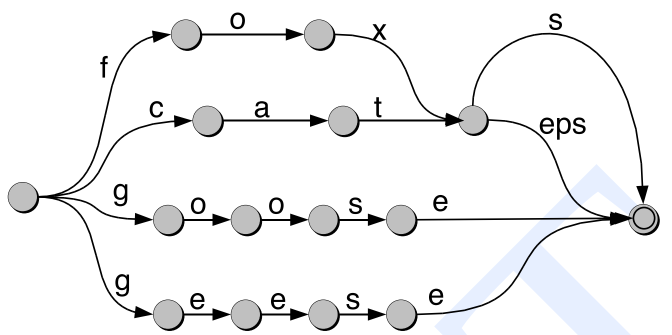

Figure 3.7 Expanded FSA for a few English nouns with their inflection. Note that this automaton will incorrectly accept the input *foxs*. We will see beginning on page 19 how to correctly deal with the inserted *e* in *foxes*.

Fig. 3.7 shows the noun-recognition FSA produced by expanding the Nominal Inflection FSA of Fig. 3.3 with sample regular and irregular nouns for each class. We can use Fig. 3.7 to recognize strings like *aardvarks* by simply starting at the initial state, and comparing the input letter by letter with each word on each outgoing arc, and so on, just as we saw in Ch. 2.

3.4 Finite-State Transducers

We've now seen that FSAs can represent the morphotactic structure of a lexicon, and can be used for word recognition. In this section we introduce the finite-state transducer. The next section will show how transducers can be applied to morphological parsing.

A transducer maps between one representation and another; a **finite-state transducer** or **FST** is a type of finite automaton which maps between two sets of symbols. We can visualize an FST as a two-tape automaton which recognizes or generates *pairs* of strings. Intuitively, we can do this by labeling each arc in the finite-state machine with two symbol strings, one from each tape. Fig. 3.8 shows an example of an FST where each arc is labeled by an input and output string, separated by a colon.

FST

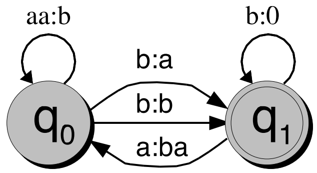

Figure 3.8 A finite-state transducer, modified from Mohri (1997).

The FST thus has a more general function than an FSA; where an FSA defines a formal language by defining a set of strings, an FST defines a *relation* between sets of strings. Another way of looking at an FST is as a machine that reads one string and generates another. Here's a summary of this four-fold way of thinking about transducers:

- **FST as recognizer:** a transducer that takes a pair of strings as input and outputs *accept* if the string-pair is in the string-pair language, and *reject* if it is not.
- **FST as generator:** a machine that outputs pairs of strings of the language. Thus the output is a yes or no, and a pair of output strings.
- **FST as translator:** a machine that reads a string and outputs another string
- **FST as set relater:** a machine that computes relations between sets.

All of these have applications in speech and language processing. For morphological parsing (and for many other NLP applications), we will apply the FST as translator metaphor, taking as input a string of letters and producing as output a string of morphemes.

Let's begin with a formal definition. An FST can be formally defined with 7 parameters:

Q a finite set of N states $q_0, q_1, \ldots, q_{N-1}$

Σ a finite set corresponding to the input alphabet

Δ a finite set corresponding to the output alphabet

$q_0 \in Q$ the start state

$F \subseteq Q$ the set of final states

$\delta(q,w)$ the transition function or transition matrix between states; Given a state $q \in Q$ and a string $w \in \Sigma^*$, $\delta(q,w)$ returns a set of new states $Q' \in Q$. δ is thus a function from $Q \times \Sigma^*$ to 2^Q (because there are 2^Q possible subsets of Q). δ returns a set of states rather than a single state because a given input may be ambiguous in which state it maps to.

$\sigma(q,w)$ the output function giving the set of possible output strings for each state and input. Given a state $q \in Q$ and a string $w \in \Sigma^*$, $\sigma(q,w)$ gives a set of output strings, each a string $o \in \Delta^*$. σ is thus a function from $Q \times \Sigma^*$ to 2^{Δ^*}

REGULAR RELATIONS

Where FSAs are isomorphic to regular languages, FSTs are isomorphic to **regular relations**. Regular relations are sets of pairs of strings, a natural extension of the regular languages, which are sets of strings. Like FSAs and regular languages, FSTs and regular relations are closed under union, although in general they are not closed under difference, complementation and intersection (although some useful subclasses of FSTs *are* closed under these operations; in general FSTs that are not augmented with the ϵ ε are more likely to have such closure properties). Besides union, FSTs have two additional closure properties that turn out to be extremely useful:

INVERSION

- **inversion**: The inversion of a transducer T (T^{-1}) simply switches the input and output labels. Thus if T maps from the input alphabet I to the output alphabet O, T^{-1} maps from O to I.

COMPOSITION

- **composition**: If T_1 is a transducer from I_1 to O_1 and T_2 a transducer from O_1 to O_2, then $T_1 \circ T_2$ maps from I_1 to O_2.

Inversion is useful because it makes it easy to convert a FST-as-parser into an FST-as-generator.

Composition is useful because it allows us to take two transducers that run in series and replace them with one more complex transducer. Composition works as in algebra; applying $T_1 \circ T_2$ to an input sequence S is identical to applying T_1 to S and then T_2 to the result; thus $T_1 \circ T_2(S) = T_2(T_1(S))$.

Fig. 3.9, for example, shows the composition of `[a:b]+` with `[b:c]+` to produce `[a:c]+`.

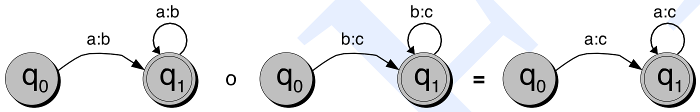

Figure 3.9 The composition of `[a:b]+` with `[b:c]+` to produce `[a:c]+`.

PROJECTION

The **projection** of an FST is the FSA that is produced by extracting only one side of the relation. We can refer to the projection to the left or upper side of the relation as the **upper** or **first** projection and the projection to the lower or right side of the relation as the **lower** or **second** projection.

3.4.1 Sequential Transducers and Determinism

Transducers as we have described them may be nondeterministic, in that a given input may translate to many possible output symbols. Thus using general FSTs requires the kinds of search algorithms discussed in Ch. 2, making FSTs quite slow in the general case. This suggests that it would nice to have an algorithm to convert a nondeterministic FST to a deterministic one. But while every non-deterministic FSA is equivalent to some deterministic FSA, not all finite-state transducers can be determinized.

SEQUENTIAL TRANSDUCERS

Sequential transducers, by contrast, are a subtype of transducers that are deterministic on their input. At any state of a sequential transducer, each given symbol of the input alphabet Σ can label at most one transition out of that state. Fig. 3.10 gives an example of a sequential transducer from Mohri (1997); note that here, unlike the transducer in Fig. 3.8, the transitions out of each state are deterministic based on the state and the input symbol. Sequential transducers can have epsilon symbols in the output string, but not on the input.

Sequential transducers are not necessarily sequential on their output. Mohri's transducer in Fig. 3.10 is not, for example, since two distinct transitions leaving state 0 have the same output (b). Since the inverse of a sequential transducer may thus not be sequential, we always need to specify the direction of the transduction when discussing sequentiality. Formally, the definition of sequential transducers modifies the δ and σ

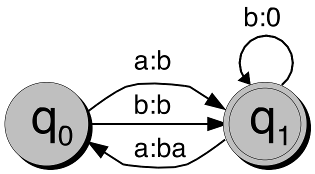

Figure 3.10 A sequential finite-state transducer, from Mohri (1997).

functions slightly; δ becomes a function from $Q \times \Sigma^*$ to Q (rather than to 2^Q), and σ becomes a function from $Q \times \Sigma^*$ to Δ^* (rather than to 2^{Δ^*}).

SUBSEQUENTIAL TRANSDUCER

One generalization of sequential transducers is the **subsequential transducer** (Schützenberger, 1977), which generates an additional output string at the final states, concatenating it onto the output produced so far.

What makes sequential and subsequential transducers important is their efficiency; because they are deterministic on input, they can be processed in time proportional to the number of symbols in the input (they are linear in their input length) rather than proportional to some much larger number which is a function of the number of states. Another advantage of subsequential transducers is that there exist efficient algorithms for their determinization (Mohri, 1997) and minimization (Mohri, 2000), extending the algorithms for determinization and minimization of finite-state automata that we saw in Ch. 2. also an equivalence algorithm.

While both sequential and subsequential transducers are deterministic and efficient, neither of them is able to handle ambiguity, since they transduce each input string to exactly one possible output string. Since ambiguity is a crucial property of natural language, it will be useful to have an extension of subsequential transducers that can deal with ambiguity, but still retain the efficiency and other useful properties of sequential transducers. One such generalization of subsequential transducers is the ***p*-subsequential** transducer. A ***p*-subsequential** transducer allows for $p(p \geq 1)$ final output strings to be associated with each final state (Mohri, 1996). They can thus handle a finite amount of ambiguity, which is useful for many NLP tasks. Fig. 3.11 shows an example of a 2-subsequential FST.

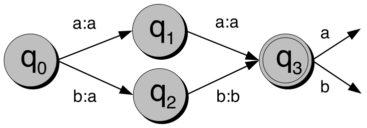

Figure 3.11 A 2-subsequential finite-state transducer, from Mohri (1997).

Mohri (1996, 1997) show a number of tasks whose ambiguity can be limited in this way, including the representation of dictionaries, the compilation of morphological and phonological rules, and local syntactic constraints. For each of these kinds of problems, he and others have shown that they are **p-subsequentializable**, and thus can be determinized and minimized. This class of transducers includes many, although not necessarily all, morphological rules.

3.5 FSTs for Morphological Parsing

Let's now turn to the task of morphological parsing. Given the input *cats*, for instance, we'd like to output *cat +N +Pl*, telling us that *cat* is a plural noun. Given the Spanish input *bebo* ('I drink'), we'd like to output *beber +V +PInd +1P +Sg* telling us that *bebo* is the present indicative first person singular form of the Spanish verb *beber*, 'to drink'.

SURFACE

In the **finite-state morphology** paradigm that we will use, we represent a word as a correspondence between a **lexical level**, which represents a concatenation of morphemes making up a word, and the **surface** level, which represents the concatenation of letters which make up the actual spelling of the word. Fig. 3.12 shows these two levels for (English) *cats* and (Spanish) *bebo*.

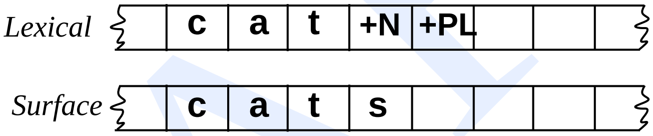

Figure 3.12 Schematic examples of the lexical and surface tapes; the actual transducers will involve intermediate tapes as well.

LEXICAL TAPE

For finite-state morphology it's convenient to view an FST as having two tapes. The **upper** or **lexical tape**, is composed from characters from one alphabet Σ. The **lower** or **surface** tape, is composed of characters from another alphabet Δ. In the **two-level morphology** of Koskenniemi (1983), we allow each arc only to have a single symbol from each alphabet. We can then combine the two symbol alphabets Σ and Δ to create a new alphabet, Σ', which makes the relationship to FSAs quite clear. Σ' is a finite alphabet of complex symbols. Each complex symbol is composed of an input-output pair $i:o$; one symbol i from the input alphabet Σ, and one symbol o from an output alphabet Δ, thus $\Sigma' \subseteq \Sigma \times \Delta$. Σ and Δ may each also include the epsilon symbol ϵ. Thus where an FSA accepts a language stated over a finite alphabet of single symbols, such as the alphabet of our sheep language:

(3.2) $$\Sigma = \{b, a, !\}$$

an FST defined this way accepts a language stated over *pairs* of symbols, as in:

(3.3) $$\Sigma' = \{a:a,\ b:b,\ !:!,\ a:!,\ a:\epsilon,\ \epsilon:!\}$$

FEASIBLE PAIRS

In two-level morphology, the pairs of symbols in Σ' are also called **feasible pairs**. Thus each feasible pair symbol $a:b$ in the transducer alphabet Σ' expresses how the symbol a from one tape is mapped to the symbol b on the other tape. For example $a:\epsilon$ means that an a on the upper tape will correspond to *nothing* on the lower tape. Just as for an FSA, we can write regular expressions in the complex alphabet Σ'. Since it's most common for symbols to map to themselves, in two-level morphology we call pairs like $a : a$ **default pairs**, and just refer to them by the single letter a.

DEFAULT PAIRS

MORPHEME BOUNDARY

#

WORD BOUNDARY

We are now ready to build an FST morphological parser out of our earlier morphotactic FSAs and lexica by adding an extra "lexical" tape and the appropriate morphological features. Fig. 3.13 shows an augmentation of Fig. 3.3 with the nominal morphological features (`+Sg` and `+Pl`) that correspond to each morpheme. The symbol ˆ indicates a **morpheme boundary**, while the symbol # indicates a **word boundary**. The morphological features map to the empty string ϵ or the boundary symbols since there is no segment corresponding to them on the output tape.

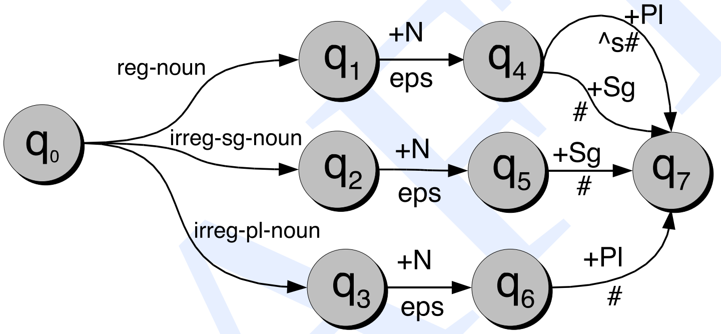

Figure 3.13 A schematic transducer for English nominal number inflection T_{num}. The symbols above each arc represent elements of the morphological parse in the lexical tape; the symbols below each arc represent the surface tape (or the intermediate tape, to be described later), using the morpheme-boundary symbol ˆ and word-boundary marker #. The arcs need to be expanded by individual words in the lexicon.

In order to use Fig. 3.13 as a morphological noun parser, it needs to be expanded with all the individual regular and irregular noun stems, replacing the labels **reg-noun** etc. In order to do this we need to update the lexicon for this transducer, so that irregular plurals like *geese* will parse into the correct stem `goose +N +Pl`. We do this by allowing the lexicon to also have two levels. Since surface *geese* maps to lexical `goose`, the new lexical entry will be "`g:g o:e o:e s:s e:e`". Regular forms are simpler; the two-level entry for *fox* will now be "`f:f o:o x:x`", but by relying on the orthographic convention that `f` stands for `f:f` and so on, we can simply refer to it as `fox` and the form for *geese* as "`g o:e o:e s e`". Thus the lexicon will look only slightly more complex:

reg-noun	**irreg-pl-noun**	**irreg-sg-noun**
fox	g o:e o:e s e	goose
cat	sheep	sheep
aardvark	m o:i u:ϵ s:c e	mouse

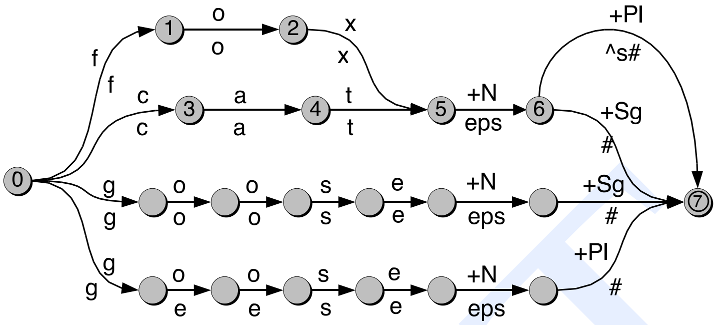

Figure 3.14 A fleshed-out English nominal inflection FST T_{lex}, expanded from T_{num} by replacing the three arcs with individual word stems (only a few sample word stems are shown).

The resulting transducer, shown in Fig. 3.14, will map plural nouns into the stem plus the morphological marker `+Pl`, and singular nouns into the stem plus the morphological marker `+Sg`. Thus a surface *cats* will map to `cat +N +Pl`. This can be viewed in feasible-pair format as follows:

```
c:c a:a t:t +N:ϵ +Pl:ˆs#
```

Since the output symbols include the morpheme and word boundary markers ˆ and #, the lower labels Fig. 3.14 do not correspond exactly to the surface level. Hence we refer to tapes with these morpheme boundary markers in Fig. 3.15 as **intermediate** tapes; the next section will show how the boundary marker is removed.

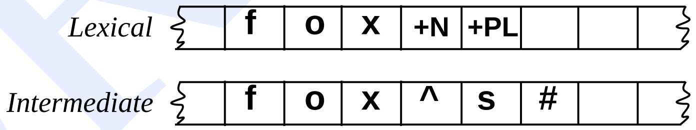

Figure 3.15 A schematic view of the lexical and intermediate tapes.

3.6 Transducers and Orthographic Rules

The method described in the previous section will successfully recognize words like *aardvarks* and *mice*. But just concatenating the morphemes won't work for cases where there is a spelling change; it would incorrectly reject an input like *foxes* and accept an input like *foxs*. We need to deal with the fact that English often requires spelling changes at morpheme boundaries by introducing **spelling rules** (or **orthographic rules**) This section introduces a number of notations for writing such rules

SPELLING RULES

and shows how to implement the rules as transducers. In general, the ability to implement rules as a transducer turns out to be useful throughout speech and language processing. Here's some spelling rules:

Name	Description of Rule	Example
Consonant doubling	1-letter consonant doubled before *-ing*/*-ed*	beg/begging
E deletion	Silent e dropped before *-ing* and *-ed*	make/making
E insertion	e added after *-s*,*-z*,*-x*,*-ch*, *-sh* before *-s*	watch/watches
Y replacement	*-y* changes to *-ie* before *-s*, *-i* before *-ed*	try/tries
K insertion	verbs ending with *vowel* + *-c* add *-k*	panic/panicked

We can think of these spelling changes as taking as input a simple concatenation of morphemes (the "intermediate output" of the lexical transducer in Fig. 3.14) and producing as output a slightly-modified (correctly-spelled) concatenation of morphemes. Fig. 3.16 shows in schematic form the three levels we are talking about: lexical, intermediate, and surface. So for example we could write an E-insertion rule that performs the mapping from the intermediate to surface levels shown in Fig. 3.16. Such a rule might say something like "insert an *e* on the surface tape just when the lexical tape has a morpheme ending in *x* (or *z*, etc) and the next morpheme is *-s*". Here's a formalization of the rule:

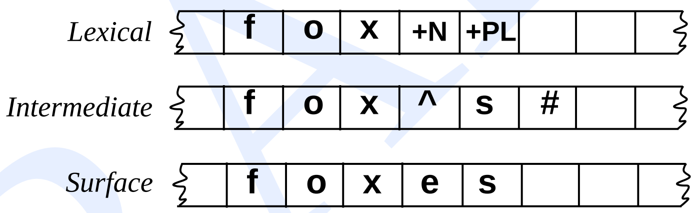

Figure 3.16 An example of the lexical, intermediate, and surface tapes. Between each pair of tapes is a two-level transducer; the lexical transducer of Fig. 3.14 between the lexical and intermediate levels, and the E-insertion spelling rule between the intermediate and surface levels. The E-insertion spelling rule inserts an *e* on the surface tape when the intermediate tape has a morpheme boundary ˆ followed by the morpheme *-s*.

$$\epsilon \rightarrow \mathrm{e} \,/\, \left\{ \begin{array}{c} x \\ s \\ z \end{array} \right\} \hat{}\ \underline{\quad}\ \mathrm{s\#} \tag{3.4}$$

This is the rule notation of Chomsky and Halle (1968); a rule of the form $a \rightarrow b/c\underline{\quad}d$ means "rewrite a as b when it occurs between c and d". Since the symbol ϵ means an empty transition, replacing it means inserting something. Recall that the symbol ˆ indicates a morpheme boundary. These boundaries are deleted by including the symbol ˆ:ϵ in the default pairs for the transducer; thus morpheme boundary markers are deleted on the surface level by default. The # symbol is a special symbol that marks a word boundary. Thus (3.4) means "insert an *e* after a morpheme-final *x*, *s*, or *z*, and before the morpheme *s*". Fig. 3.17 shows an automaton that corresponds to this rule.

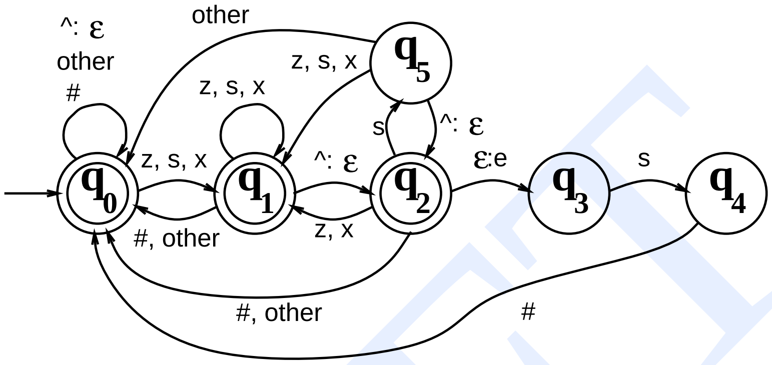

Figure 3.17 The transducer for the E-insertion rule of (3.4), extended from a similar transducer in Antworth (1990). We additionally need to delete the # symbol from the surface string; this can be done either by interpreting the symbol # as the pair #:ϵ, or by postprocessing the output to remove word boundaries.

The idea in building a transducer for a particular rule is to express only the constraints necessary for that rule, allowing any other string of symbols to pass through unchanged. This rule is used to ensure that we can only see the ϵ:*e* pair if we are in the proper context. So state q_0, which models having seen only default pairs unrelated to the rule, is an accepting state, as is q_1, which models having seen a *z*, *s*, or *x*. q_2 models having seen the morpheme boundary after the *z*, *s*, or *x*, and again is an accepting state. State q_3 models having just seen the E-insertion; it is not an accepting state, since the insertion is only allowed if it is followed by the *s* morpheme and then the end-of-word symbol *#*.

The *other* symbol is used in Fig. 3.17 to safely pass through any parts of words that don't play a role in the E-insertion rule. *other* means "any feasible pair that is not in this transducer"; it is thus like a version of @:@ which is context-dependent in a transducer-by-transducer way. So for example when leaving state q_0, we go to q_1 on the *z*, *s*, or *x* symbols, rather than following the *other* arc and staying in q_0. The semantics of *other* depends on what symbols are on other arcs; since # is mentioned on some arcs, it is (by definition) not included in *other*, and thus, for example, is explicitly mentioned on the arc from q_2 to q_0.

A transducer needs to correctly reject a string that applies the rule when it shouldn't. One possible bad string would have the correct environment for the E-insertion, but have no insertion. State q_5 is used to ensure that the *e* is always inserted whenever the environment is appropriate; the transducer reaches q_5 only when it has seen an *s* after an appropriate morpheme boundary. If the machine is in state q_5 and the next symbol is #, the machine rejects the string (because there is no legal transition on # from q_5). Fig. 3.18 shows the transition table for the rule which makes the illegal transitions explicit with the "–" symbol.

State\Input	s:s	x:x	z:z	^:ϵ	ϵ:e	#	other
q_0:	1	1	1	0	-	0	0
q_1:	1	1	1	2	-	0	0
q_2:	5	1	1	0	3	0	0
q_3	4	-	-	-	-	-	-
q_4	-	-	-	-	-	0	-
q_5	1	1	1	2	-	-	0

Figure 3.18 The state-transition table for E-insertion rule of Fig. 3.17, extended from a similar transducer in Antworth (1990).

The next section will show a trace of this E-insertion transducer running on a sample input string.

3.7 Combining FST Lexicon and Rules

We are now ready to combine our lexicon and rule transducers for parsing and generating. Fig. 3.19 shows the architecture of a two-level morphology system, whether used for parsing or generating. The lexicon transducer maps between the lexical level, with its stems and morphological features, and an intermediate level that represents a simple concatenation of morphemes. Then a host of transducers, each representing a single spelling rule constraint, all run in parallel so as to map between this intermediate level and the surface level. Putting all the spelling rules in parallel is a design choice; we could also have chosen to run all the spelling rules in series (as a long cascade), if we slightly changed each rule.

CASCADE

The architecture in Fig. 3.19 is a two-level **cascade** of transducers. Cascading two automata means running them in series with the output of the first feeding the input to the second. Cascades can be of arbitrary depth, and each level might be built out of many individual transducers. The cascade in Fig. 3.19 has two transducers in series: the transducer mapping from the lexical to the intermediate levels, and the collection of parallel transducers mapping from the intermediate to the surface level. The cascade can be run top-down to generate a string, or bottom-up to parse it; Fig. 3.20 shows a trace of the system *accepting* the mapping from *fox^s* to *foxes*.

The power of finite-state transducers is that the exact same cascade with the same state sequences is used when the machine is generating the surface tape from the lexical tape, or when it is parsing the lexical tape from the surface tape. For example, for generation, imagine leaving the Intermediate and Surface tapes blank. Now if we run the lexicon transducer, given `fox +N +PL`, it will produce *fox^s#* on the Intermediate tape via the same states that it accepted the Lexical and Intermediate tapes in our earlier example. If we then allow all possible orthographic transducers to run in parallel, we will produce the same surface tape.

AMBIGUITY

Parsing can be slightly more complicated than generation, because of the problem of **ambiguity**. For example, *foxes* can also be a verb (albeit a rare one, meaning “to baffle or confuse”), and hence the lexical parse for *foxes* could be `fox +V +3Sg` as

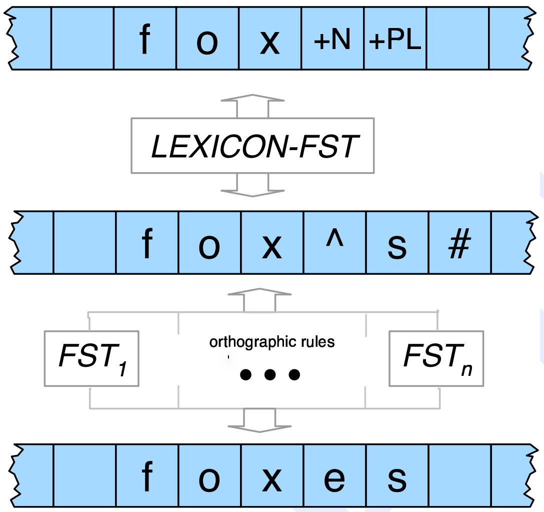

Figure 3.19 Generating or parsing with FST lexicon and rules

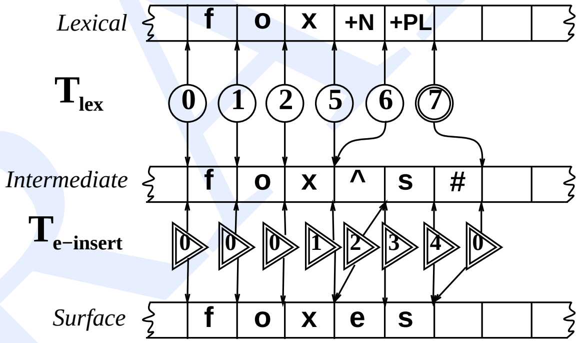

Figure 3.20 Accepting *foxes*: The lexicon transducer T_{lex} from Fig. 3.14 cascaded with the E-insertion transducer in Fig. 3.17.

well as `fox +N +PL`. How are we to know which one is the proper parse? In fact, for ambiguous cases of this sort, the transducer is not capable of deciding. **Disambiguating** will require some external evidence such as the surrounding words. Thus *foxes* is likely to be a noun in the sequence *I saw two foxes yesterday*, but a verb in the sequence *That trickster foxes me every time!*. We will discuss such disambiguation algorithms in Ch. 5 and Ch. 19. Barring such external evidence, the best our transducer can do is just enumerate the possible choices; so we can transduce *foxˆs#* into both `fox +V +3SG` and `fox +N +PL`.

DISAMBIGUATING

There is a kind of ambiguity that we need to handle: local ambiguity that occurs during the process of parsing. For example, imagine parsing the input verb *assess*. After seeing *ass*, our E-insertion transducer may propose that the *e* that follows is inserted by the spelling rule (for example, as far as the transducer is concerned, we might have been parsing the word *asses*). It is not until we don't see the # after *asses*, but rather run into another *s*, that we realize we have gone down an incorrect path.

Because of this non-determinism, FST-parsing algorithms need to incorporate some sort of search algorithm. Exercise 3.7 asks the reader to modify the algorithm for non-deterministic FSA recognition in Fig. **??** in Ch. 2 to do FST parsing.

Note that many possible spurious segmentations of the input, such as parsing *assess* as ˆaˆsˆsesˆs will be ruled out since no entry in the lexicon will match this string.

INTERSECTION

Running a cascade, particularly one with many levels, can be unwieldy. Luckily, we've already seen how to compose a cascade of transducers in series into a single more complex transducer. Transducers in parallel can be combined by **automaton intersection**. The automaton intersection algorithm just takes the Cartesian product of the states, i.e., for each state q_i in machine 1 and state q_j in machine 2, we create a new state q_{ij}. Then for any input symbol a, if machine 1 would transition to state q_n and machine 2 would transition to state q_m, we transition to state q_{nm}. Fig. 3.21 sketches how this intersection (∧) and composition (◦) process might be carried out.

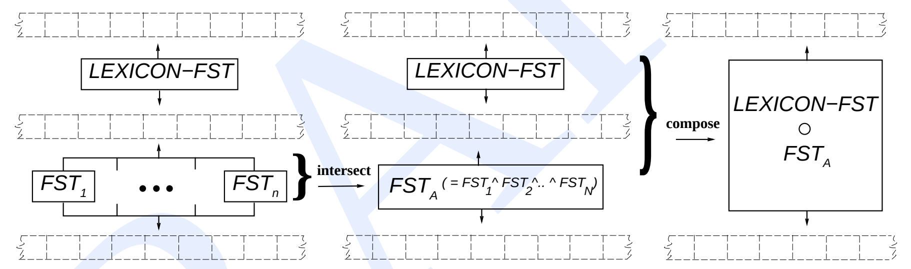

Figure 3.21 Intersection and composition of transducers.

Since there are a number of rule→FST compilers, it is almost never necessary in practice to write an FST by hand. Kaplan and Kay (1994) give the mathematics that define the mapping from rules to two-level relations, and Antworth (1990) gives details of the algorithms for rule compilation. Mohri (1997) gives algorithms for transducer minimization and determinization.

3.8 Lexicon-Free FSTs: The Porter Stemmer

KEYWORDS

While building a transducer from a lexicon plus rules is the standard algorithm for morphological parsing, there are simpler algorithms that don't require the large on-line lexicon demanded by this algorithm. These are used especially in Information Retrieval (IR) tasks like web search (Ch. 21), in which a query such as a Boolean combination of relevant **keywords** or phrases, e.g., (*marsupial OR kangaroo OR koala*) returns documents that have these words in them. Since a document with the word *marsupials* might not match the keyword *marsupial*, some IR systems first run a stemmer on the query and document words, Morphological information in IR is thus only used to determine that two words have the same stem; the suffixes are thrown away.

STEMMING

One of the most widely used such **stemming** algorithms is the simple and efficient Porter (1980) algorithm, which is based on a series of simple cascaded rewrite rules. Since cascaded rewrite rules are just the sort of thing that could be easily implemented as an FST, we think of the Porter algorithm as a lexicon-free FST stemmer (this idea will be developed further in the exercises (Exercise 3.6). The algorithm contains rules like these:

ATIONAL → ATE (e.g., relational → relate)
ING → ϵ if stem contains vowel (e.g., motoring → motor)

See Porter (1980) or Martin Porter's official homepage for the Porter stemmer for more details.

Krovetz (1993) showed that stemming tends to somewhat improve the performance of information retrieval, especially with smaller documents (the larger the document, the higher the chance the keyword will occur in the exact form used in the query). Nonetheless, not all IR engines use stemming, partly because of stemmer errors such as these noted by Krovetz:

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
generalization	generic	matrices	matrix
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

3.9 Word and Sentence Tokenization

We have focused so far in this chapter on a problem of segmentation: how words can be segmented into morphemes. We turn now to a brief discussion of the very related problem of segmenting running text into words and sentences. This task is called **tokenization**.

TOKENIZATION

Word tokenization may seem very simple in a language like English that separates words via a special 'space' character. As we will see below, not every language does this (Chinese, Japanese, and Thai, for example, do not). But a closer examination will make it clear that whitespace is not sufficient by itself. Consider the following sentences from a Wall Street Journal and New York Times article, respectively:

```
Mr.  Sherwood said reaction to Sea Containers' proposal
has been "very positive." In New York Stock Exchange composite
trading yesterday, Sea Containers closed a t $62.625, up
62.5 cents.
''I said, 'what're you?  Crazy?'  ''  said Sadowsky.  ''I
can't afford to do that.''
```

Segmenting purely on white-space would produce words like these:

```
cents.      said,      positive."      Crazy?
```

We could address these errors by treating punctuation, in addition to whitespace, as a word boundary. But punctuation often occurs word internally, in examples like *m.p.h*, *Ph.D.*, *AT&T*, *cap'n*, *01/02/06*, and *google.com*. Similarly, assuming that we want *62.5* to be a word, we'll need to avoid segmenting every period, since that will segment this into 62 and 5. Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: *555,500.50*. Languages differ on punctuation styles for numbers; many continental European languages like Spanish, French, and German, by contrast, uses a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas: *555 500,50*.

Another useful task a tokenizer can do for us is to expand clitic contractions that are marked by apostrophes, for example converting *what're* above to the two tokens *what are*, and *we're* to *we are*. This task is complicated by the fact that apostrophes are quite ambiguous, since they are also used as genitive markers (as in *the book's over* or in *Containers'* above) or as quotative markers (as in *'what're you? Crazy?'* above). Such contractions occur in other alphabetic languages, including articles and pronouns in French (*j'ai*, *l'homme*). While these contractions tend to be clitics, not all clitics are marked this way with contraction. In general, then, segmenting and expanding clitics can be done as part of the process of morphological parsing presented earlier in the chapter.

Depending on the application, tokenization algorithms may also tokenize multiword expressions like *New York* or *rock 'n' roll*, which requires a multiword expression dictionary of some sort. This makes tokenization intimately tied up with the task of detecting names, dates, and organizations, which is called *named entity detection* and will be discussed in Ch. 17.

SENTENCE SEGMENTATION

In addition to word segmentation, **sentence segmentation** is a crucial first step in text processing. Segmenting a text into sentences is generally based on punctuation. This is because certain kinds of punctuation (periods, question marks, exclamation points) tend to mark sentence boundaries. Question marks and exclamation points are relatively unambiguous markers of sentence boundaries. Periods, on the other hand, are more ambiguous. The period character '.' is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization tend to be addressed jointly.

In general, sentence tokenization methods work by building a binary classifier (based on a sequence of rules, or on machine learning) which decides if a period is part of the word or is a sentence boundary marker. In making this decision, it helps to know if the period is attached to a commonly used abbreviation; thus an abbreviation dictionary is useful.

State-of-the-art methods for sentence tokenization are based on machine learning and will be introduced in later chapters. But a useful first step can still be taken via a sequence of regular expressions. We introduce here the first part; a word tokenization algorithm. Fig. 3.22 gives a simple Perl word tokenization algorithm based on Grefenstette (1999). The algorithm is quite minimal, designed mainly to clarify many of the segmentation issues we discussed in previous paragraphs.

The algorithm consists of a sequence of regular expression substitution rules. The first rule separates unambiguous punctuation like question marks and parentheses. The next rule segments commas unless they are inside numbers. We then disambiguate apostrophes and pull off word-final clitics. Finally, we deal with periods, using a (toy) abbreviation dictionary and some heuristics for detecting other abbreviations.

```
#!/usr/bin/perl

$letternumber = "[A-Za-z0-9]";
$notletter = "[ˆA-Za-z0-9]";
$alwayssep = "[\\?!()\";/\\|‘]";
$clitic = "(’|:|-|’S|’D|’M|’LL|’RE|’VE|N’T|’s|’d|’m|’ll|’re|’ve|n’t)";

$abbr{"Co."} = 1; $abbr{"Dr."} = 1; $abbr{"Jan."} = 1; $abbr{"Feb."} = 1;

while (<>){

    # put whitespace around unambiguous separators
    s/$alwayssep/ $& /g;

    # put whitespace around commas that aren’t inside numbers
    s/([ˆ0-9]),/$1 , /g;
    s/,([ˆ0-9])/ , $1/g;

    # distinguish singlequotes from apostrophes by
    # segmenting off single quotes not preceded by letter
    s/ˆ’/$& /g;
    s/($notletter)’/$1 ’/g;

    # segment off unambiguous word-final clitics and punctuation
    s/$clitic$/ $&/g;
    s/$clitic($notletter)/ $1 $2/g;

   # now deal with periods.  For each possible word
   @possiblewords=split(/\s+/,$_);
   foreach $word (@possiblewords) {
      # if it ends in a period,
      if (($word =˜ /$letternumber\./)
             && !($abbr{$word})  # and isn’t on the abbreviation list
                # and isn’t a sequence of letters and periods (U.S.)
                # and doesn’t resemble an abbreviation (no vowels: Inc.)
             && !($word =˜ /ˆ([A-Za-z]\.([A-Za-z]\.)+|[A-Z][bcdfghj-nptvxz]+\.)$/)) {
          # then segment off the period
          $word =˜ s/\.$/ \./;
      }
      # expand clitics
      $word =˜s/’ve/have/;
      $word =˜s/’m/am/;
      print $word," ";
   }
 print "\n";
}
```

Figure 3.22 A sample English tokenization script, adapted from Grefenstette (1999) and Palmer (2000). A real script would have a longer abbreviation dictionary.

The fact that a simple tokenizer can be build with such simple regular expression patterns suggest that tokenizers like the one in Fig. 3.22 can be easily implemented in FSTs. This is indeed the case, and (Karttunen et al., 1996) and (Beesley and Karttunen, 2003) give descriptions of such FST-based tokenizers.

3.9.1 Segmentation in Chinese

We mentioned above that some languages, including Chinese, Japanese, and Thai, do not use spaces to mark potential word-boundaries. Alternative segmentation methods are used for these languages.

MAXIMUM MATCHING

In Chinese, for example, words are composed of characters known as *hanzi*. Each character generally represents a single morpheme and is pronounceable as a single syllable. Words on average are about 2.4 characters long. A simple algorithm that does remarkably well for segmenting Chinese, and is often used as a baseline comparison for more advanced methods, is a version of greedy search called **maximum matching** or sometimes **maxmatch**. The algorithm requires a dictionary (wordlist) of the language.

The maximum matching algorithm starts by pointing at the beginning of a string. It chooses the longest word in the dictionary that matches the input at the current position. The pointer is then advanced past each character in that word. If no word matches, the pointer is instead advanced one character (creating a one-character word). The algorithm is then iteratively applied again starting from the new pointer position. To help visualize this algorithm, Palmer (2000) gives an English analogy, which approximates the Chinese situation by removing the spaces from the English sentence *the table down there* to produce *thetabledownthere*. The maximum match algorithm (given a long English dictionary) would first match the word *theta* in the input, since that is the longest sequence of letters that matches a dictionary word. Starting from the end of *theta*, the longest matching dictionary word is *bled*, followed by *own* and then *there*, producing the incorrect sequence *theta bled own there*.

The algorithm seems to work better in Chinese (with such short words) than in languages like English with long words, as our failed example shows. Even in Chinese, however, maxmatch has a number of weakness, particularly with **unknown words** (words not in the dictionary) or **unknown genres** (genres which differ a lot from the assumptions made by the dictionary builder).

There is an annual competition (technically called a **bakeoff**) for Chinese segmentation algorithms. These most successful modern algorithms for Chinese word segmentation are based on machine learning from hand-segmented training sets. We will return to these algorithms after we introduce probabilistic methods in Ch. 5.

3.10 Detecting and Correcting Spelling Errors

> 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 section we introduce the problem of detecting and correcting spelling errors. Since the standard algorithm for spelling error correction is probabilistic, we will continue our spell-checking discussion later in Ch. 5 after we define the probabilistic noisy channel model.

OCR

The detection and correction of spelling errors is an integral part of modern word-processors and search engines, and is also important in correcting errors in **optical character recognition** (**OCR**), the automatic recognition of machine or hand-printed characters, and **on-line handwriting recognition**, the recognition of human printed or cursive handwriting as the user is writing.

Following Kukich (1992), we can distinguish 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 accidentally 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

Detecting non-word errors is generally done by marking any word that is not found in a dictionary. For example, the misspelling *graffe* 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 the rare words *wont* or *veery* are also common misspelling of *won't* and *very*. In practice, Damerau and Mays (1989) found that while some misspellings were hidden by real words in a larger dictionary, the larger dictionary proved more help than harm by avoiding marking rare words as errors. This is especially true with probabilistic spell-correction algorithms that can use word frequency as a factor. Thus modern spell-checking systems tend to be based on large dictionaries.

The finite-state morphological parsers described throughout this chapter provide a technology for implementing such large dictionaries. By giving a morphological parser for a word, an FST parser is inherently a word recognizer. Indeed, an FST morphological parser can be turned into an even more efficient FSA word recognizer by using the **projection** operation to extract the lower-side language graph. Such FST dictionaries also have the advantage of representing productive morphology like the English *-s* and *-ed* inflections. This is important for dealing with new legitimate combinations of stems and inflection . For example, a new stem can be easily added to the dictionary, and then all the inflected forms are easily recognized. This makes FST dictionaries especially powerful for spell-checking in morphologically rich languages where a single stem can have tens or hundreds of possible surface forms.[5]

FST dictionaries can thus help with non-word error detection. But how about error correction? Algorithms for isolated-word error correction operate by finding words which are the likely source of the errorful form. For example, correcting the spelling error *graffe* requires searching through all possible words like *giraffe*, *graf*, *craft*, *grail*, etc, to pick the most likely source. To choose among these potential sources we need a **distance metric** between the source and the surface error. Intuitively, *giraffe* is a more likely source than *grail* for *graffe*, because *giraffe* is closer in spelling to *graffe* than *grail* is to *graffe*. The most powerful way to capture this similarity intuition requires the use of probability theory and will be discussed in Ch. 4. The algorithm underlying this solution, however, is the non-probabilistic **minimum edit distance** algorithm that we introduce in the next section.

3.11 Minimum Edit Distance

DISTANCE

Deciding which of two words is closer to some third word in spelling is a special case of the general problem of **string distance**. The distance between two strings is a measure of how alike two strings are to each other.

MINIMUM EDIT DISTANCE

ALIGNMENT

Many important algorithms for finding string distance rely on 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 the words *intention* and *execution* is five operations, shown in Fig. 3.23 as an **alignment** between the two strings. Beneath the aligned strings are a series of symbols expressing an **operation list** for converting the top string into the bottom string; d for deletion, s for substitution, i for insertion.

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).[6] Thus the Levenshtein

[5] Early spelling error detectors for English, by contrast, 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*.

[6] We assume that the substitution of a letter for itself, e.g. substitution *t* for *t*, has zero cost.

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Figure 3.23 Representing the minimum edit distance between two strings as an alignment. The final row gives the operation list for converting the top string into the bottom string; d for deletion, s for substitution, i for insertion.

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.

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; besides minimum edit distance, these include the **Viterbi** and **forward** algorithms (Ch. 6), and the **CYK** and **Earley** algorithm (Ch. 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* shown in Fig. 3.24.

```
                      i n t e n t i o n
          delete i →
                      n t e n t i o n
   substitute n by e →
                      e t e n t i o n
   substitute t by x →
                      e x e n t i o n
            insert u →
                      e x e n u t i o n
   substitute n by c →
                      e x e c u t i o n
```

Figure 3.24 Operation list transforming *intention* to *execution* (after Kruskal 1983)

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 computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$distance[i,j] = \min \begin{cases} distance[i-1,j] + \text{ins-cost}(target_{i-1}) \\ distance[i-1,j-1] + \text{subst-cost}(source_{j-1}, target_{i-1}) \\ distance[i,j-1] + \text{del-cost}(source_{j-1})) \end{cases} \tag{3.5}$$

The algorithm itself is summarized in Fig. 3.25, while Fig. 3.26 shows the results of applying the algorithm to the distance between *intention* and *execution* assuming the version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost(·) = del-cost(·) = 1), and substitutions have a cost of 2 (except substitution of identical letters has zero cost).

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

  n ← LENGTH(target)
  m ← LENGTH(source)
  Create a distance matrix distance[n+1,m+1]
  Initialize the zeroth row and column to be the distance from the empty string
    distance[0,0] = 0
    for each column i from 1 to n do
       distance[i,0] ← distance[i-1,0] + ins-cost(target[i])
    for each row j from 1 to m do
       distance[0,j] ← distance[0,j-1] + del-cost(source[j])
  for each column i from 1 to n do
    for each row j from 1 to m do
      distance[i,j] ← MIN( distance[i−1,j] + ins-cost(target_{i−1}),
                           distance[i−1,j−1] + subst-cost(source_{j−1}, target_{i−1}),
                           distance[i,j−1] + del-cost(source_{j−1}))
  return distance[n,m]
```

Figure 3.25 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g. $\forall x, \text{ins-cost}(x) = 1$), or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e. $\text{subst-cost}(x,x) = 0$).

Knowing the minimum edit distance is useful for algorithms like finding potential spelling error corrections. But the edit distance algorithm is important in another

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

Figure 3.26 Computation of minimum edit distance between *intention* and *execution* via algorithm of Fig. 3.25, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. In italics are the initial values representing the distance from the empty string.

way; with a small change, it can also provide the minimum cost **alignment** between two strings. Aligning two strings is useful throughout speech and language processing. In speech recognition, minimum edit distance alignment is used to compute word error rate in speech recognition (Ch. 9). Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched up to each other.

In order to extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Fig. 3.27 shows this path with the boldfaced cell. Each boldfaced cell represents an alignment of a pair of letters in the two strings. If two boldfaced cells occur in the same row, there will be an insertion in going from the source to the target; two boldfaced cells in the same column indicates a deletion.

Fig. 3.27 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that were extended from in entering the current cell. We've shown a schematic of these backpointers in Fig. 3.27, after a similar diagram in Gusfield (1997). Some cells have multiple backpointers, because the minimum extension could have come from multiple previous cells. In the second step, we perform a **backtrace**. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 3.12 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

BACKTRACE

There are various publicly available packages to compute edit distance, including UNIX diff, and the NIST `sclite` program (NIST, 2005); Minimum edit distance can also be augmented in various ways. The Viterbi algorithm, for example, is an extension of minimum edit distance which uses probabilistic definitions of the operations. In this case instead of computing the "minimum edit distance" between two strings, we are

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

Figure 3.27 When entering a value in each cell, we mark which of the 3 neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) via a **backtrace**, starting at the **8** in the upper right corner and following the arrows. The sequence of boldfaced distances represents one possible minimum cost alignment between the two strings.

interested in the "maximum probability **alignment**" of one string with another. The Viterbi algorithm is crucial in probabilistic tasks like speech recognition and part-of-speech tagging.

3.12 Human Morphological Processing

In this section we briefly survey psycholinguistic studies on how multi-morphemic words are represented in the minds of speakers of English. For example, consider the word *walk* and its inflected forms *walks*, and *walked*. Are all three in the human lexicon? Or merely *walk* along with *-ed* and *-s*? How about the word *happy* and its derived forms *happily* and *happiness*? We can imagine two ends of a theoretical spectrum of representations. The **full listing** hypothesis proposes that all words of a language are listed in the mental lexicon without any internal morphological structure. On this view, morphological structure is simply an epiphenomenon, and *walk*, *walks*, *walked*, *happy*, and *happily* are all separately listed in the lexicon. This hypothesis is certainly untenable for morphologically complex languages like Turkish. The **minimum redundancy** hypothesis suggests that only the constituent morphemes are represented in the lexicon, and when processing *walks*, (whether for reading, listening, or talking) we must access both morphemes (*walk* and *-s*) and combine them.

FULL LISTING

MINIMUM REDUNDANCY

Some of the earliest evidence that the human lexicon represents at least some morphological structure comes from **speech errors**, also called **slips of the tongue**. In conversational speech, speakers often mix up the order of the words or sounds:

if you break it it'll drop

In slips of the tongue collected by Fromkin and Ratner (1998) and Garrett (1975), inflectional and derivational affixes can appear separately from their stems. The ability of these affixes to be produced separately from their stem suggests that the mental lexicon contains some representation of morphological structure.

it's not only us who have screw looses (for "screws loose")
words of rule formation (for "rules of word formation")
easy enoughly (for "easily enough")

PRIMED

More recent experimental evidence suggests that neither the full listing nor the minimum redundancy hypotheses may be completely true. Instead, it's possible that some but not all morphological relationships are mentally represented. Stanners et al. (1979), for example, found that some derived forms (*happiness*, *happily*) are stored separately from their stem (*happy*), but that regularly inflected forms (*pouring*) are not distinct in the lexicon from their stems (*pour*). They did this by using a repetition priming experiment. In short, repetition priming takes advantage of the fact that a word is recognized faster if it has been seen before (if it is **primed**). They found that *lifting* primed *lift*, and *burned* primed *burn*, but for example *selective* didn't prime *select*. In a more recent study, Marslen-Wilson et al. (1994) found that *spoken* derived words can prime their stems, but only if the meaning of the derived form is closely related to the stem. For example *government* primes *govern*, but *department* does not prime *depart*. Marslen-Wilson et al. (1994) represent a model compatible with their own findings as follows:

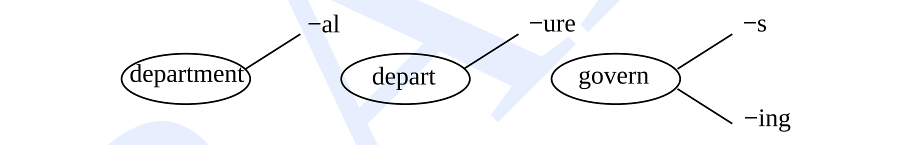

Figure 3.28 Marslen-Wilson et al. (1994) result: Derived words are linked to their stems only if semantically related.

In summary, these results suggest that morphology does play a role in the human lexicon, especially productive morphology like inflection.

3.13 SUMMARY

This chapter introduced **morphology**, the arena of language processing dealing with the subparts of words, and the **finite-state transducer**, the computational device that is important for morphology but will also play a role in many other tasks in later chapters. We also introduced **stemming**, **word and sentence tokenization**, and **spelling error detection**.

Here's a summary of the main points we covered about these ideas:

- **Morphological parsing** is the process of finding the constituent **morphemes** in a word (e.g., `cat +N +PL` for *cats*).
- English mainly uses **prefixes** and **suffixes** to express **inflectional** and **derivational** morphology.

- English **inflectional** morphology is relatively simple and includes person and number agreement (*-s*) and tense markings (*-ed* and *-ing*).
- English **derivational** morphology is more complex and includes suffixes like *-ation*, *-ness*, *-able* as well as prefixes like *co-* and *re-*.
- Many constraints on the English **morphotactics** (allowable morpheme sequences) can be represented by finite automata.
- **Finite-state transducers** are an extension of finite-state automata that can generate output symbols.
- Important operations for FSTs include **composition**, **projection**, and **intersection**.
- **Finite-state morphology** and **two-level morphology** are applications of finite-state transducers to morphological representation and parsing.
- **Spelling rules** can be implemented as transducers.
- There are automatic transducer-compilers that can produce a transducer for any simple rewrite rule.
- The lexicon and spelling rules can be combined by **composing** and **intersecting** various transducers.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It is not as accurate as a transducer model that includes a lexicon, but may be preferable for applications like **information retrieval** in which exact morphological structure is not needed.
- **Word tokenization** can be done by simple regular expressions substitutions or by transducers.
- **Spelling error detection** is normally done by finding words which are not in a dictionary; an FST dictionary can be useful for this.

Bibliographical and Historical Notes

Despite the close mathematical similarity of finite-state transducers to finite-state automata, the two models grew out of somewhat different traditions. Ch. 2 described how the finite automaton grew out of Turing's (1936) model of algorithmic computation, and McCulloch and Pitts finite-state-like models of the neuron. The influence of the Turing machine on the transducer was somewhat more indirect. Huffman (1954) proposed what was essentially a state-transition table to model the behavior of sequential circuits, based on the work of Shannon (1938) on an algebraic model of relay circuits. Based on Turing and Shannon's work, and unaware of Huffman's work, Moore (1956) introduced the term **finite automaton** for a machine with a finite number of states with an alphabet of input symbols and an alphabet of output symbols. Mealy (1955) extended and synthesized the work of Moore and Huffman.

The finite automata in Moore's original paper, and the extension by Mealy differed in an important way. In a Mealy machine, the input/output symbols are associated with the transitions between states. In a Moore machine, the input/output symbols are associated with the state. The two types of transducers are equivalent; any Moore machine can be converted into an equivalent Mealy machine and vice versa. Further early work on finite-state transducers, sequential transducers, and so on, was conducted by ? (?), Schützenberger (1977), ? (?).

Many early programs for morphological parsing used an **affix-stripping** approach to parsing. For example Packard's (1973) parser for ancient Greek iteratively stripped prefixes and suffixes off the input word, making note of them, and then looked up the remainder in a lexicon. It returned any root that was compatible with the stripped-off affixes. This approach is equivalent to the **bottom-up** method of parsing that we will discuss in Ch. 12.

AMPLE (A Morphological Parser for Linguistic Exploration) (Weber and Mann, 1981; Weber et al., 1988; Hankamer and Black, 1991) is another early bottom-up morphological parser. It contains a lexicon with all possible surface variants of each morpheme (these are called **allomorphs**), together with constraints on their occurrence (for example in English the *-es* allomorph of the plural morpheme can only occur after s, x, z, sh, or ch). The system finds every possible sequence of morphemes which match the input and then filters out all the sequences which have failing constraints.

An alternative approach to morphological parsing is called *generate-and-test* or *analysis-by-synthesis* approach. Hankamer's (1986) keCi is a morphological parser for Turkish which is guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, and applies every possible phonological rule to it, checking each result against the input. If one of the outputs succeeds, the program then follows the finite-state morphotactics to the next morpheme and tries to continue matching the input.

The idea of modeling spelling rules as finite-state transducers is really based on Johnson's (1972) early idea that phonological rules (to be discussed in Ch. 7) have finite-state properties. Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Ronald Kaplan and Martin Kay, first in an unpublished talk (Kaplan and Kay, 1981) and then finally in print (Kaplan and Kay, 1994) (see page **??** for a discussion of multiple independent discoveries). Kaplan and Kay's work was followed up and most fully worked out by Koskenniemi (1983), who described finite-state morphological rules for Finnish. Karttunen (1983) built a program called KIMMO based on Koskenniemi's models. Antworth (1990) gives many details of two-level morphology and its application to English. Besides Koskenniemi's work on Finnish and that of Antworth (1990) on English, two-level or other finite-state models of morphology have been worked out for many languages, such as Turkish (Oflazer, 1993) and Arabic (Beesley, 1996). Barton, Jr. et al. (1987) bring up some computational complexity problems with two-level models, which are responded to by Koskenniemi and Church (1988). Finally, readers with further interest in finite-state morphology should turn to the definitive text, (Beesley and Karttunen, 2003).

A number of practical implementations of sentence segmentation were available by the 1990s. Summaries of sentence segmentation history and various algorithms can be found in Palmer (2000), Grefenstette (1999), and Mikheev (2003). Word segmentation has been studied especially in Japanese and Chinese. While the max-match algorithm we describe is very commonly used as a baseline, or when a simple but accurate algorithm is required, more recent algorithms rely on stochastic and machine learning algorithms; see for example such algorithms as Sproat et al. (1996), Xue and Shen (2003), and ? (?).

Gusfield (1997) is an excellent book covering everything you could want to know about string distance, minimum edit distance, and related areas.

Students interested in further details of the fundamental mathematics of automata theory should see Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1988). Roche and Schabes (1997) is the definitive mathematical introduction to finite-state transducers for language applications, and together with Mohri (1997) and Mohri (2000) give many useful algorithms such as those for transducer minimization and determinization.

Sproat (1993) gives a broad general introduction to computational morphology.

Exercises

3.1 Give examples of each of the noun and verb classes in Fig. 3.6, and find some exceptions to the rules.

3.2 Extend the transducer in Fig. 3.17 to deal with `sh` and `ch`.

3.3 Write a transducer(s) for the K insertion spelling rule in English.

3.4 Write a transducer(s) for the consonant doubling spelling rule in English.

3.5 The Soundex algorithm (Odell and Russell, 1922; Knuth, 1973) is a method commonly used in libraries and older Census records for representing people's names. It has the advantage that versions of the names that are slightly misspelled or otherwise modified (common, for example, in hand-written census records) will still have the same representation as correctly-spelled names. (e.g., Jurafsky, Jarofsky, Jarovsky, and Jarovski all map to J612).

a. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y

b. Replace the remaining letters with the following numbers:

b, f, p, v → 1
c, g, j, k, q, s, x, z → 2
d, t → 3
l → 4
m, n → 5
r → 6

c. Replace any sequences of identical numbers , only if they derive from two or more letters that were *adjacent* in the original name, with a single number (i.e., 666 → 6).

d. Convert to the form `Letter Digit Digit Digit` by dropping digits past the third (if necessary) or padding with trailing zeros (if necessary).

The exercise: write a FST to implement the Soundex algorithm.

3.6 Implement one of the steps of the Porter Stemmer as a transducer.

3.7 Write the algorithm for parsing a finite-state transducer, using the pseudo-code introduced in Chapter 2. You should do this by modifying the algorithm ND-RECOGNIZE in Fig. **??** in Chapter 2.

3.8 Write a program that takes a word and, using an on-line dictionary, computes possible anagrams of the word, each of which is a legal word.

3.9 In Fig. 3.17, why is there a *z, s, x* arc from q_5 to q_1?

3.10 Computing minimum edit distances by hand, figure out whether *drive* is closer to *brief* or to *divers*, and what the edit distance is. You may use any version of *distance* that you like.

3.11 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

3.12 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.

Antworth, E. L. (1990). *PC-KIMMO: A Two-level Processor for Morphological Analysis*. Summer Institute of Linguistics, Dallas, TX.

Barton, Jr., G. E., Berwick, R. C., and Ristad, E. S. (1987). *Computational Complexity and Natural Language*. MIT Press, Cambridge, MA.

Bauer, L. (1983). *English word-formation*. Cambridge University Press, Cambridge.

Beesley, K. R. (1996). Arabic finite-state morphological analysis and generation. In *COLING-96*, Copenhagen, pp. 89–94.

Beesley, K. R. and Karttunen, L. (2003). *Finite-State Morphology*. CSLI Publications, Stanford University.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. Harper and Row, New York.

Damerau, F. J. and Mays, E. (1989). An examination of undetected typing errors. *Information Processing and Management*, *25*(6), 659–664.

Fromkin, V. and Ratner, N. B. (1998). Speech production. In Gleason, J. B. and Ratner, N. B. (Eds.), *Psycholinguistics*. Harcourt Brace, Fort Worth, TX.

Garrett, M. F. (1975). The analysis of sentence production. In Bower, G. H. (Ed.), *The Psychology of Learning and Motivation*, Vol. 9. Academic, New York.

Grefenstette, G. (1999). Tokenization. In van Halteren, H. (Ed.), *Syntactic Wordclass Tagging*. Kluwer Academic Publishers, Dordrecht.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Hankamer, J. (1986). Finite state morphology and left to right phonology. In *Proceedings of the Fifth West Coast Conference on Formal Linguistics*, pp. 29–34.

Hankamer, J. and Black, H. A. (1991). Current approaches to computational morphology. Unpublished manuscript.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Huffman, D. A. (1954). The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, *3*, 161–191. Continued in Volume 4.

Johnson, C. D. (1972). *Formal Aspects of Phonological Description*. Mouton, The Hague. Monographs on Linguistic Analysis No. 3.

Kaplan, R. M. and Kay, M. (1981). Phonological rules and finite-state transducers. Paper presented at the Annual meeting of the Linguistics Society of America. New York.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, *20*(3), 331–378.

Karttunen, L., Chanod, J., Grefenstette, G., and Schiller, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, *2*(4), 305–238.

Karttunen, L. (1983). KIMMO: A general morphological processor. In *Texas Linguistics Forum 22*, pp. 165–186.

Knuth, D. E. (1973). *Sorting and Searching: The Art of Computer Programming Volume 3*. Addison-Wesley, Reading, MA.

Koskenniemi, K. (1983). Two-level morphology: A general computational model of word-form recognition and production. Tech. rep. Publication No. 11, Department of General Linguistics, University of Helsinki.

Koskenniemi, K. and Church, K. W. (1988). Complexity, two-level morphology, and Finnish. In *COLING-88*, Budapest, pp. 335–339.

Krovetz, R. (1993). Viewing morphology as an inference process. In *SIGIR-93*, pp. 191–202. ACM.

Kruskal, J. B. (1983). An overview of sequence comparison. In Sankoff, D. and Kruskal, J. B. (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 1–44. Addison-Wesley, Reading, MA.

Kukich, K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys*, *24*(4), 377–439.

Lerner, A. J. (1978). *The Street Where I Live*. Da Capo Press, New York.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, *10*(8), 707–710. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

Lewis, H. and Papadimitriou, C. (1988). *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ. Second edition.

Marslen-Wilson, W., Tyler, L. K., Waksler, R., and Older, L. (1994). Morphology and meaning in the English mental lexicon. *Psychological Review*, *101*(1), 3–33.

McCawley, J. D. (1978). Where you can shove infixes. In Bell, A. and Hooper, J. B. (Eds.), *Syllables and Segments*, pp. 213–221. North-Holland, Amsterdam.

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *Bell System Technical Journal*, *34*(5), 1045–1079.

Mikheev, A. (2003). Text segmentation. In Mitkov, R. (Ed.), *Oxford Handbook of Computational Linguistics*. Oxford University Press, Oxford.

Mohri, M. (1996). On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, *2*(1), 61–80.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, *23*(2), 269–312.

Mohri, M. (2000). Minimization algorithms for sequential transducers. *Theoretical Computer Science*, *234*, 177–201.

Moore, E. F. (1956). Gedanken-experiments on sequential machines. In Shannon, C. and McCarthy, J. (Eds.), *Automata Studies*, pp. 129–153. Princeton University Press, Princeton, NJ.

NIST (2005). Speech recognition scoring toolkit (sctk) version 2.1. Available at http://www.nist.gov/speech/tools/.

Odell, M. K. and Russell, R. C. (1918/1922). U.S. Patents 1261167 (1918), 1435663 (1922)†. Cited in Knuth (1973).

Oflazer, K. (1993). Two-level description of Turkish morphology. In *Proceedings, Sixth Conference of the European Chapter of the ACL*.

Orgun, O. (1995). A declarative theory of phonology-morphology interleaving. Unpublished manuscript, U. of California-Berkeley, Department of Linguistics.

Packard, D. W. (1973). Computer-assisted morphological analysis of ancient Greek. In Zampolli, A. and Calzolari, N. (Eds.), *Computational and Mathematical Linguistics: Proceedings of the International Conference on Computational Linguistics*, Pisa, pp. 343–355. Leo S. Olschki.

Palmer, D. D. (2000). Tokenisation and sentence segmentation. In Dale, R., Somers, H. L., and Moisl, H. (Eds.), *Handbook of Natural Language Processing*. Marcel Dekker, Inc., New York, NY, USA.

Peterson, J. L. (1986). A note on undetected typing errors. *Communications of the ACM*, *29*(7), 633–637.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, *14*(3), 130–127.

Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*. Longman, London.

Roche, E. and Schabes, Y. (1997). Introduction. In Roche, E. and Schabes, Y. (Eds.), *Finite-State Language Processing*, pp. 1–65. MIT Press, Cambridge, MA.

Schützenberger, M. P. (1977). Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, *4*, 47–57.

Seuss, D. (1960). *One Fish Two Fish Red Fish Blue Fish*. Random House, New York.

Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, *57*, 713–723.

Sproat, R. (1993). *Morphology and Computation*. MIT Press, Cambridge.

Sproat, R., Shih, C., Gale, W. A., and Chang, N. (1996). A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics*, *22*(3), 377–404.

Stanners, R. F., Neiser, J., Hernon, W. P., and Hall, R. (1979). Memory representation for morphologically related words. *Journal of Verbal Learning and Verbal Behavior*, *18*, 399–412.

Tseng, H., Jurafsky, D., and Manning, C. (2005). Morphological features help pos tagging of unknown words across language varieties. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*.

Veblen, T. (1899). *Theory of the Leisure Class*. Macmillan Company, New York.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the Association for Computing Machinery*, *21*, 168–173.

Weber, D. J., Black, H. A., and McConnel, S. R. (1988). AMPLE: A tool for exploring morphology. Tech. rep. Occasional Publications in Academic Computing No. 12, Summer Institute of Linguistics, Dallas.

Weber, D. J. and Mann, W. C. (1981). Prospects for computer-assisted dialect adaptation. *American Journal of Computational Linguistics*, *7*, 165–177. Abridged from Summer Instititute of Linguistics *Notes on Linguistics* Special Publication 1, 1979.

Xue, N. and Shen, L. (2003). Chinese word segmentation as lmr tagging. In *Proceedings of the 2nd SIGHAN Workshop on Chinese Language Processing*, Sapporo, Japan.