Chapter 2
Regular Expressions and Automata

User$_1$: *Men are all alike.*
ELIZA$_1$: *IN WHAT WAY*
User$_2$: *They're always bugging us about something or other.*
ELIZA$_2$: *CAN YOU THINK OF A SPECIFIC EXAMPLE*
User$_3$: *Well, my boyfriend made me come here.*
ELIZA$_3$: *YOUR BOYFRIEND MADE YOU COME HERE*
User$_4$: *He says I'm depressed much of the time.*
ELIZA$_4$: *I AM SORRY TO HEAR YOU ARE DEPRESSED*

Weizenbaum (1966)

Imagine that you have become a passionate fan of woodchucks and have recently learned that *groundhog* and *woodchuck* are different names for the same animal. Since you are writing a term paper on woodchucks, you now need to search through your paper for every occurrence of the term *woodchuck* and replace *woodchucks* with *woodchucks (groundhogs)*. But you also need to replace singular *woodchuck* with *woodchuck (groundhog)*. Instead of having to do this search twice, you would prefer to perform a single command for something like *woodchuck with an optional final s*. Or perhaps you might want to search for all the prices in some document; you might want to see all strings that look like *$199* or *$25* or *$24.99* in order to automatically extract a table of prices. In this chapter we introduce the **regular expression**, the standard notation for characterizing text sequences. The regular expression is used for specifying text strings in all sorts of text processing and information extraction applications.

After we have defined regular expressions, we show how they can be implemented with the **finite-state automaton**. The finite-state automaton is not only the mathematical device used to implement regular expressions but also one of the most significant tools of computational linguistics. Variations of automata such as finite-state transducers, hidden Markov models, and *N*-gram grammars are important components of applications that we introduce in later chapters, including speech recognition and synthesis, machine translation, spell-checking, and information extraction.

2.1 Regular Expressions

SIR ANDREW: *Her C's, her U's and her T's: why that?*

Shakespeare, *Twelfth Night*

Regular expression

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. The regular ex-

pression languages used for searching texts in Unix tools like grep and Emacs, in Perl, Python, Ruby, Java, and .NET, and also in Microsoft Word are very similar, and many RE features exist in Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics.

Strings

A regular expression (first developed by Kleene (1956) but see the Historical Notes section for more details) is a formula in a special language that specifies simple classes of **strings**. A string is a sequence of symbols; for most text-based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol ␣.

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus, they can specify search strings as well as define a language in a formal way. We begin by talking about regular expressions as a way of specifying searches in texts and proceed to other uses. Section 2.3 shows that the use of just three regular expression operators is sufficient to characterize strings, but we use the more convenient and commonly used regular expression syntax of the Perl language throughout this section. Since common text-processing programs agree on most of the syntax of regular expressions, most of what we say extends to all UNIX and Microsoft Word regular expressions.

Corpus

Regular expression search requires a **pattern** that we want to search for and a **corpus** of texts to search through. A regular expression search function will search through the corpus, returning all texts that match the pattern. In an information retrieval (IR) system such as a Web search engine, the texts might be entire documents or Web pages. In a word processor, the texts might be individual words or lines of a document. In the rest of this chapter, we use this last paradigm. Thus, when we give a search pattern, we assume that the search engine returns the *line of the document* returned. This is what the Unix `grep` command does. We underline the exact part of the pattern that matches the regular expression. A search can be designed to return all matches to a regular expression or only the first match. We show only the first match.

2.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. For example, to search for *woodchuck*, we type `/woodchuck/`. So the regular expression `/Buttercup/` matches any string containing the substring *Buttercup*, for example, the line *I'm called little Buttercup* (recall that we are assuming a search application that returns entire lines).

From here on we put slashes around each regular expression to make it clear what is a regular expression and what is a pattern. We use the slash since this is the notation used by Perl, but the slashes are *not* part of the regular expressions.

The search string can consist of a single character (like `/!/`) or a sequence of characters (like `/urgl/`). The *first* instance of each match to the regular expression is underlined below (although a given application might choose to return more than just the first instance):

RE	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/Claire␣says,/	" "Dagmar, my gift please," Claire says,"
/DOROTHY/	"SURRENDER DOROTHY"
/!/	"You've left the burglar behind again!" said Nori

Regular expressions are **case sensitive**; lower case /s/ is distinct from upper case /S/ (/s/ matches a lower case *s* but not an uppercase *S*). This means that the pattern /woodchucks/ will not match the string *Woodchucks*. We can solve this problem with the use of the square braces [and]. The string of characters inside the braces specify a **disjunction** of characters to match. For example, Fig. 2.1 shows that the pattern /[wW]/ matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.1 The use of the brackets [ ] to specify a disjunction of characters.

The regular expression /[1234567890]/ specified any single digit. While such classes of characters as digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify

/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/

to mean "any capital letter"). In these cases the brackets can be used with the dash (-) to specify any one character in a **range**. The pattern /[2-5]/ specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern /[b-g]/ specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.2.

Range

RE	Match	Example Patterns Matched
/[A-Z]/	an upper case letter	"we should call it 'Drenched Blossoms' "
/[a-z]/	a lower case letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.2 The use of the brackets [ ] plus the dash - to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ^. If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated. For example, the pattern /[^a]/ matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.3 shows some examples.

The use of square braces solves our capitalization problem for *woodchucks*. But we still haven't answered our original question; how do we specify both *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question mark /?/, which means "the preceding character or nothing", as shown in Fig. 2.4.

RE	Match (single characters)	Example Patterns Matched
`[^A-Z]`	not an upper case letter	"Oyfn pripetchik"
`[^Ss]`	neither 'S' nor 's'	"I have no exquisite reason for't"
`[^\.]`	not a period	"our resident Djinn"
`[e^]`	either 'e' or '^'	"look up ^ now"
`a^b`	the pattern 'a^b'	"look up a^ b now"

Figure 2.3 Uses of the caret ^ for negation or just to mean ^. We discuss below the need to escape the period by a backslash.

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Figure 2.4 The question mark ? marks optionality of the previous expression.

We can think of the question mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want. So far we haven't needed to specify that we want more than one of something. But sometimes we need regular expressions that allow repetitions. For example, consider the language of (certain) sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
...

This language consists of strings with a *b*, followed by at least two *a*'s, followed by an exclamation point. The set of operators that allows us to say things like "some number of *a*'s" are based on the asterisk or `*`, commonly called the **Kleene** * (pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa*, but it will also match *Off Minor* since the string *Off Minor* has zero *a*'s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means "zero or more *a*'s or *b*'s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*.

*Kleene **

We now know enough to specify part of our regular expression for prices: multiple digits. Recall that the regular expression for an individual digit was `/[0-9]/`. So the regular expression for an integer (a string of digits) is `/[0-9][0-9]*/`. (Why isn't it just `/[0-9]*/`?)

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character. This is the **Kleene +**, which means "one or more of the previous character". Thus, the expression `/[0-9]+/` is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

Kleene +

One very important special character is the period (`/./`), a **wildcard** expression that matches any single character (*except* a carriage return), as shown in Fig. 2.5.

RE	Match	Example Patterns
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.5 The use of the period `.` to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example, suppose we want to find any line in which a particular word, for example, *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

Anchors

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret `^` and the dollar sign `$`. The caret `^` matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. Thus, the caret `^` has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow Perl to know which function a given caret is supposed to have?) The dollar sign `$` matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog*. (We have to use the backslash here since we want the `.` to mean "period" and not the wildcard.)

There are also two other anchors: `\b` matches a word boundary, and `\B` matches a non-boundary. Thus, `/\bthe\b/` matches the word *the* but not the word *other*. More technically, Perl defines a word as any sequence of digits, underscores, or letters; this is based on the definition of "words" in programming languages like Perl or C. For example, `/\b99\b/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

2.1.2 Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string *cat* or the string *dog*. Since we can't use the square brackets to search for "cat or dog" (why not?), we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

Disjunction

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene`*`. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

Precedence

The parenthesis operator (is also useful when we are using counters like the Kleene*. Unlike the | operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any number of columns; instead, it will match a single column followed by any number of spaces! The star here applies only to the space ␣ that precedes it, not to the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

Operator precedence

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence.

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *theny*.

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

Greedy

2.1.3 A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
/the/
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
/[tT]he/
```

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

```
/\b[tT]he\b/
```

Suppose we wanted to do this without the use of `/\b/`. We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to

find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

```
/[^a-zA-Z][tT]he[^a-zA-Z]/
```

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded instances of *the*, implies that there must be some single (although non-alphabetic) character before the *the*. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character, and the same at the end of the line:

```
/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/
```

False positive
False negative

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in implementing speech and language processing systems. Reducing the error rate for an application thus involves two antagonistic efforts:

- Increasing **accuracy** (minimizing false positives)
- Increasing **coverage** (minimizing false negatives)

2.1.4 A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any PC with more than 6 GHz and 256 GB of disk space for less than $1000". To do this kind of retrieval, we first need to be able to look for expressions like *6 GHz* or *256 GB* or *Dell* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits. Note that Perl is smart enough to realize that `$` here doesn't mean end-of-line; how might it know that?

```
/$[0-9]+/
```

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional and to make sure we're at a word boundary:

```
/\b$[0-9]+(\.[0-9][0-9])?\b/
```

How about specifications for processor speed (in megahertz = MHz or gigahertz = GHz)? Here's a pattern for that:

```
/\b[0-9]+␣*(MHz|[Mm]egahertz|GHz|[Gg]igahertz)\b/
```

Note that we use `/␣*/` to mean "zero or more spaces" since there might always be extra spaces lying around. Dealing with disk space or memory size (in GB = gigabytes), we need to allow for optional fractions again (*5.5 GB*). Note the use of `?` for making the final `s` optional:

```
/\b[0-9]+(\.[0-9]+)?␣*(GB|[Gg]igabytes?)\b/
```

Finally, we might want some simple patterns to specify operating systems:

```
/\b(Windows␣*(Vista|XP)?)\b/
/\b(Mac|Macintosh|Apple|OS␣X)\b/
```

2.1.5 Advanced Operators

There are also some useful advanced regular expression operators. Figure 2.6 shows some aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene + we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression `/{3}/` means "exactly 3 occurrences of the previous character or expression". So `/a\.{24}z/` will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

RE	Expansion	Match	Examples
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.6 Aliases for common sets of characters.

A range of numbers can also be specified. So `/{n,m}/` specifies from *n* to *m* occurrences of the previous char or expression, and `/{n,}/` means at least *n* occurrences of the previous expression. REs for counting are summarized in Fig. 2.7.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression

Figure 2.7 Regular expression operators for counting.

Newline

Finally, certain special characters are referred to by special notation based on the backslash (\) (see Fig. 2.8). The most common of these are the **newline** character \n and the **tab** character \t. To refer to characters that are special themselves (like `.`, `*`, `[`, and \), precede them with a backslash, (i.e., `/\./`, `/\*/`, `/\[/`, and `/\\/`).

RE	Match	Example Patterns Matched
`\*`	an asterisk "*"	"K*A*P*L*A*N"
`\.`	a period "."	"Dr. Livingston, I presume"
`\?`	a question mark	"Why don't they come and lend a hand?"
`\n`	a newline	
`\t`	a tab	

Figure 2.8 Some characters that need to be backslashed.

2.1.6 Regular Expression Substitution, Memory, and ELIZA

Substitution

An important use of regular expressions is in **substitutions**. For example, the Perl substitution operator `s/regexp1/pattern/` allows a string characterized by a regular expression to be replaced by another string:

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, for example, changing *the 35 boxes* to *the <35> boxes*. We'd like a way to refer to the integer we've found so that we can easily add the brackets. To do this, we put parentheses `(` and `)` around the first pattern and use the **number** operator `\1` in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator `\1`, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the `\1` will be replaced by whatever string matched the first item in parentheses. So this will match *The bigger they were, the bigger they will be* but not *The bigger they were, the faster they will be*.

The number operator can be used with other numbers. If you match two different sets of parenthesis, `\2` means whatever matched the *second* set. For example,

```
/the (.*)er they (.*), the \1er we \2/
```

will match *The faster they ran, the faster we ran* but not *The faster they ran, the faster we ate*. These numbered memories are called **registers** (e.g., register 1, register 2, register 3). This memory feature is not part of every regular expression language and is often considered an "extended" feature of regular expressions.

Register

Substitutions using memory are useful in implementing simple natural-language-understanding programs like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

$User_1$: Men are all alike.
$ELIZA_1$: IN WHAT WAY
$User_2$: They're always bugging us about something or other.
$ELIZA_2$: CAN YOU THINK OF A SPECIFIC EXAMPLE
$User_3$: Well, my boyfriend made me come here.
$ELIZA_3$: YOUR BOYFRIEND MADE YOU COME HERE
$User_4$: He says I'm depressed much of the time.
$ELIZA_4$: I AM SORRY TO HEAR YOU ARE DEPRESSED

ELIZA works by having a cascade of regular expression substitutions that each match some part of the input lines and changes them. The first substitutions change all instances of *my* to *YOUR*, and *I'm* to *YOU ARE*, and so on. The next set of substitutions matches and replaces other patterns in the input. Here are some examples:

```
s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions can apply to a given input, substitutions are assigned a rank and applied in order. Creating patterns is the topic of Exercise 2.2.

2.2 Finite-State Automata

Finite-state automaton
FSA

The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a **finite-state automaton** (**FSA**). Finite-state automata are the theoretical foundation of a good deal of the computational work we describe in this book. Any regular expression can be implemented as a finite-state automaton (except regular expressions that use the memory feature; more on this later). Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind of formal language called a **regular language**. Both regular expressions and finite-state automata can be used to describe regular languages. A third equivalent method of characterizing the regular languages, the **regular grammar**, is introduced in Chapter 16. The relation among these theoretical constructions is sketched in Fig. 2.9.

Regular language

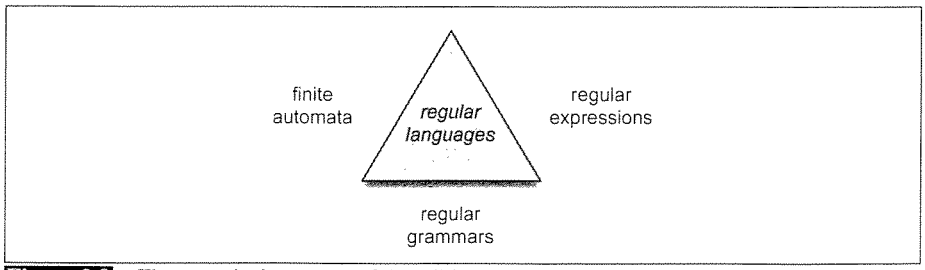

Figure 2.9 Three equivalent ways of describing regular languages.

This section begins by introducing finite-state automata for some of the regular expressions from the last section and then suggests how the mapping from regular expressions to automata proceeds in general. Although we begin with their use for implementing regular expressions, FSAs have a wide variety of other uses that we explore in this chapter and the next.

2.2.1 Use of an FSA to Recognize Sheeptalk

> *After a while, with the parrot's help, the Doctor got to learn the language of the animals so well that he could talk to them himself and understand everything they said.*
>
> Hugh Lofting, *The Story of Doctor Dolittle*

Let's begin with the "sheep language" we discussed previously. Recall that we defined the sheep language as any string from the following (infinite) set:

baa!
baaa!
baaaa!
baaaaa!
...

Automaton

The regular expression for this kind of "sheeptalk" is `/baa+!/`. Figure 2.10 shows an **automaton** for modeling this regular expression. The automaton (i.e., machine, also called **finite automaton**, **finite-state automaton**, or **FSA**) recognizes a set of strings, in this case the strings characterizing sheep talk in the same way that a regular expression does. We represent the automaton as a directed graph: a finite set of vertices (also called nodes), together with a set of directed links between pairs of vertices called arcs. We'll represent vertices with circles and arcs with arrows. The automaton has five **states**, which are represented by nodes in the graph. State 0 is the **start state**. In our examples, state 0 will generally be the start state; to mark another state as the start state, we can add an incoming arrow to the start state. State 4 is the **final state** or **accepting state**, which we represent by the double circle. It also has five **transitions**, which we represent by arcs in the graph.

State

Start state

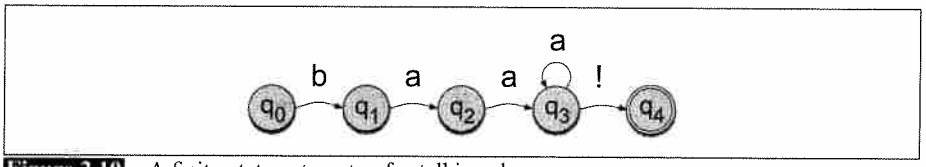

Figure 2.10 A finite-state automaton for talking sheep.

The FSA can be used for recognizing (we also say **accepting**) strings in the following way. First, think of the input as being written on a long tape broken up into cells, with one symbol written in each cell of the tape, as in Fig. 2.11.

The machine starts in the start state (q_0) and iterates the following process: Check the next letter of the input. If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input. If we are in the accepting state (q_4) when we run out of input, the machine has

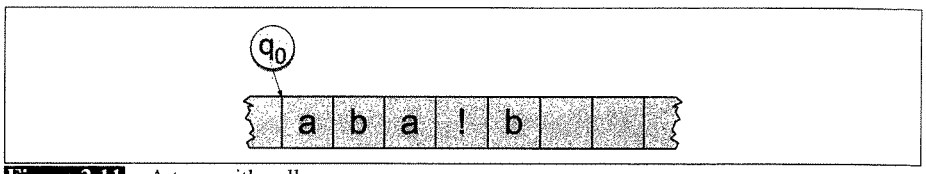

Figure 2.11 A tape with cells.

successfully recognized an instance of sheeptalk. If the machine never gets to the final state, either because it runs out of input or it gets some input that doesn't match an arc (as in Fig. 2.11), or if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.

Rejecting

State-transition table

We can also represent an automaton with a **state-transition table**. As in the graph notation, the state-transition table represents the start state, the accepting states, and what transitions leave each state with which symbols. On the right is the state-transition table for the FSA of Fig. 2.10. We've marked state 4 with a colon to indicate that it's a final state (you can have as many final states as you want), and the $\emptyset$ indicates an illegal or missing transition. We can read the first row as "if we're in state 0 and we see the input **b** we must go to state 1. If we're in state 0 and we see the input **a** or **!**, we fail".

	Input		
State	**b**	**a**	**!**
0	1	$\emptyset$	$\emptyset$
1	$\emptyset$	2	$\emptyset$
2	$\emptyset$	3	$\emptyset$
3	$\emptyset$	3	4
4:	$\emptyset$	$\emptyset$	$\emptyset$

More formally, a finite automaton is defined by the following five parameters:

$Q = q_0 q_1 q_2 \ldots q_{N-1}$	a finite set of N **states**
Σ	a finite **input alphabet** of symbols
q_0	the **start state**
F	the set of **final states**, $F \subseteq Q$
$\delta(q,i)$	the **transition function** or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q;

For the sheeptalk automaton in Fig. 2.10, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, !\}$, $F = \{q_4\}$, and $\delta(q,i)$ is defined by the transition table above.

Figure 2.12 presents an algorithm for recognizing a string using a state-transition table. The algorithm is called D-RECOGNIZE for "deterministic recognizer". A **deterministic** algorithm is one that has no choice points; the algorithm always knows what to do for any input. The next section introduces non-deterministic automata that must make decisions about which states to move to.

Deterministic

D-RECOGNIZE takes as input a tape and an automaton. It returns *accept* if the string it is pointing to on the tape is accepted by the automaton, and *reject* otherwise. Note that since D-RECOGNIZE assumes it is already pointing at the string to be checked, its task is only a subpart of the general problem that we often use regular expressions for, finding a string in a corpus. (The general problem is left as Exercise 2.9 for the reader.)

D-RECOGNIZE begins by setting the variable *index* to the beginning of the tape, and *current-state* to the machine's initial state. D-RECOGNIZE then enters a loop that drives

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
   if End of input has been reached then
    if current-state is an accept state then
      return accept
    else
      return reject
   elsif transition-table[current-state,tape[index]] is empty then
     return reject
   else
     current-state ← transition-table[current-state,tape[index]]
     index ← index + 1
  end
```

Figure 2.12 An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

the algorithm. It first checks whether it has reached the end of its input. If so, it either accepts the input (if the current state is an accept state) or rejects the input (if not).

If there is input left on the tape, D-RECOGNIZE looks at the transition table to decide which state to move to. The variable *current-state* indicates which row of the table to consult, and the current symbol on the tape indicates which column of the table to consult. The resulting transition-table cell is used to update the variable *current-state* and *index* is incremented to move forward on the tape. If the transition-table cell is empty, then the machine has nowhere to go and must reject the input.

Figure 2.13 traces the execution of this algorithm on the sheep language FSA given the sample input string *baaa!*.

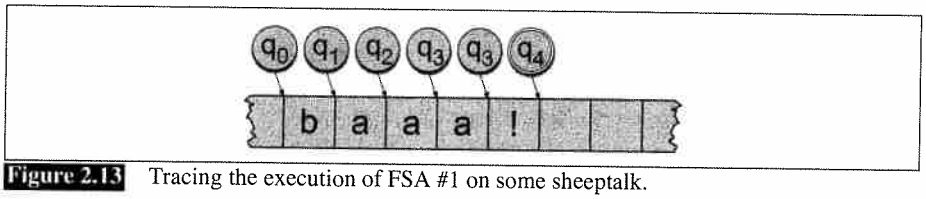

Figure 2.13 Tracing the execution of FSA #1 on some sheeptalk.

Before examining the beginning of the tape, the machine is in state q_0. Finding a *b* on input tape, it changes to state q_1 as indicated by the contents of *transition-table*[q_0,*b*] on page 28. It then finds an *a* and switches to state q_2, another *a* puts it in state q_3, a third *a* leaves it in state q_3, where it reads the *!* and switches to state q_4. Since there is no more input, the end of input condition at the beginning of the loop is satisfied for the first time and the machine halts in q_4. State q_4 is an accepting state, so the machine has accepted the string *baaa!* as a sentence in the sheep language.

The algorithm fails whenever there is no legal transition for a given combination of state and input. The input *abc* will fail to be recognized since there is no legal transition out of state q_0 on the input a (i.e., this entry of the transition table on page 28 has a $\emptyset_1$. Even if the automaton had allowed an initial *a*, it would have certainly failed on *c* since *c* isn't even in the sheeptalk alphabet! We can think of these "empty" elements in the table as if they all pointed at one "empty" state, which we might call the **fail state** or **sink state**. In a sense then, we could view any machine with empty transitions *as if* we had augmented it with a fail state and had drawn in all the extra arcs so that we always had somewhere to go from any state on any possible input. Just for completeness, Fig. 2.14 shows the FSA from Fig. 2.10 with the fail state q_F filled in.

Fail state

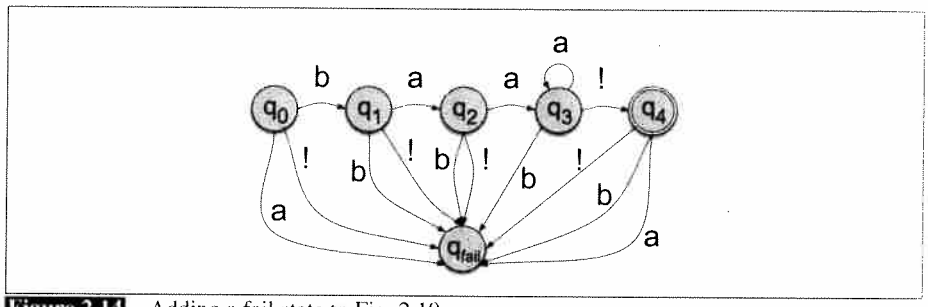

Figure 2.14 Adding a fail state to Fig. 2.10.

2.2.2 Formal Languages

We can use the same graph in Fig. 2.10 as an automaton for GENERATING sheeptalk. If we do, we would say that the automaton starts at state q_0 and crosses arcs to new states, printing out the symbols that label each arc it follows. When the automaton gets to the final state it stops. Notice that at state q_3, the automaton has to choose between printing a ! and going to state q_4, or printing an **a** and returning to state q_3. Let's say for now that we don't care how the machine makes this decision; maybe it flips a coin. For now, we don't care which exact string of sheeptalk we generate, as long as it's a string captured by the regular expression for sheeptalk above.

> **Formal Language:** A model that can both generate and recognize all and only the strings of a formal language acts as a *definition* of the formal language.

Formal language
Alphabet

A **formal language** is a set of strings, each string composed of symbols from a finite symbol set called an **alphabet** (the same alphabet used above for defining an automaton). The alphabet for the sheep language is the set $\Sigma = \{a, b, !\}$. Given a model m (such as a particular FSA), we can use $L(m)$ to mean "the formal language characterized by m". So the formal language defined by our sheeptalk automaton m in Fig. 2.10 (and the transition table on page 28) is the infinite set

$$L(m) = \{baa!, baaa!, baaaa!, baaaaa!, baaaaaa!, \ldots\} \tag{2.1}$$

The usefulness of an automaton for defining a language is that it can express an infinite set (such as the one above) in a closed form. Formal languages are not the same

Natural language

as **natural languages**, which are the languages that real people speak. In fact, a formal language may bear no resemblance at all to a real language (e.g., a formal language can be used to model the different states of a soda machine). But we often use a formal language to model part of a natural language, such as parts of the phonology, morphology, or syntax. The term **generative grammar** is sometimes used in linguistics to mean a grammar of a formal language; the origin of the term is this use of an automaton to define a language by generating all possible strings.

2.2.3 Another Example

In the previous examples, our formal alphabet consisted of letters; but we can also have a higher-level alphabet consisting of words. In this way we can write finite-state automata that model facts about word combinations. For example, suppose we wanted to build an FSA that modeled the subpart of English dealing with amounts of money. Such a formal language would model the subset of English consisting of phrases like *ten cents*, *three dollars*, *one dollar thirty-five cents*, and so on.

We might break this down by first building just the automaton to account for the numbers from 1 to 99, since we'll need them to deal with cents. Figure 2.15 shows this.

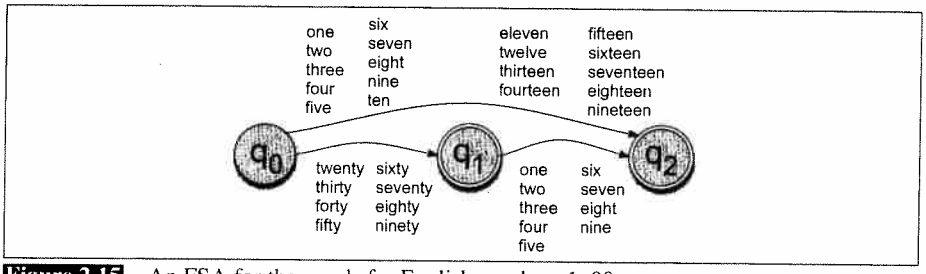

Figure 2.15 An FSA for the words for English numbers 1–99.

We could now add *cents* and *dollars* to our automaton. Figure 2.16 shows a simple version of this, where we just made two copies of the automaton in Fig. 2.15 with minor changes, and appended the words *cents* and *dollars*.

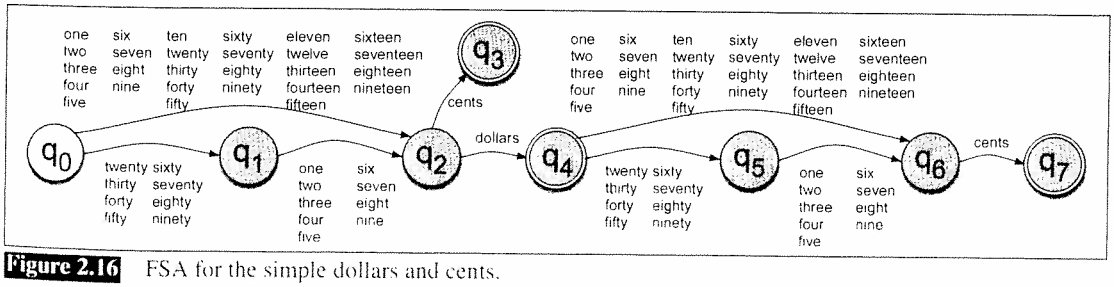

Figure 2.16 FSA for the simple dollars and cents.

We would now need to add in the grammar for different amounts of dollars; including higher numbers like *hundred*, *thousand*. We'd also need to make sure that the

nouns like *cents* and *dollars* are singular when appropriate (*one cent*, *one dollar*), and plural when appropriate (*ten cents*, *two dollars*). This is left as an exercise for the reader (Exercise 2.3). We can think of the FSAs in Fig. 2.15 and Fig. 2.16 as simple grammars of parts of English. We return to grammar-building in Part III of this book, particularly in Chapter 12.

2.2.4 Non-Deterministic FSAs

Let's extend our discussion now to another class of FSAs: **non-deterministic FSA**s (or **NFSA**s). Consider the sheeptalk automaton in Fig. 2.17, which is much like our first automaton in Fig. 2.10.

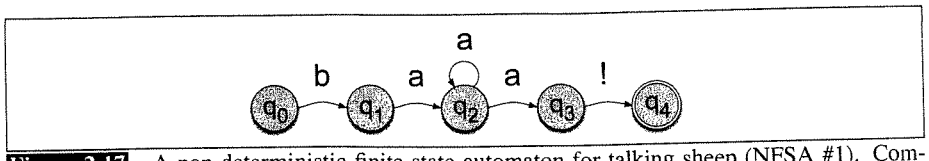

Figure 2.17 A non-deterministic finite-state automaton for talking sheep (NFSA #1). Compare with the deterministic automaton in Fig. 2.10.

The only difference between this automaton and the previous one is that here in Fig. 2.17 the self-loop is on state 2 instead of state 3. Consider using this network as an automaton for recognizing sheeptalk. When we get to state 2, if we see an **a** we don't know whether to remain in state 2 or go on to state 3. Automata with decision points like this are called **non-deterministic FSA**s (or **NFSA**s). Recall by contrast that Fig. 2.10 specified a **deterministic** automaton, that is one whose behavior during recognition is fully *determined* by the state it is in and the symbol it is looking at. A deterministic automaton can be referred to as a **DFSA**. That is not true for the machine in Fig. 2.17 (NFSA #1).

Non-deterministic NFSA

DFSA

Another common type of non-determinism is one caused by arcs that have no symbols on them (called **ϵ-transitions**). The automaton in Fig. 2.18 defines exactly the same language as the last one and our first one, but it does it with an ϵ-transition.

ϵ-transition

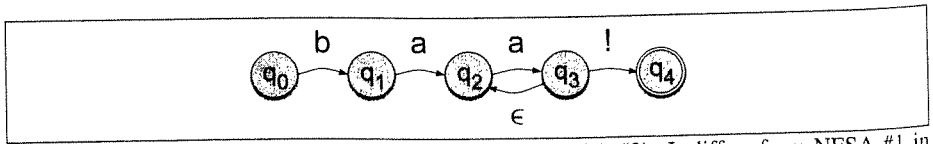

Figure 2.18 Another NFSA for the sheep language (NFSA #2). It differs from NFSA #1 in Fig. 2.17 in having an ϵ-transition.

We interpret this new arc as follows: If we are in state q_3, we are allowed to move to state q_2 *without* looking at the input or advancing our input pointer. So this introduces another kind of non-determinism—we might not know whether to follow the ϵ-transition or the ! arc.

2.2.5 Use of an NFSA to Accept Strings

If we want to know whether a string is an instance of sheeptalk and if we use a non-deterministic machine to recognize it, we might follow the wrong arc and reject it when we should have accepted it. That is, since there is more than one choice at some point, we might take the wrong choice. This problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing. There are three standard **solutions to the problem of non-determinism**:

Backup

- **Backup:** Whenever we come to a choice point, we could put a *marker* to mark where we were in the input and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.

Look-ahead

- **Look-ahead:** We could look ahead in the input to help us decide which path to take.

Parallelism

- **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

We focus here on the backup approach and defer discussion of the look-ahead and parallelism approaches to later chapters.

The backup approach suggests that we should blithely make choices that might lead to dead ends, knowing that we can always return to the unexplored alternatives. There are two keys to this approach: we need to remember all the alternatives for each choice point, and we need to store sufficient information about each alternative so that we can return to it when necessary. When a backup algorithm reaches a point in its processing where no progress can be made (because it runs out of input or has no legal transitions), it returns to a previous choice point, selects one of the unexplored alternatives, and continues from there. Applying this notion to our non-deterministic recognizer, we need only remember two things for each choice point: the state, or node, of the machine that we can go to and the corresponding position on the tape. We will call the combination of the node and position the **search-state** of the recognition algorithm. To avoid confusion, we will refer to the state of the automaton (as opposed to the state of the search) as a **node** or a **machine-state**.

Search-state

Before going on to describe the main part of this algorithm, we should note two changes to the transition table that drives it. First, to represent nodes that have outgoing ϵ-transitions, we add a new **ϵ-column** to the transition table. If a node has an ϵ-transition, we list the destination node in the ϵ-column for that node's row. The second addition is needed to account for multiple transitions to different nodes from the same input symbol. We let each cell entry consist of a list of destination nodes rather than a single node. On the right we show the transition table for the machine in Fig. 2.17 (NFSA #1). While it has no ϵ-transitions, it does show that in machine-state q_2, the input *a* can lead back to q_2 or on to q_3.

	Input			
State	b	a	!	ϵ
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Figure 2.19 shows the algorithm for using a non-deterministic FSA to recognize an input string. The function ND-RECOGNIZE uses the variable *agenda* to keep track of all the currently unexplored choices generated during the course of processing. Each

choice (search-state) is a tuple consisting of a node (state) of the machine and a position on the tape. The variable *current-search-state* represents the branch choice being currently explored.

ND-RECOGNIZE begins by creating an initial search-state and placing it on the agenda. For now we don't specify in what order the search-states are placed on the agenda. This search-state consists of the initial machine-state of the machine and a pointer to the beginning of the tape. The function NEXT is then called to retrieve an item from the agenda and assign it to the variable *current-search-state*.

As with D-RECOGNIZE, the first task of the main loop is to determine if the entire contents of the tape have been successfully recognized. This is done by a call to ACCEPT-STATE?, which returns *accept* if the current search-state contains both an accepting machine-state and a pointer to the end of the tape. If we're not done, the machine generates a set of possible next steps by calling GENERATE-NEW-STATES, which creates search-states for any ϵ-transitions and any normal input-symbol transitions from the transition table. All of these search-state tuples are then added to the current agenda.

Finally, we attempt to get a new search-state to process from the agenda. If the agenda is empty, we've run out of options and have to reject the input. Otherwise, an unexplored option is selected and the loop continues.

It is important to understand why ND-RECOGNIZE returns a value of reject only when the agenda is found to be empty. Unlike D-RECOGNIZE, it does not return reject when it reaches the end of the tape in a non-accept machine-state or when it finds itself unable to advance the tape from some machine-state. This is because, in the nondeterministic case, such roadblocks indicate failure only down a given path, not overall failure. We can only be sure we can reject a string when all possible choices have been examined and found lacking.

Figure 2.20 illustrates the progress of ND-RECOGNIZE as it attempts to handle the input `baaa!`. Each strip illustrates the state of the algorithm at a given point in its processing. The *current-search-state* variable is captured by the solid bubbles representing the machine-state along with the arrow representing progress on the tape. Each strip lower down in the figure represents progress from one *current-search-state* to the next.

Little of interest happens until the algorithm finds itself in state q_2 while looking at the second a on the tape. An examination of the entry for transition-table[q_2,a] returns both q_2 and q_3. Search states are created for each of these choices and placed on the agenda. Unfortunately, our algorithm chooses to move to state q_3, a move that results in neither an accept state nor any new states since the entry for transition-table[q_3, a] is empty. At this point, the algorithm simply asks the agenda for a new state to pursue. Since the choice of returning to q_2 from q_2 is the only unexamined choice on the agenda, it is returned with the tape pointer advanced to the next a. Somewhat diabolically, ND-RECOGNIZE finds itself faced with the same choice. The entry for transition-table[q_2,a] still indicates that looping back to q_2 or advancing to q_3 are valid choices. As before, states representing both are placed on the agenda. These search states are not the same as the previous ones since their tape index values have advanced. This time the agenda provides the move to q_3 as the next move. The move to q_4, and success, is then uniquely determined by the tape and the transition table.

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
      return accept
    else
      agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node,ϵ], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
  then
    return true
  else
    return false
```

Figure 2.19 An algorithm for NFSA recognition. The word *node* means a state of the FSA, and *state* or *search-state* means "the state of the search process", i.e., a combination of *node* and *tape position*.

2.2.6 Recognition as Search

ND-RECOGNIZE accomplishes the task of recognizing strings in a regular language by providing a way to systematically explore all the possible paths through a machine. If this exploration yields a path ending in an accept state, ND-RECOGNIZE accepts the string; otherwise, it rejects the string. This systematic exploration is made possible by the agenda mechanism, which on each iteration selects a partial path to explore and keeps track of any remaining, as yet unexplored, partial paths.

State-space search

Algorithms, such as ND-RECOGNIZE, which operate by systematically searching for solutions, are known as **state-space search** algorithms. In such algorithms, the problem definition creates a space of possible solutions; the goal is to explore this space, returning an answer when one is found or rejecting the input when the space

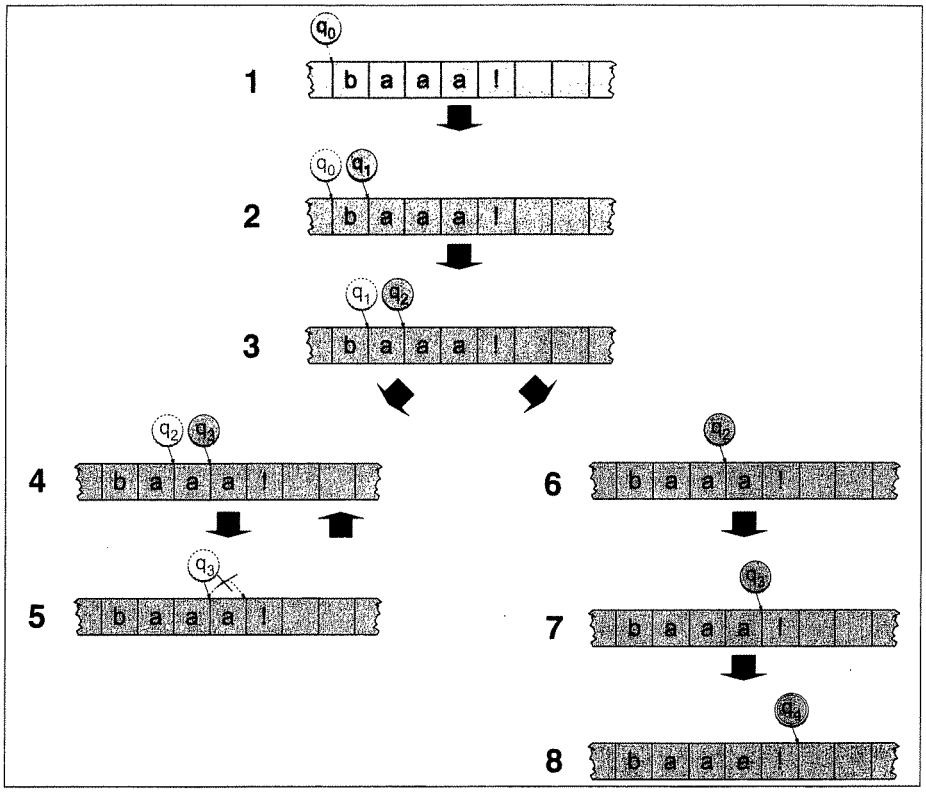

Figure 2.20 Tracing the execution of NFSA #1 (Fig. 2.17) on some sheeptalk.

has been exhaustively explored. In ND-RECOGNIZE, search states consist of pairings of machine-states with positions on the input tape. The state-space consists of all the pairings of machine-state and tape positions that are possible given the machine in question. The goal of the search is to navigate through this space from one state to another, looking for a pairing of an accept state with an end of tape position.

The key to the effectiveness of such programs is often the *order* in which the states in the space are considered. A poor ordering of states may lead to the examination of a large number of unfruitful states before a successful solution is discovered. Unfortunately, it is typically not possible to tell a good choice from a bad one, and often the best we can do is to ensure that each possible solution is eventually considered.

Careful readers may have noticed that the ordering of states in ND-RECOGNIZE has been left unspecified. We know only that unexplored states are added to the agenda as they are created and that the (undefined) function NEXT returns an unexplored state from the agenda when asked. How should the function NEXT be defined? Consider an ordering strategy by which the states that are considered next are the most recently created ones. We can implement such a policy by placing newly created states at the front of the agenda and having NEXT return the state at the front of the agenda when called. Thus, the agenda is implemented by a **stack**. This is commonly referred to as a

Depth-first

depth-first search or **last in first out** (**LIFO**) strategy.

Such a strategy dives into the search space following newly developed leads as they are generated. It will only return to consider earlier options when progress along a current lead has been blocked. The trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Fig. 2.20 illustrates a depth-first search. The algorithm hits the first choice point after seeing `ba` when it has to decide whether to stay in q_2 or advance to state q_3. At this point, it chooses one alternative and follows it until sure the choice was wrong. The algorithm then backs up and tries another older alternative.

Depth-first strategies have one major pitfall: under certain circumstances they can enter an infinite loop. This is possible either if the search space happens to be set up in such a way that a search-state can be accidentally revisited, or if there are an infinite number of search states. We revisit this question when we turn to more complicated search problems in parsing in Chapter 13.

The second way to order the states in the search space is to consider states in the order in which they are created. We can implement such a policy by placing newly created states at the back of the agenda and still have NEXT return the state at the front of the agenda. Thus, the agenda is implemented via a **queue**. This is commonly referred to as a **breadth-first search** or **first in first out** (**FIFO**) strategy. Consider a different trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Fig. 2.21. Again, the algorithm hits its first choice point after seeing `ba` when it had to decide whether to stay in q_2 or advance to state q_3. But now rather than picking one choice and following it up, we imagine examining all possible choices, expanding one ply of the search tree at a time.

Breadth-first

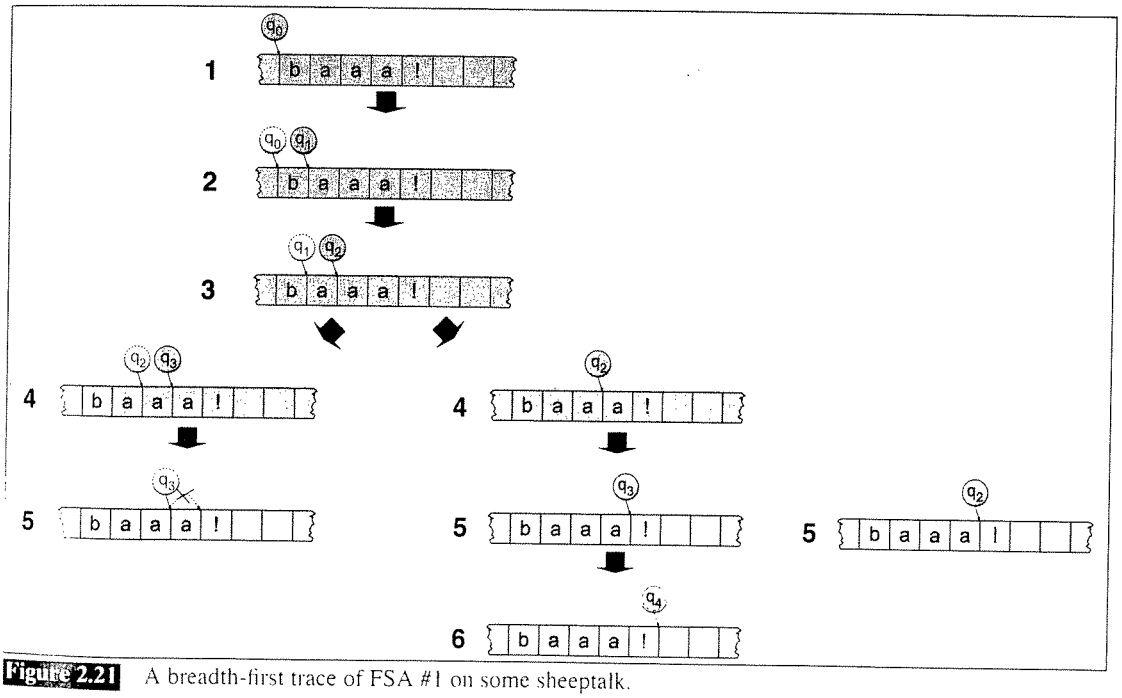

Figure 2.21 A breadth-first trace of FSA #1 on some sheeptalk.

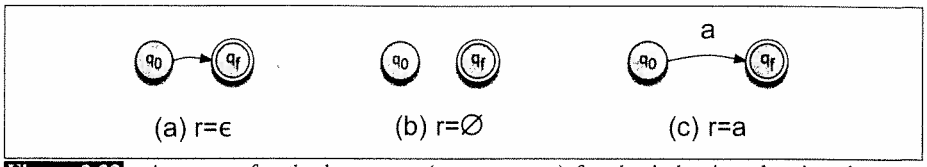

Figure 2.22 Automata for the base case (no operators) for the induction showing that any regular expression can be turned into an equivalent automaton.

- **concatenation**: As shown in Fig. 2.23, we just string two FSAs next to each other by connecting all the final states of FSA_1 to the initial state of FSA_2 by an ϵ-transition.

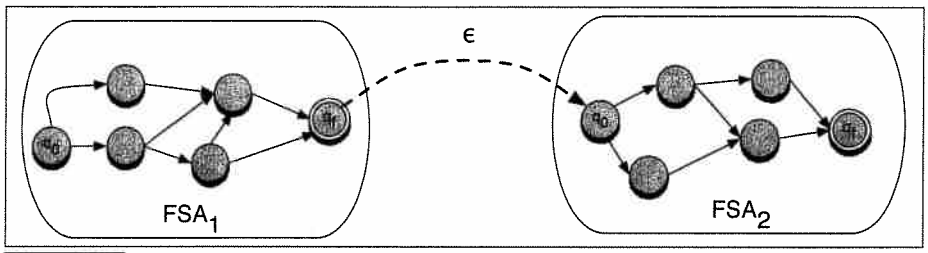

Figure 2.23 The concatenation of two FSAs.

- **closure**: As shown in Fig. 2.24, we create a new final and initial state, connect the original final states of the FSA back to the initial states by ϵ-transitions (this implements the repetition part of the Kleene *), and then put direct links between the new initial and final states by ϵ-transitions (this implements the possibility of having *zero* occurrences). We'd omit this last part to implement Kleene + instead.

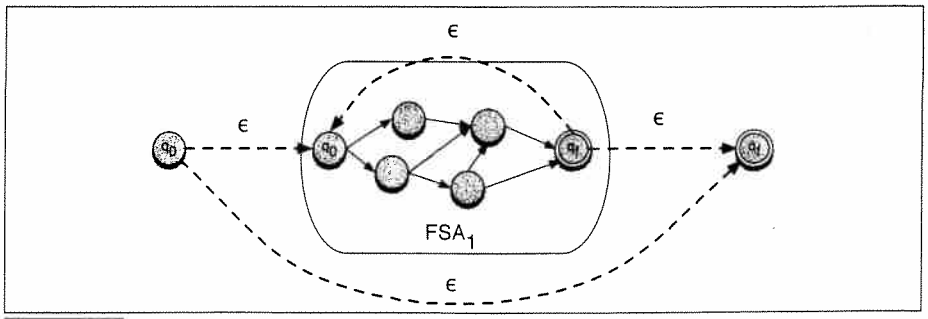

Figure 2.24 The closure (Kleene *) of an FSA.

- **union**: Finally, as shown in Fig. 2.25, we add a single new initial state q_0', and add new ϵ-transitions from it to the former initial states of the two machines to be joined.

We return to regular languages and regular grammars in Chapter 16.

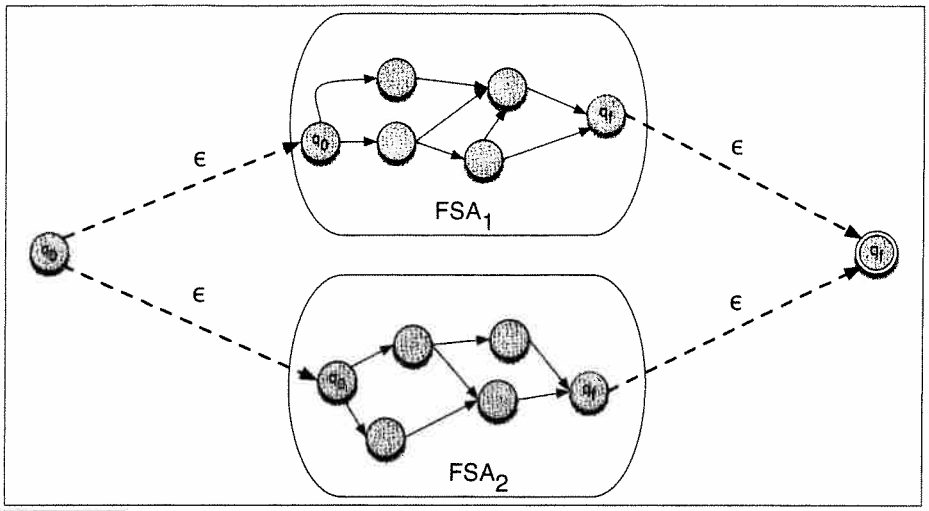

Figure 2.25 The union (|) of two FSAs.

2.4 Summary

This chapter introduced the most important fundamental concept in language processing, the **finite automaton**, and the practical tool based on automaton, the **regular expression**. Here's a summary of the main points we covered about these ideas:

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols (`[]`, `|`, and `.`), **counters** (`*`, `+`, and `{n,m}`), **anchors** (`^`, `$`) and precedence operators (`(,)`).
- Any regular expression can be realized as a **finite-state automaton** (**FSA**).
- Memory (`\1` together with `()`) is an advanced operation that is often considered part of regular expressions but that cannot be realized as a finite automaton.
- An automaton implicitly defines a **formal language** as the set of strings the automaton **accepts** over any vocabulary (set of symbols).
- The behavior of a **deterministic** automaton (**DFSA**) is fully determined by the state it is in.
- A **non-deterministic** automaton (**NFSA**) sometimes has to choose between multiple paths to take given the same current state and next input.
- Any **NFSA** can be converted to a **DFSA**.
- The order in which an **NFSA** chooses the next state to explore on the agenda defines its **search strategy**. The **depth-first search** or **LIFO** strategy corresponds to the agenda-as-stack; the **breadth-first search** or **FIFO** strategy corresponds to the agenda-as-queue.
- Any regular expression can automatically be compiled into a **NFSA** and hence into a **FSA**.

Bibliographical and Historical Notes

Finite automata arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. The Turing machine was an abstract machine with a finite control and an input/output tape. In one move, the Turing machine could read a symbol on the tape, write a different symbol on the tape, change state, and move left or right. Thus, the Turing machine differs from a finite-state automaton mainly in its ability to change the symbols on its tape.

McCulloch-Pitts neuron

Inspired by Turing's work, McCulloch and Pitts built an automata-like model of the neuron (see von Neumann, 1963, p. 319). Their model, which is now usually called the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), was a simplified model of the neuron as a kind of "computing element" that could be described in terms of propositional logic. The model was a binary device, at any point either active or not, that took excitatory and inhibitory input from other neurons and fired if its activation passed some fixed threshold. Based on the McCulloch-Pitts neuron, Kleene (1951) and (1956) defined the finite automaton and regular expressions and proved their equivalence. Non-deterministic automata were introduced by Rabin and Scott (1959), who also proved them equivalent to deterministic ones.

Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the Unix `grep` utility.

There are many general-purpose introductions to the mathematics underlying automata theory, such as Hopcroft and Ullman (1979) and Lewis and Papadimitriou (1988). These cover the mathematical foundations of the simple automata of this chapter, as well as the finite-state transducers of Chapter 3, the context-free grammars of Chapter 12, and the Chomsky hierarchy of Chapter 16. Friedl (1997) is a useful comprehensive guide to the advanced use of regular expressions.

The metaphor of problem-solving as search is basic to Artificial Intelligence (AI); more details on search can be found in any AI textbook such as Russell and Norvig (2002).

Exercises

2.1 Write regular expressions for the following languages. You may use either Perl/Python notation or the minimal "algebraic" notation of Section 2.3, but make sure to say which one you are using. By "word", we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.

1. the set of all alphabetic strings;
2. the set of all lower case alphabetic strings ending in a *b*;

3. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug");
4. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;
5. all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;
6. all strings that have both the word *grotto* and the word *raven* in them (but not, e.g., words like *grottos* that merely *contain* the word *grotto*);
7. write a pattern that places the first word of an English sentence in a register. Deal with punctuation.

2.2 Implement an ELIZA-like program, using substitutions such as those described on page 26. You may choose a different domain than a Rogerian psychologist, if you wish, although keep in mind that you would need a domain in which your program can legitimately engage in a lot of simple repetition.

2.3 Complete the FSA for English money expressions in Fig. 2.15 as suggested in the text following the figure. You should handle amounts up to $100,000, and make sure that "cent" and "dollar" have the proper plural endings when appropriate.

2.4 Design an FSA that recognizes simple date expressions like *March 15, the 22nd of November, Christmas*. You should try to include all such "absolute" dates (e.g., not "deictic" ones relative to the current day, like *the day before yesterday*). Each edge of the graph should have a word or a set of words on it. You should use some sort of shorthand for classes of words to avoid drawing too many arcs (e.g., furniture → desk, chair, table).

2.5 Now extend your date FSA to handle deictic expressions like *yesterday, tomorrow, a week from tomorrow, the day before yesterday, Sunday, next Monday, three weeks from Saturday*.

2.6 Write an FSA for time-of-day expressions like *eleven o'clock, twelve-thirty, midnight*, or *a quarter to ten*, and others.

2.7 (Thanks to Pauline Welby; this problem probably requires the ability to knit.) Write a regular expression (or draw an FSA) that matches all knitting patterns for scarves with the following specification: *32 stitches wide, K1P1 ribbing on both ends, stockinette stitch body, exactly two raised stripes*. All knitting patterns must include a cast-on row (to put the correct number of stitches on the needle) and a bind-off row (to end the pattern and prevent unraveling). Here's a sample pattern for one possible scarf matching the above description:[2]

[2] *Knit* and *purl* are two different types of stitches. The notation K*n* means do *n* knit stitches. Similarly for purl stitches. Ribbing has a striped texture—most sweaters have ribbing at the sleeves, bottom, and neck. Stockinette stitch is a series of knit and purl rows that produces a plain pattern—socks or stockings are knit with this basic pattern, hence the name.

1.	Cast on 32 stitches.	*cast on; puts stitches on needle*
2.	K1 P1 across row (i.e., do (K1 P1) 16 times).	*K1P1 ribbing*
3.	Repeat instruction 2 seven more times.	*adds length*
4.	K32, P32.	*stockinette stitch*
5.	Repeat instruction 4 an additional 13 times.	*adds length*
6.	P32, P32.	*raised stripe stitch*
7.	K32, P32.	*stockinette stitch*
8.	Repeat instruction 7 an additional 251 times.	*adds length*
9.	P32, P32.	*raised stripe stitch*
10.	K32, P32.	*stockinette stitch*
11.	Repeat instruction 10 an additional 13 times.	*adds length*
12.	K1 P1 across row.	*K1P1 ribbing*
13.	Repeat instruction 12 an additional 7 times.	*adds length*
14.	Bind off 32 stitches.	*binds off row: ends pattern*

2.8 Write a regular expression for the language accepted by the NFSA in Fig. 2.26.

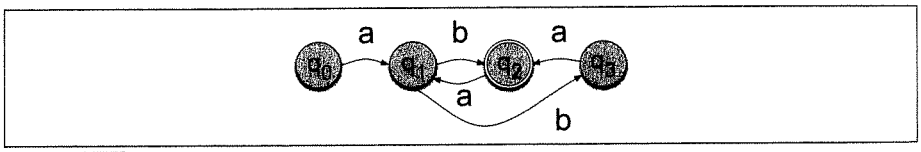

Figure 2.26 A mystery language.

2.9 Currently the function D-RECOGNIZE in Fig. 2.12 solves only a subpart of the important problem of finding a string in some text. Extend the algorithm to solve the following two deficiencies: (1) D-RECOGNIZE currently assumes that it is already pointing at the string to be checked, and (2) D-RECOGNIZE fails if the string it is pointing to includes as a proper substring a legal string for the FSA. That is, D-RECOGNIZE fails if there is an extra character at the end of the string.

2.10 Give an algorithm for negating a deterministic FSA. The negation of an FSA accepts exactly the set of strings that the original FSA rejects (over the same alphabet) and rejects all the strings that the original FSA accepts.

2.11 Why doesn't your previous algorithm work with NFSAs? Now extend your algorithm to negate an NFSA.

Chapter 3
Words and Transducers

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

Chapter 2 introduced the regular expression, showing, for example, how a single search string could help us 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*, 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 we pluralize English words ending in *-y* 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 a vowel change.

Morphological parsing
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 means taking an input and producing some sort of linguistic structure for it. We use the term parsing very broadly throughout this book to include many kinds of structures that might be produced; morphological, syntactic, semantic, discourse; in the form of a string, 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 form

Morphological parsing is important throughout speech and language processing. It plays a crucial role in Web search for morphologically complex languages like Russian or German; in Russian the word *Moscow* has different endings in the phrases *Moscow*, *of Moscow*, *from Moscow*, and so on. We want to be able to automatically search for the inflected forms of the word even if the user only typed in the base form. Morphological parsing also plays a crucial role in part-of-speech tagging for these morphologically complex languages, as we show in Chapter 5. It is important for producing the large dictionaries that are necessary for robust spell-checking. We need it in machine 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? Sometimes we can do this, and, for example, for English speech

[1] See, for example, Seuss (1960).

recognition this is exactly what we do. But for many NLP applications this isn't possible because *-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 these:

(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 return to them again in later chapters.

After describing morphological parsing, we 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 (IR) and web search, 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 describe a simple stemming algorithm called the **Porter stemmer**.

Stemming

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]

Lemmatization

Tokenization

Next, we 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, for many applications we need to know how similar two words are orthographically. Morphological parsing is one method for computing this similarity; another is to use the **minimum edit distance** algorithm to compare the letters in the two words. We introduce this important NLP algorithm and also show how it can be used in spelling correction.

3.1 Survey of (Mostly) English Morphology

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 one morpheme (the morpheme *fox*) and the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*.

Morpheme

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, and the affixes add "additional" meanings of various kinds.

Stem

Affix

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 really have 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 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*) plus three affixes (*un-*, *-able*, and *-ly*). While English doesn't tend to stack more than four

[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 Chapter 20.

[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).

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 as 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 some 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*).

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).

Regular verb

In this chapter, we are mostly concerned with the main and primary verbs because these have inflectional endings. Of these verbs a large class are **regular**, that is, 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 follows:

Morphological Class	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 show in Chapter 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 verb

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 irregular verbs constitute 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*), its vowel and some consonants (*catch/caught*), or with no change at all (*cut/cut*).

Preterite

Morphological Class	Irregularly Inflected Verbs		
stem	eat	catch	cut
-s form	eats	catches	cuts
-ing participle	eating	catching	cutting
preterite	ate	caught	cut
past participle	eaten	caught	cut

The way these forms are used in a sentence is discussed in the syntax and semantics chapters 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 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

Progressive

[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.

Gerund
Perfect

mark present or ongoing activity (*It is raining*) or when the verb is treated as a noun; this latter kind of nominal use of a verb is called a **gerund**: *Fishing is fine if you live near water.* The *-ed/-en* 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 the *-ing* and *-ed* suffixes (*beg/begging/begged*) is added. 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 *-ing* and *-ed* (*merge/merging/merged*) are added. 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 50 distinct verb forms for each regular verb. Figure 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 Indicative	**Imperfect Indicative**	**Future**	**Preterite**	**Present Subjunctive**	**Conditional**	**Imperfect Subjunctive**	**Future Subjunctive**
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 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 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 between that of an affix and a word. The phonological behavior of clitics is like affixes; they tend to be short and unaccented (we talk more about phonology in Chapter 8). Their syntactic behavior is more like words, often acting as pronouns, articles, conjunctions, or verbs. Clitics preceding a word are called **proclitics**, and those following are **enclitics**.

Proclitic
Enclitic

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 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 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

The kind of morphology we have discussed so far, in which a word is composed of a string of concatenated morphemes 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 common in Arabic, Hebrew, and other Semitic languages. In Hebrew, for example, a verb (as well as other parts-of-speech) is constructed from 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

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 (e.g., 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.

Agreement

Gender

Noun class

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 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 is to take input forms like those in the first and third columns of Fig. 3.2 and produce output forms like those in the second and fourth column.

The second column contains the stem of each word as well as assorted morphological **features**. These features specify additional information about the stem. For

Feature

English		Spanish		
Input	**Morphological Parse**	**Input**	**Morphological Parse**	**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 +3P +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.

example, the feature +N means that the word is a noun; +Sg means it is singular; +Pl that it is plural. Morphological features are referred to again in Chapter 5 and in more detail in Chapter 15; 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*) are ambiguous between different morphological parses. For now, we will consider the goal of morphological parsing merely to list all possible parses. We return to the task of disambiguating among morphological parses in Chapter 5.

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.

Orthographic rules

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 $y \rightarrow ie$ spelling rule discussed above that changes *city* + *-s* to *cities* rather than *citys*).

The next section discusses 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 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 Construction of 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.

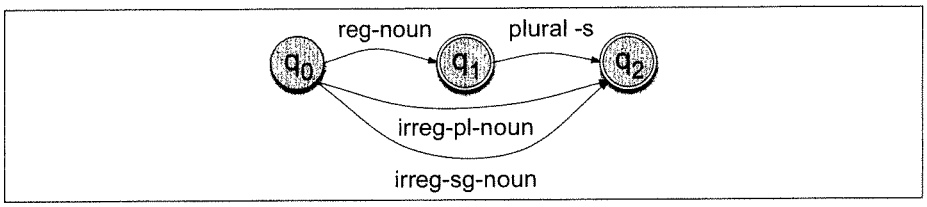

Figure 3.3 A finite-state automaton for English nominal inflection.

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.

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

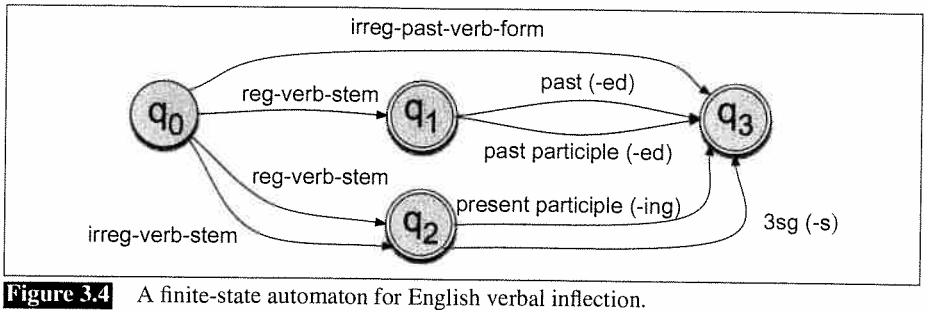

Figure 3.4 A finite-state automaton for English verbal inflection.

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 Chapter 12 (see also (Sproat, 1993)).

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

big, bigger, biggest,	cool, cooler, coolest, coolly
happy, happier, happiest, happily	red, redder, reddest
unhappy, unhappier, unhappiest, unhappily	real, unreal, really
clear, clearer, clearest, clearly, unclear, unclearly	

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 FSA in Fig. 3.5.

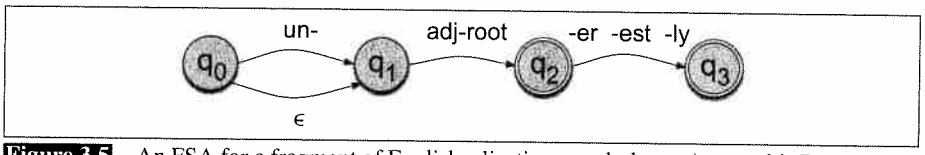

Figure 3.5 An FSA for a fragment of English adjective morphology: Antworth's Proposal #1.

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*), and **adj-root**$_2$ will include adjectives that can't (*big*, *small*), and so on.

This gives an idea of the complexity to be expected from English derivation. As a further example, we give in Fig. 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.

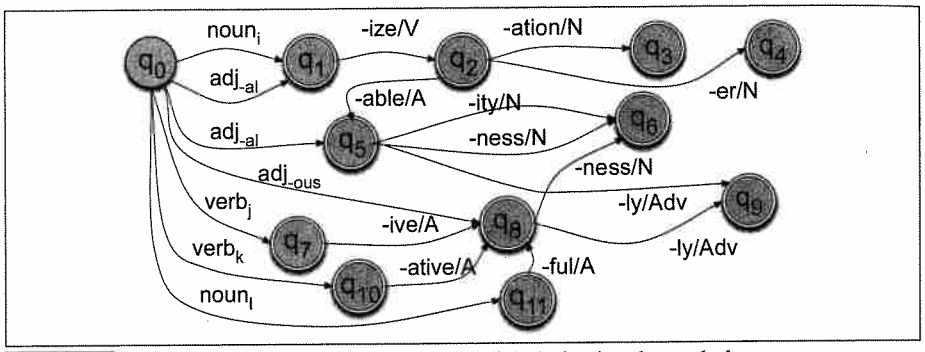

Figure 3.6 An FSA for another fragment of English derivational morphology.

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. We do this by taking the morphotactic FSAs and plugging 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 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 Chapter 2.

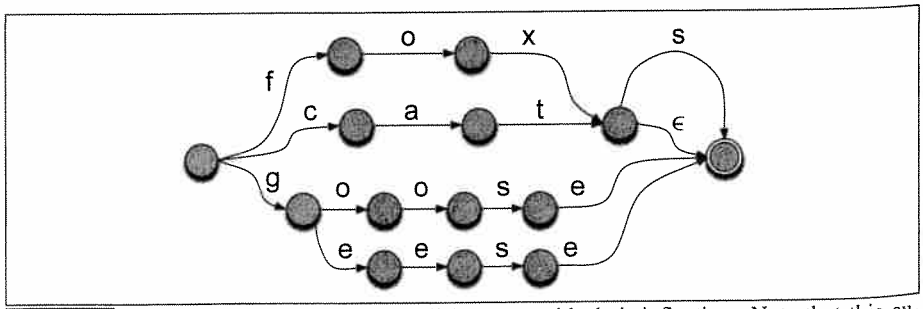

Figure 3.7 Expanded FSA for a few English nouns with their inflection. Note that this automaton will incorrectly accept the input *foxs*. We see, beginning on page 62, how to correctly deal with the inserted *e* in *foxes*.

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 shows how transducers can be applied to morphological parsing.

FST

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 that 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. Figure 3.8 shows an example of an FST where each arc is labeled by an input and output string, separated by a colon.

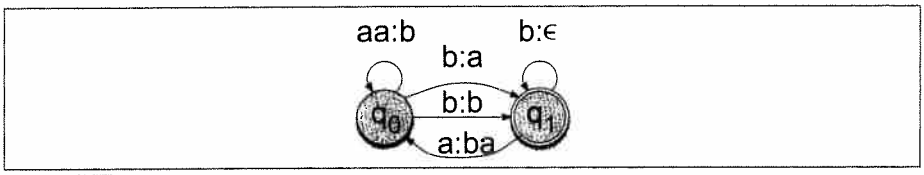

Figure 3.8 A finite-state transducer.

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 fourfold 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 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 seven 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 as to 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 relation

Whereas 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 are more likely to have such closure properties). Besides union, FSTs have two additional closure properties that turn out to be extremely useful:

Intersection

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 an FST-as-parser into an FST-as-generator.

Composition is useful because it allows us to replace two transducers that run in series 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, $T_1 \circ T_2(S) = T_2(T_1(S))$. Figure 3.9 shows, for example, the composition of `[a:b]+` with `[b:c]+` to produce `[a:c]+`.

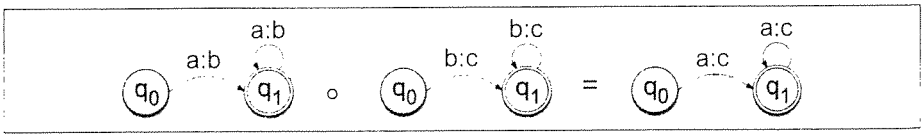

Figure 3.9 The composition of `[a:b]+` with `[b:c]+` to produce `[a:c]+`.

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 Chapter 2, making FSTs quite slow in the general case. This suggests that it would nice to have an algorithm to convert a non-deterministic 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. Figure 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.

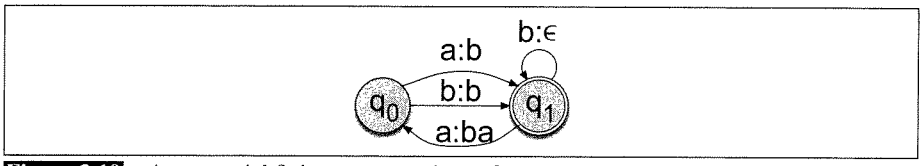

Figure 3.10 A sequential finite-state transducer, from Mohri (1997).

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 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

A generalization of sequential transducers, the **subsequential transducer**, generates an additional output string at the final states, concatenating it onto the output produced so far (Schützenberger, 1977). 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 that 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 Chapter 2.

While both sequential and subsequential transducers are deterministic and efficient, neither of them can 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. Figure 3.11 shows an example of a 2-subsequential FST.

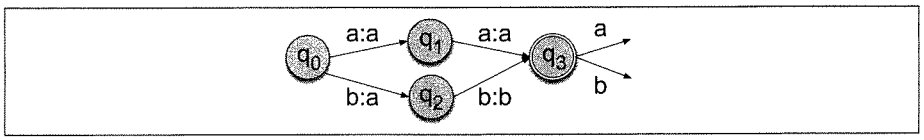

Figure 3.11 A 2-subsequential finite-state transducer, from Mohri (1997).

Mohri (1996, 1997) shows 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 *beber +V +PInd +1P +Sg*, telling us that *bebo* is the present indicative first person singular form of the Spanish verb *beber*, "to drink".

In the **finite-state morphology** paradigm that we 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 making up the actual spelling of the word. Figure 3.12 shows these two levels for (English) *cats*.

Surface level

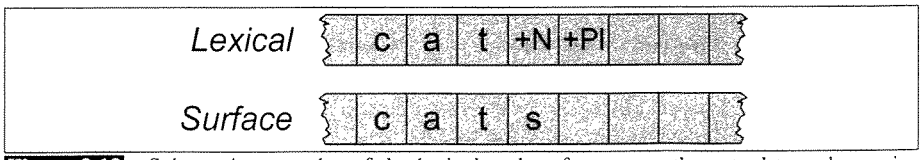

Figure 3.12 Schematic examples of the lexical and surface tapes; the actual transducers involve intermediate tapes as well.

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

Lexical tape

lower or **surface tape** is composed of characters from another alphabet Δ. In the **two-level morphology** of Koskenniemi (1983), each arc is allowed 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 $i:o$, that has 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, whereas an FSA accepts a language stated over a finite alphabet of single symbols, such as the alphabet of our sheep language:

$$\Sigma = \{b, a, !\} \tag{3.2}$$

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

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

Feasible pair

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 pair

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. Figure 3.13 shows an augmentation of Fig. 3.3 with the nominal morphological features (`+Sg` and `+Pl`) that correspond to each morpheme. The symbol ^ indicates a **morpheme boundary**, and the symbol # indicates a **word boundary**. The morphological features map to the empty string ϵ or the boundary symbols since no segment on the output tape corresponds to them.

Morpheme boundary

#

Word boundary

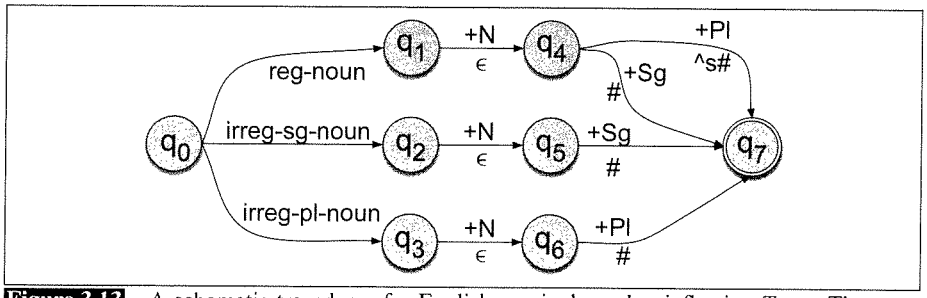

Figure 3.13 A schematic transducer for English nominal number inflection T_{num}. The symbols above each arc represent elements of the morphological parse in the lexical tape; the symbols below each arc represent the surface tape (or the intermediate tape, described later), using the morpheme-boundary symbol ^ and word-boundary marker #. The labels on the arcs leaving q_0 are schematic and must be expanded by individual words in the lexicon.

In order for us 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

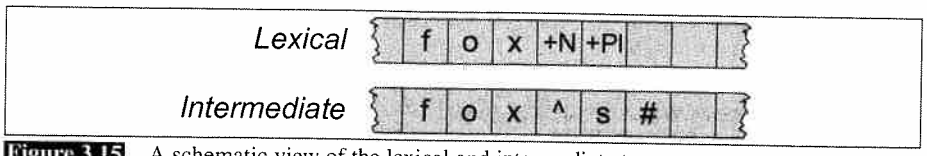

Figure 3.15 A schematic view of the lexical and intermediate tapes.

which 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 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 are some spelling rules:

Spelling rule

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. Figure 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.

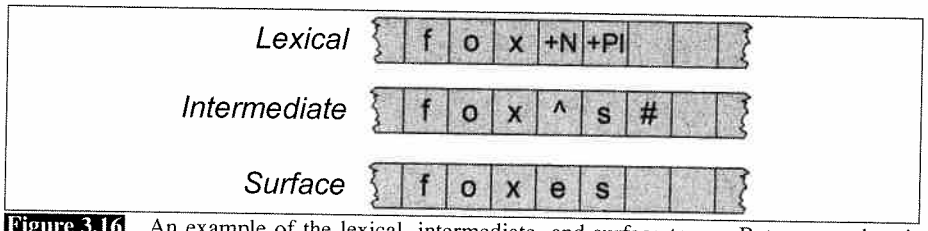

Figure 3.16 An example of the lexical, intermediate, and surface tapes. Between each pair of tapes is a two-level transducer; the lexical transducer of Fig. 3.14 between the lexical and intermediate levels, and the E-insertion spelling rule between the intermediate and surface levels. The E-insertion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

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:

$$\epsilon \rightarrow \text{e} / \left\{ \begin{matrix} x \\ s \\ z \end{matrix} \right\} \hat{} \_\_ \text{s\#} \tag{3.4}$$

This is the rule notation of Chomsky and Halle (1968); a rule of the form $a \rightarrow b / c\_\_d$ 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 ^ indicates a morpheme boundary. These boundaries are deleted by inclusion of the symbol ^: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". Figure 3.17 shows an automaton that corresponds to this rule.

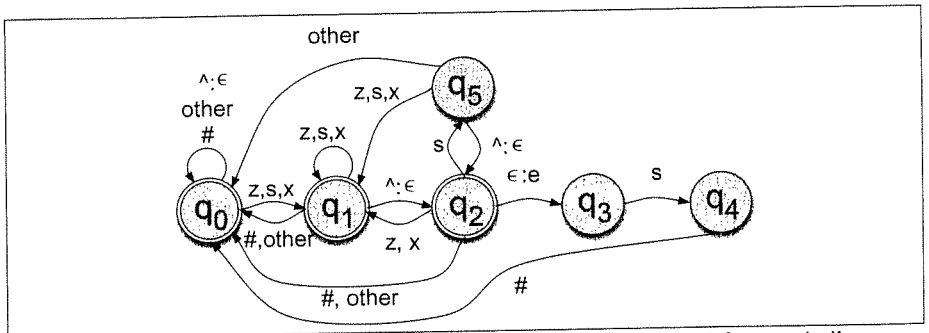

Figure 3.17 The transducer for the E-insertion rule of (3.4), extended from a similar transducer in Antworth (1990). We additionally need to delete the # symbol from the surface string; we can do this either by interpreting the symbol # as the pair #:ϵ or by postprocessing the output to remove word boundaries.

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 ensures 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 allowed only 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". 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). Figure 3.18 shows the transition table for the rule that makes the illegal transitions explicit with the "–" symbol. The next section shows a trace of this E-insertion transducer running on a sample input string.

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 the E-insertion rule of Fig. 3.17, extended from a similar transducer in Antworth (1990).

3.7 The Combination of an FST Lexicon and Rules

We are now ready to combine our lexicon and rule transducers for parsing and generating. Figure 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 to map between this intermediate level and the surface level. (We could instead 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 +N +PL` to *foxes*.

The power of finite-state transducers is that exactly the 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

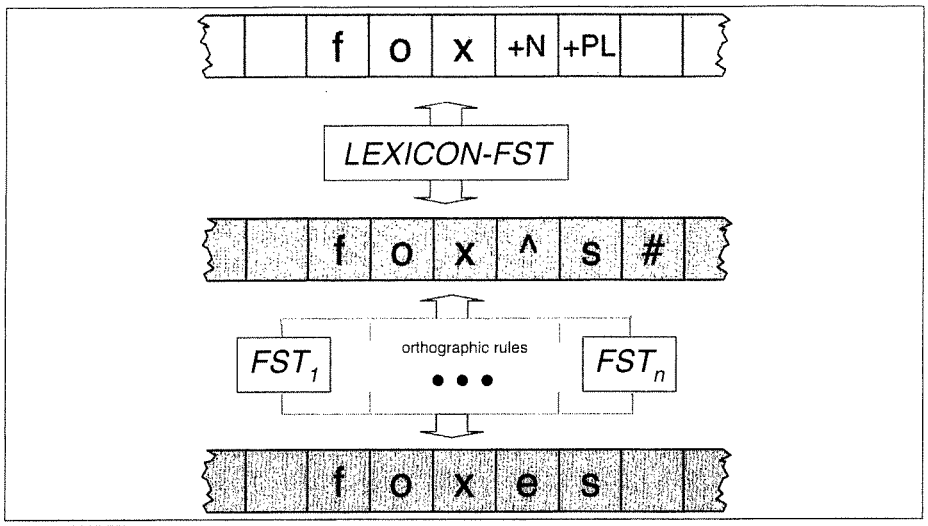

Figure 3.19 Generating or parsing with FST lexicon and rules.

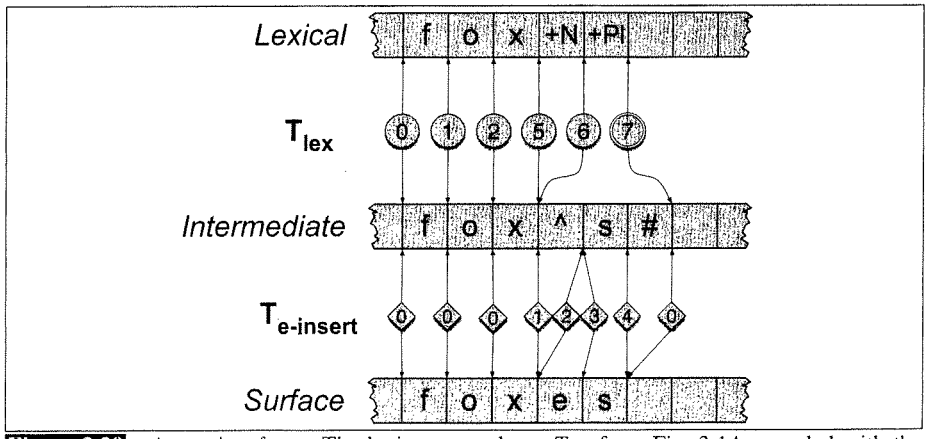

Figure 3.20 Accepting *foxes*: The lexicon transducer T_{lex} from Fig. 3.14 cascaded with the E-insertion transducer in Fig. 3.17.

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 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 discuss such disambiguation algorithms

Disambiguating

in Chapter 5 and Chapter 20. Barring such external evidence, the best our transducer can do is just enumerate the possible choices so we can transduce *fox^s#* into both `fox +V +3SG` and `fox +N +PL`.

There is a kind of ambiguity that we do 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 (e.g., 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. 2.19 in Chapter 2 to do FST parsing.

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

Running a cascade can be made more efficient by **composing** and **intersecting** the transducers. We've already seen how to compose a cascade of transducers in series into a single more complex transducer. The intersection of two transducers/relations F and G ($F \wedge G$) defines a relation R such that R(x,y) if and only if F(x,y) and G(x,y). While transducers in general are not closed under intersection, as discussed on page 58, transducers between strings of equal length (without ϵ) are, and two-level rules can be written this way by treating the ϵ symbol as an ordinary symbol in the rule system. The

Intersection

intersection algorithm takes the Cartesian product of the states, that is, 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}. Figure 3.21 sketches how this intersection ($\wedge$) and composition ($\circ$) process might be carried out.

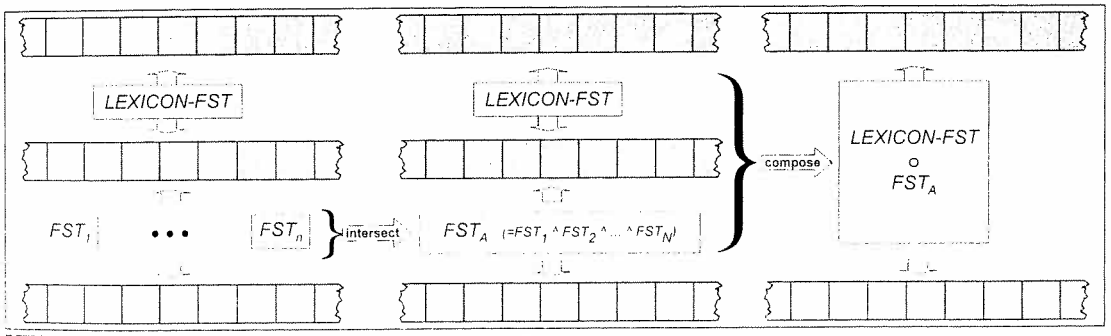

Figure 3.21 Intersection and composition of transducers.

Since there are a number of rule-to-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

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 IR tasks like Web search (Chapter 23), in which a query such as a Boolean combination of relevant **keywords** or phrases, for example, (*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.

Keyword

Stemming

Porter stemmer

One of the most widely used **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, the Porter algorithm also can be viewed as a lexicon-free FST stemmer (this idea is developed further in the exercises (Exercise 3.6). The algorithm contains a series of rules like these:

$$\begin{aligned} \text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\ \text{ING} &\rightarrow \epsilon \quad \text{if stem contains vowel (e.g., motoring} \rightarrow \text{motor)} \\ \text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)} \end{aligned}$$

Detailed rule lists for the Porter stemmer, as well as code (in Java, Python, etc.) can be found on Martin Porter's homepage; see also the original paper (Porter, 1980).

Stemming tends to 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). But lexicon-free stemmers like the Porter algorithm, while simpler than full lexicon-based morphological parsers, commit errors like the following (Krovetz, 1993):

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

Modern stemmers tend to be more complicated because, for example, we don't want to stem, say the word *Illustrator* to *illustrate*, since the capitalized form *Illustrator* tends to refer to the software package. We return to this issue in Chapter 23.

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 related problem

Tokenization

of segmenting running text into words and sentences. This task is called **tokenization**.

Word tokenization may seem simple in a language like English that separates words by a special 'space' character. As shown later, 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 even for English. Consider the following sentences from *Wall Street Journal* and *New York Times* articles, respectively:

```
Mr.  Sherwood said reaction to Sea Containers' proposal
has been "very positive." In New York Stock Exchange
composite trading yesterday, Sea Containers closed at
$62.625, up 62.5 cents.


''I said, 'what're you?  Crazy?'  ''  said Sadowsky.  ''I
can't afford to do that.''
```

Segmenting purely on whitespace 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 number 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, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, *555 500,50*.

A tokenizer can also be used 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 requires ambiguity resolution, since apostrophes 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, a process called *named entity detection* which is discussed in Chapter 22.

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) that 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 are introduced in later chapters. But a useful first step can still be taken through a sequence of regular expressions. We introduce here the first part; a word tokenization algorithm. Figure 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.

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) describe 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.

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 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.

Maximum matching

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 that 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 En-

```
#!/usr/bin/perl

$letternumber = "[A-Za-z0-9]";
$notletter = "[^A-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 ($line = <>){ # read the next line from standard input

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

    # put whitespace around commas that aren't inside numbers
    $line =~ s/([^0-9]),/$1 , /g;
    $line =~ s/,([^0-9])/ , $1/g;

    # distinguish singlequotes from apostrophes by
    # segmenting off single quotes not preceded by letter
    $line =~ s/^'/$& /g;
    $line =~ s/($notletter)'/$1 '/g;

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

   # now deal with periods.  For each possible word
   @possiblewords=split(/\s+/,$line);
   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 =~
                  /^([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.

glish 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. The most successful modern algorithms for Chinese word segmentation are based on machine learning from hand-segmented training sets. We return to these algorithms after we introduce probabilistic methods in Chapter 5.

3.10 Detection and Correction of 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 Earnest*

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 continue our spell-checking discussion later in Chapter 5 after we define the probabilistic noisy channel model. The detection and correction of spelling errors is an integral part of modern word processors and search engines. It is also important in correcting errors in

OCR **optical character recognition (OCR)**, the automatic recognition of machine or hand-printed characters, and in **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) that 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 helpful than harmful 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 in which 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 that 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 is discussed in Chapter 5. 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

String 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.

Many important algorithms for finding string distance rely on some version of the

[5] Early spell-checkers, by contrast, allowed any word to have any suffix; thus, early versions of Unix `spell` accepted bizarre prefixed words like *unschlm* and *antiundoggingly* and suffixed words from *the*, like *thehood* and *theness*.

Minimum edit distance

minimum edit distance algorithm, named by Wagner and Fischer (1974) but independently discovered by many people (summarized later, in the Historical Notes section of Chapter 6). 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. Given two sequences, an **alignment** is a correspondence between substrings of the two sequences. Thus, I aligns with the empty string, N with E, T with X, and so on. Beneath the aligned strings is another representation; 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.

Alignment

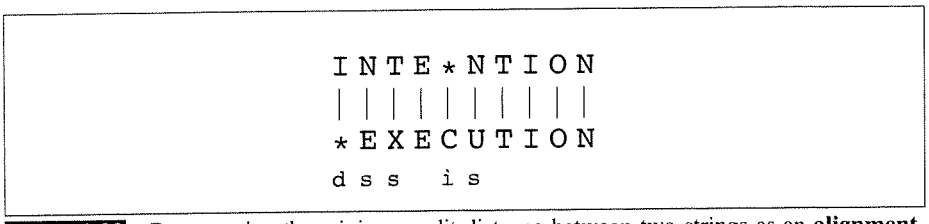

Figure 3.23 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

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 distance between *intention* and *execution* is 5. Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of 1 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 includes the most commonly used algorithms in speech and language processing; besides minimum edit distance, these include the **Viterbi** and **forward** algorithms (Chapter 6) and the **CKY** and **Earley** algorithm (Chapter 13).

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various sub-problems. 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.

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

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

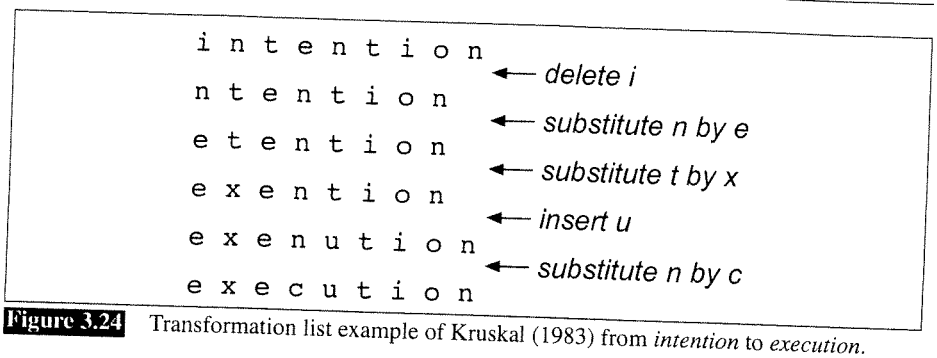

Figure 3.24 Transformation list example of Kruskal (1983) from *intention* to *execution*.

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:

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

The algorithm itself is summarized in Fig. 3.25 and 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 that substitution of identical letters has zero cost).

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

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. Figure 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.

```
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] + sub-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., $\forall x$, 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., sub-cost$(x,x) = 0$).

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* with the 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.

Figure 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 we came 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

Backtrace

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.

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 three neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) by using a **backtrace**, starting at the **8** in the upper-right corner and following the arrows. The sequence of dark grey cells represents one possible minimum cost alignment between the two strings.

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 that uses probabilistic definitions of the operations. Instead of computing the "minimum edit distance" between two strings, Viterbi computes 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 multimorphemic words are represented in the minds of speakers of English. 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 spectrum of possible representations. The **full listing** hypothesis proposes that all words of a language are listed in the mental lexicon without any internal morphological structure. In this view, morphological structure is an epiphenomenon, and *walk*, *walks*, *walked*, *happy*, and *happily* are all separately listed in the lexicon. This hypothesis is 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 always access both morphemes

(*walk* and *-s*) and combine them. This view is probably too strict as well.

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")

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*) seem to be stored separately from their stem (*happy*) but that regularly inflected forms (*pouring*) are not distinct in the lexicon from their stems (*pour*). Stanners et al. 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*. 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*. A Marslen-Wilson et al. (1994) model compatible with their findings is shown in Fig. 3.28.

Priming

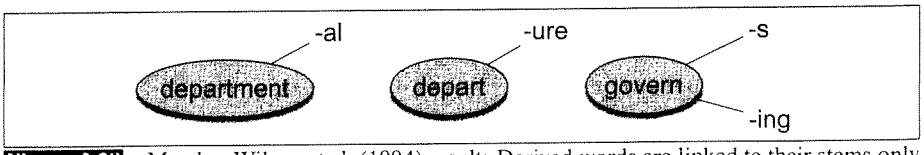

Figure 3.28 Marslen-Wilson et al. (1994) result: Derived words are linked to their stems only if semantically related.

In summary, these results suggest that (at least) productive morphology like inflection does play an online role in the human lexicon. More recent studies have shown effects of non-inflectional morphological structure on word reading time as well, such as the **morphological family size**. The morphological family size of a word is the number of other multimorphemic words and compounds in which it appears; the family for *fear*, for example, includes *fearful*, *fearfully*, *fearfulness*, *fearless*, *fearlessly*, *fearlessness*, *fearsome*, and *godfearing* (according to the CELEX database), for a total size of 9. Baayen and others (Baayen et al., 1997; De Jong et al., 2002; Moscoso del Prado Martín et al., 2004a) have shown that words with a larger morphological family size are recognized faster. Recent work has further shown that word recognition speed

Morphological family size

is affected by the total amount of **information** (or **entropy**) contained by the morphological paradigm (Moscoso del Prado Martín et al., 2004a); entropy will be introduced in the next chapter.

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 that also plays 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* and *-ness* and 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 FST operations include **composition**, **projection**, and **intersection**.
- **Finite-state morphology** and **two-level morphology** are applications of finite-state transducers to morphological representation and parsing.
- Automatic transducer compilers can produce a transducer for any rewrite rule. The lexicon and spelling rules can be combined by **composing** and **intersecting** transducers.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It is not as accurate as a lexicon-based transducer model but is relevant for tasks 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 that are not in a dictionary; an FST dictionary can be useful for this.
- The **minimum edit distance** between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by **dynamic programming**, which also results in an **alignment** of the two strings.

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. Chapter 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 Salomaa (1973) and by Schützenberger (1977).

Early algorithms for morphological parsing used either the **bottom-up** or **top-down** methods that we discuss when we turn to parsing in Chapter 13. An early bottom-up **affix-stripping** approach was Packard's (1973) parser for ancient Greek that 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. 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. Hankamer's (1986) keCi is an early top-down *generate-and-test* or *analysis-by-synthesis* morphological parser for Turkish, guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, applies every possible phonological rule to it, and checks 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 (discussed in Chapter 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 13 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,

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. Chapter 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 Salomaa (1973) and by Schützenberger (1977).

Early algorithms for morphological parsing used either the **bottom-up** or **top-down** methods that we discuss when we turn to parsing in Chapter 13. An early bottom-up **affix-stripping** approach was Packard's (1973) parser for ancient Greek that 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. 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. Hankamer's (1986) keCi is an early top-down *generate-and-test* or *analysis-by-synthesis* morphological parser for Turkish, guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, applies every possible phonological rule to it, and checks 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 (discussed in Chapter 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 13 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).

Readers with further interest in finite-state morphology should turn to Beesley and Karttunen (2003). Readers with further interest in computational models of Arabic and Semitic morphology should see Smrž (1998), Kiraz (2001), and Habash et al. (2005).

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 commonly used as a baseline or when a simple but reasonably 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 Tseng et al. (2005a).

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 automata theory should see Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1988). Roche and Schabes (1997b) 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.

The CELEX dictionary is an extremely useful database for morphological analysis, containing full morphological parses of a large lexicon of English, German, and Dutch (Baayen et al., 1995). Roark and Sproat (2007) is a general introduction to computational issues in morphology and syntax. Sproat (1993) is an older 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 (Knuth, 1973; Odell and Russell, 1922) 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, e.g., 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).

1. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y.
2. 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
3. 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 (e.g., 666 → 6).
4. 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 an FST to implement the Soundex algorithm.

3.6 Read Porter (1980) or see Martin Porter's official homepage on the Porter stemmer. 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 pseudocode introduced in Chapter 2. You should do this by modifying the algorithm ND-RECOGNIZE in Fig. 2.19 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.