Exercise 4.4
Copy `lpcana.c` to `lpcana2.c`. Amend `lpcana2.c` so that it writes out `lp` rather than `coeffs`. Compare the `lp` files with some original audio recordings.

Exercise 4.5
Examine `joe.dat` (or another recorded signal) to determine the approximate location of each vowel. Write down the sample numbers of points approximately ¼ and ¾ of the way through each vowel. Use `lpc_spectrum` to estimate the first three formants of each vowel. Compare them to published tables of vowel formants, such as those in Olive et al. 1993.

Further reading

Most of the topics covered in this chapter are described (without implementation in software) in Wakita 1996. For more about Fourier analysis in C and the Fast Fourier Transform, see Press et al. 1992: 496–510 and 537–53. For more on windowing, see Press et al. 1992: 553–8. Cepstral analysis is well described by Wakita 1996, autocorrelation pitch tracking by Johnson 1997: 33–6 and Linear Predictive Coding by Johnson 1997: 40–4 and Schroeder 1985. The C implementation given in this chapter is closely based on Press et al. 1992: 564–72.

Reading in preparation for the next chapter

Chomsky 1957: 18–20.

CHAPTER

5 Finite-state machines

CHAPTER PREVIEW

KEY TERMS

language processing
automata
finite state
Prolog
transducers
phonology
syntax

The previous chapters concentrated on signal-oriented methods for speech processing. In this chapter, our attention turns to language processing, starting with finite-state machines. These are rather simple computational devices applicable to various kinds of language-processing tasks. We examine a variety of examples of their use.

5.1 Some simple examples

A finite-state machine is an abstract computing device. (You will sometimes see the terms 'finite-state automaton' or 'finite-state transition network' instead, which mean the same thing as finite-state machine.) We will look at some concrete implementations of finite-state machines and the uses to which they are put in due course. In fact, it is reasonable to say that finite-state machines are nowadays the most important computational technique in spoken language processing. They are used in relating signals to word transcriptions, in morphological and syntactic processing, and even in machine translation. The uses of these abstract computing devices are many and varied, and the particular purposes that we will put them to here are representative examples, mainly applications involving speech and the structure of words. So I am going to begin by looking at some examples that are concerned with *phonotactics*, that is, the well-formedness of sequences of phonological symbols. So rather than working on signals, we are going to start off by looking at the use of these machines for processing sequences of symbols. This is at a slightly higher level of abstraction than in the previous chapters, but as we go on I shall try to make the link between symbolic representations and representations of signals. I shall show how the two levels can be integrated using a particular kind of finite-state machine. That will lay the groundwork for some other work that comes up in chapter 7, on probabilistic finite-state machines and their use in modelling speech signals.

Gazdar and Mellish (1989) give a little example of a finite-state machine: a laughing machine, that is, a machine that generates or recognizes sequences of the letter 'h' followed by 'a', repeated any number of times and terminated by an exclamation mark (figure 5.1). This machine will generate or recognize sequences such as 'ha!', 'haha!', 'hahaha!' and so on. It is not capable of recognizing any other strings, and so if we provide any other strings as the input to this machine they will not be accepted: the machine is incapable of dealing with them.

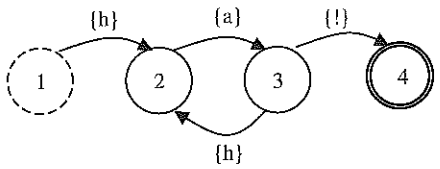


FIGURE 5.1
A laughing machine

A finite-state machine works rather like a board game in which you move a piece from one position to the next in order to get from one side of the board (the start) to the other (the end). There is a start state (state 1, marked by a dashed circle), and one or more end states (marked by a double-ringed circle): in figure 5.1 there is only one end state, state 4. The machine is allowed to move from one state to another according to the arrows, which are marked with labels (sets of symbols). The machine is used to generate strings by writing out one of the symbols on the arrow as you pass from one state to the next. Alternatively, the machine can be used to accept (i.e. recognise) strings input to the machine by checking off a symbol from the beginning of the string if it is among the set of symbols with which the arrow is labelled. The set of strings you can generate or accept by moving from the start to the end is the language defined by the machine.

Jurafsky and Martin (2000: 34) give a similar simple example. They present a finite-state machine that defines a 'sheep language'. The words of their 'sheep language' start with a *b* and then have two or more *a*s and an exclamation mark. Thus, 'baa!', 'baaa!' and 'baaaaaaaaaa!' are sheepish words, but 'ba!', 'baba!' and 'micro-organism' are not. The sheeptalk machine is reproduced in figure 5.2.

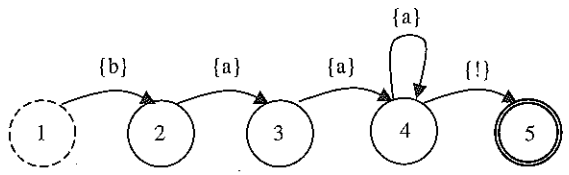


FIGURE 5.2
A sheeptalk machine (After Jurafsky, Daniel; Martin, James H., *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, 1st edition, © 2000. Reprinted by permission of Pearson Education, Inc., Upper Saddle River, NJ.)

So, a finite-state machine is an abstract computing device – an imaginary computing device, if you like, though we will see some concrete implementations shortly – consisting of (1) a set of states, (2) one of which is distinguished as the start state, (3) some of which are distinguished as end states, and (4) a set of labelled transitions between states.

5.2 A more serious example

The previous examples are instructive and easy to understand, but real spoken languages are much more complex, of course. Figure 5.3 gives an example of a machine that models (i.e. generates or recognizes) a set of monosyllabic words in a language rather like English. It is not completely right for English, but it is similar to the sequences of consonants and vowels that can occur in English monosyllables; it's an approximation to English. The labels on the transitions are sets of phoneme symbols, so this machine generates or recognizes phonemic transcriptions of monosyllabic words. (A key to the transcription system is given in table 5.1, below.)

I shall call this machine NFSA1, which stands for 'non-deterministic finite-state automaton 1'. The set of states abstractly represents the set of separate conditions the machine can be in. There are 16 states in NFSA1. When we implement this abstraction as a real, working program, the computer will actually pass through a succession of states of the program during its execution. So state 1 represents the state that the machine is in

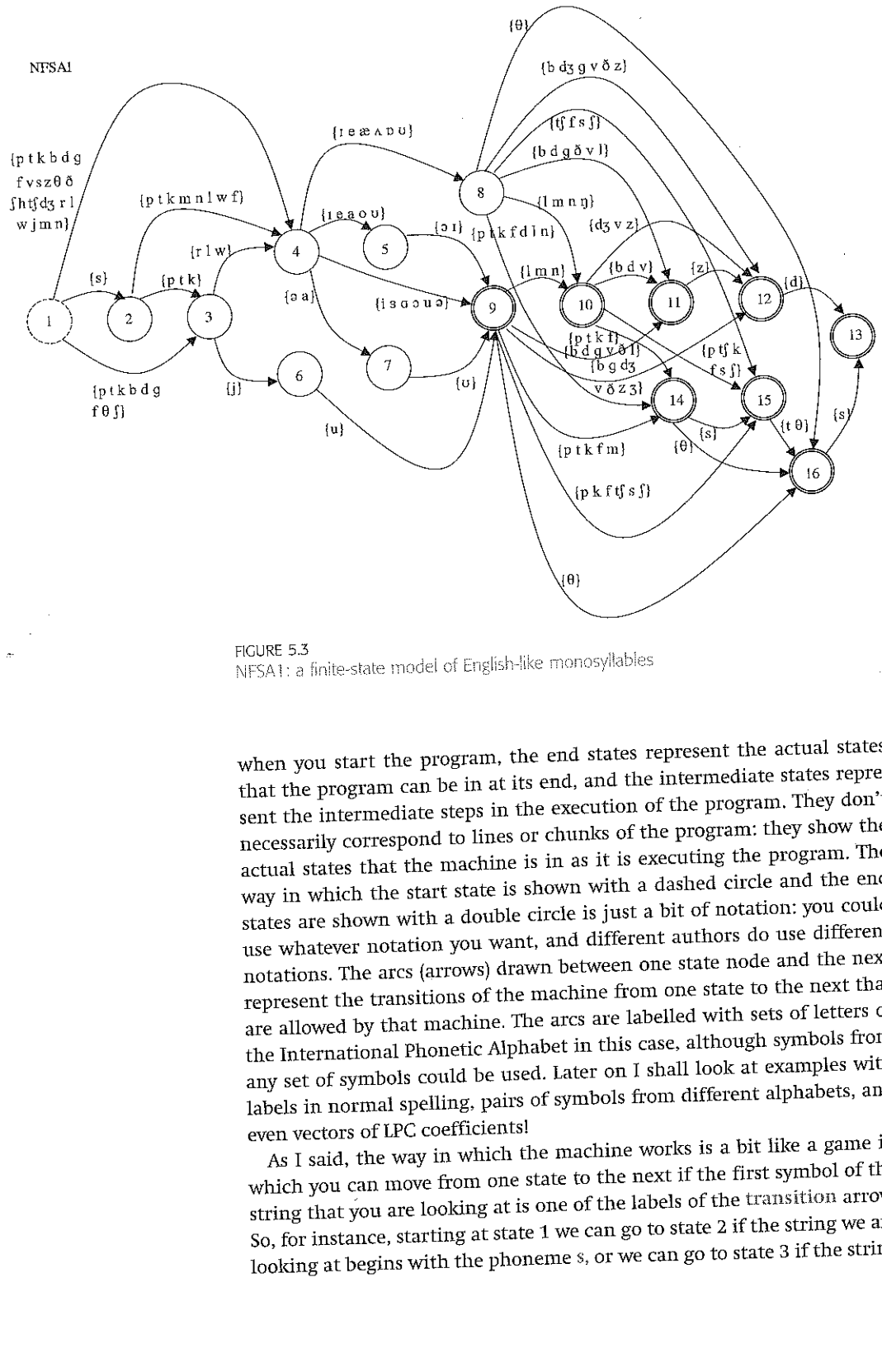


FIGURE 5.3
NFSA1: a finite-state model of English-like monosyllables

when you start the program, the end states represent the actual states that the program can be in at its end, and the intermediate states represent the intermediate steps in the execution of the program. They don't necessarily correspond to lines or chunks of the program: they show the actual states that the machine is in as it is executing the program. The way in which the start state is shown with a dashed circle and the end states are shown with a double circle is just a bit of notation: you could use whatever notation you want, and different authors do use different notations. The arcs (arrows) drawn between one state node and the next represent the transitions of the machine from one state to the next that are allowed by that machine. The arcs are labelled with sets of letters of the International Phonetic Alphabet in this case, although symbols from any set of symbols could be used. Later on I shall look at examples with labels in normal spelling, pairs of symbols from different alphabets, and even vectors of LPC coefficients!

As I said, the way in which the machine works is a bit like a game in which you can move from one state to the next if the first symbol of the string that you are looking at is one of the labels of the transition arrow. So, for instance, starting at state 1 we can go to state 2 if the string we are looking at begins with the phoneme s, or we can go to state 3 if the string

we are looking at begins with p, t, k, b, d, g, f, θ or ʃ. The rule about how we move from one state to another is: you can go from one state to another if the beginning of the string that you are processing starts with one of the symbols in the set of symbols with which the arrow is labelled. When you get to the end of the arrow, you move on past that symbol in the string. So for the next state you look at the first symbol of the *rest* of the string, after the one that we looked at in the previous transition. Let's consider the actions that this machine might go through in processing the string s, t, r, ɪ, ŋ. We start in state 1, looking at the first symbol, s. There are two routes we can take: we can either go to state 2 because s is on the arrow from 1 to 2, or we can go to state 4 because s is included in the label on the arrow from 1 to 4. We must decide which way to go – that is why the machine is called non-deterministic. (If there were only one possible path to take at every node, it would be deterministic.) For now, it doesn't matter how we decide which way to path. We could follow them in numerical order, or we could choose randomly. Let's go to state 2. The next letter of the input string is t. Well, that is OK: we can move to state 3. Then we see an r; that's OK because there is a legal transition to state 4. Then we see an ɪ and there are two ways in which we can move: we can go to state 5 or to state 8. I'm going to take the path to state 8 because it is the one that will end up working out (though a computer would not be able to see that!). In state 8 we see an ŋ next: that takes us to state 10. State 10 is an end state, and the rule about end states is that you can finish if there is nothing left of the string by the time that you reach the end state. There are no more letters left in the input string, so we can stop there. We say that the machine accepts or recognizes the string s, t, r, ɪ, ŋ. This particular machine accepts almost all the monosyllabic words in Mitton 1992, a machine-readable English dictionary, apart from a few very unusual words, mostly foreign words such as *Gdansk, Khmer, Pjerm* and *schmaltz*. It also accepts a very large number of words that are not actual, meaningful English words, but that are similar to existing words. Examples are sprenkst ('*sprenkst*'), splond ('*splawned*'), strolkt ('*strolked*'), trʌltθ ('*trultth*') and blem. It also accepts and generates a large number of words that are quite un-English, such as tlumv. Whether or not a high degree of overgeneration is acceptable depends on the application. It is often preferable to design a system that accepts a wide range of unforeseen inputs than to constrain the input so much that even inputs that ought to be acceptable are rejected.

Let's consider an input string that the machine will not accept: s, g, r, ɪ, n, t. We can get to state 2 with s, but the next symbol, g, is not one of the symbols listed in any of the transitions out of that symbol. Now what do we do? Well there are several things we could do. The first idea we shall consider is that at that point the machine just stops. It doesn't reach the end; it stops and says that you have failed. The machine doesn't recognize a string if at some point the conditions for the next symbol to be accepted aren't met. The string s, g, r, ɪ, n, t – or any other string beginning with s, g – isn't acceptable by this machine.

If you wanted NFSA1 to accept sgrɪnt or ʃnæps (*schnapps*) or gdænsk (*Gdansk*) you would have to alter the machine in some way. There is another

possibility, though, concerning what to do if the next symbol in the input isn't listed on any of the transition labels. Rather than just giving up and stopping, the machine can backtrack (go back) along the arrow that it just followed and see if there are any other routes out of the previous state that would also be acceptable. For example, when analysing s, t, r, ɪ, ŋ, if we had first decided to move from state 1 to state 4, we would not be able to go on from state 4 because the next symbol, t, is not on any of the transitions out of state 4. But that does not mean that s, t, r, ɪ, ŋ, is an unacceptable string: it's just that we were pursuing the wrong route through the machine. If we backtrack to state 1 and try a different route through the network, we can eventually accept the string. If, when you have explored every possible route, you find that you *still* can't reach an end state with no symbols left, the string isn't acceptable by *any* route through the network. At that point we say, 'no, the string we are analysing is ungrammatical (or unacceptable)'.

Student: But if you are at state 3 can you backtrack all the way to state 1?
Yes, you can; if the input was pɪt and you went from state 1 to state 3, you could backtrack from 3 to 1 in one go. You can backtrack as much as is necessary, but you must backtrack one step at a time, and you must retrace the transition you took. You can't backtrack arbitrarily far in one go; you have to backtrack to the previous state, and then you try other routes forward. If they don't work out, you can backtrack further back, and you *could* end up getting back to state 1, and be unable to proceed through the network any further. For example, if the input is sprɪmθ (*'sprimth'*), we can go from state 1 to 2, 3, 4, 8, and 10, but then we cannot go any further, as θ is not on any of the labels of the transitions out of state 10. Backtracking to 8 is no help, though if we go back to 4 we can try 5 instead. But that's no good: m is not listed on the transition out of 5. So back to 4, and no other ways forwards. Back to 3, 2, still no good. Right back to 1: we could try going from 1 to 4 instead, but it is a fool's errand: we cannot get any further. sprɪmθ is just no good.

5.3 Deterministic and non-deterministic automata

We will make use of this backtracking method in the implementation of the machine below. Note that this method is only relevant to nondeterministic FSAs: in deterministic FSAs, by definition, there is only ever one move you can make for each symbol in the input. In a deterministic finite-state machine, in each state there is only ever (at most) one transition that you can make for a given input symbol, there are never any cases like state 1 of NFSA1 where if the next symbol is s, p, t, k, b, d, g, f, θ or ʃ there are two ways you can go. Or at state 8 if the next symbol is l, there are three

ways you can go. This is a non-deterministic finite-state machine, because you sometimes can't determine the right way to go forwards. You would have to toss a coin or something, and choose one. In a *deterministic* finite-state machine there is no need for backtracking: at each step there is only one way you can go forward, for a given input string, so there is no need for backtracking.

Figure 5.4 shows a deterministic variant of NFSA1 that accepts (or generates) almost exactly the same set of strings. To make DFSA1, a few changes to NFSA1 had to be made. Some extra states had to be introduced (these are shaded) and some extra transition arcs (those with broken lines). Also, the transition labels had to be altered. Additionally, we have allowed for the possibility of empty transitions: that is, moves forwards without reading a symbol from the input string (or, equivalently, reading the empty symbol, "). For example, to read the string sʌn (*sun*), the first symbol is s, so the machine must go from state 1 to state 2. Then, the next symbol is ʌ. This is not in the transition label from state 2 to 3, nor in the transition label from 2 to 4. But the empty symbol " is in the transition label from 2 to 4, so we can go to state 4. Now ʌ is still the first symbol of the remainder of the input string: from state 4 we can go to state 8b. Then,

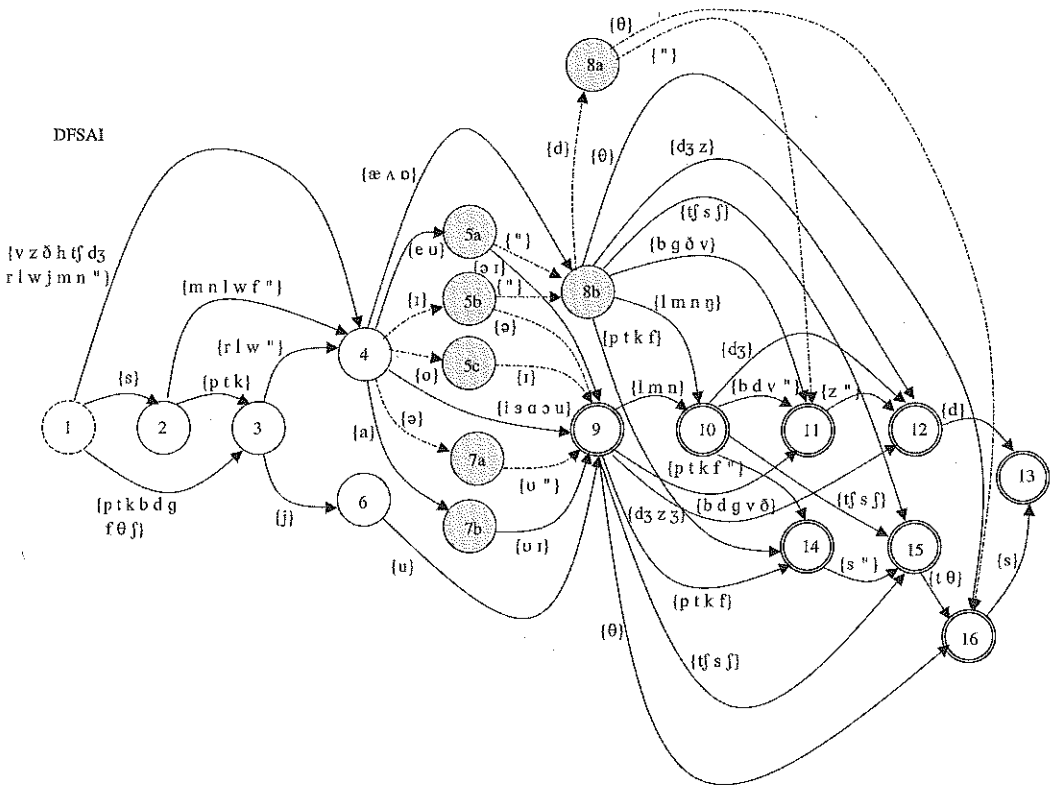


FIGURE 5.4
DFSA1: a deterministic version of NFSA1

reading n, we can go to state 10. As 10 is an end state and there are no letters left, we can stop: SAN is acceptable.

Non-deterministic FSAs are no more or less powerful than deterministic ones, in the sense that for any language that can be accepted by a non-deterministic machine, a deterministic machine can be constructed that accepts it, and vice versa. Furthermore, a non-deterministic machine can be automatically turned into an equivalent deterministic machine: Hopcroft et al. 2000 gives details.

We only use backtracking in non-deterministic finite-state machines, where at some points there might be two or more options to take. So if we take one route and it turns out not to have been right, we can backtrack and explore other possibilities. So backtracking enables us to implement a kind of parallel processing. We can't actually *do* the processing in parallel, following different routes at the same time, but we can explore various possibilities one after the other until we have exhausted all the possible paths through the network, if necessary. If we reach an end state before exploring the whole graph, then fine, the string is acceptable, and we can stop there.

5.4 Implementation in Prolog

An implementation of this example is given in listing 5.1. This particular implementation is in Prolog, a programming language that is quite easy to understand, but one that is very different from C. Some of the most salient differences are listed in the following text box, though you don't need to study this now in order to go on. A good Prolog interpreter, SWI-Prolog, is provided on the CD-ROM, and there is a helpful website that goes with it (see the companion website to this book for a link).

Some differences between Prolog and C

1. Prolog programs are not normally compiled: they are interpreted. That means that you start the Prolog interpreter, and then you can type in one definition or question at a time. Or you can put your definitions and questions in a file and run it, in which case the Prolog interpreter will go through them one at a time just as if you were typing them in.
2. There is no 'main' procedure: if your program defines various predicates (which are somewhat like C functions), you can tell the interpreter to call (i.e. run) *any* of them. This means that there are many ways of running a program. It depends on what you ask the interpreter to do.
3. There is no 'pre-processor' (a feature peculiar to C). To incorporate one program file within another (as with the C `#define filename` pre-processor command), in Prolog we talk of 'consulting' that file, expressed either as '`consult(filename).`' or '`[filename].`',

depending on the implementation of Prolog you are using. (Note that Prolog clauses end with a full stop, whereas in C they are terminated with a semicolon. The punctuation is different in other respects too. Instead of C's {. . .}, for instance, you can use (. . .) in Prolog, though it is usually not necessary.)
4. The data type of a variable does not need to be declared in advance (hooray!). In fact, a variable can be used to hold objects of various kinds. However, there are fewer numeric types than in C, and they do not relate to the actual storage of different kinds of numbers in the computer's memory. If you write anything to a file, it will be written as an ASCII description of the object: this is true even of integers and floating point numbers! The difference between variables and constants is shown typographically: variable names begin with a capital letter or the underscore symbol _. Anything else is a constant. (To make a constant that begins with a capital letter, you can put it in single quotes. Anything in single quotes is a constant.) Thus, `John` is a variable, `john` is a constant, `'John'` is a constant and `_constant` is a variable (because it begins with an underscore). `A`, `B`, `C`, `X`, `X1`, `String`, `What`, `Result` and `Anything` are typical variables. ASCII letters and numbers are constants, as are operator symbols like +, <, and the reserved words of the language such as `consult`, `is` and `name`.
5. The content of a variable is only fixed within a clause (i.e. it is local to each function). Thus, if I use the variable `A` in two clauses, there is not necessarily any connection between them.
6. Prolog is a logic programming language. One clause calls another by a process of logical inference called theorem proving. This means that you do not need to tell the interpreter the order in which the clauses in a program should be executed. Also, contradictory, inconsistent, illogical programs fail, whereas in C it is unfortunately easy to write illogical programs.

There are several good textbooks on Prolog; some titles are recommended in the 'further reading' section at the end of this chapter.

There are several main ways to represent strings in Prolog that we could use. For this implementation, we shall write a string as a list of letters, separated by commas and enclosed in square brackets. For example, we will encode pet as `[p,e,t]`. Non-roman IPA letters will need to be expressed using some kind of translation into ASCII symbols. I shall use the encoding in table 5.1, which is based on that used in Mitton 1992, with one modification: 'C' and 'J' are used instead of Mitton's 'tS' and 'dZ'. This avoids an unnecessary complication with the machine: it is not very important or necessary, though. Capital letters, like the capital T, or the capital C, and so on, should be enclosed in single quotes in

Table 5.1. An encoding of some IPA symbols into Prolog atoms and ASCII

IPA	Prolog encoding	ASCII encoding	English examples	IPA	Prolog encoding	ASCII encoding	English examples
p	p	112	**p**ot, s**p**ot	ɪ	'I'	73	p**i**t
t	t	116	**t**ot, s**t**ock	e	e	101	p**e**t, h**ea**d
k	k	107	**c**ot, so**ck**, **qu**ay, **k**ey	æ	'&'	38	p**a**t
b	b	98	**b**et	ʌ	'V'	86	p**u**tt, c**u**t, bl**oo**d
d	d	100	**d**ebt	ɒ	0	48	p**o**t
g	g	103	**g**et, **gu**ard	ʊ	'U'	85	p**u**t, f**oo**t, w**o**lf
tʃ	'C'	67	**ch**urch, e**tch**	ə	'@'	64	th**e**, **a**n
dʒ	'J'	74	**j**udge	i	i	105	f**ee**t, h**ea**t
f	f	102	**f**it, o**ff**, s**ph**ere	eɪ	e, 'I'	101, 73	f**a**te, r**ai**d, m**ay**
θ	'T'	84	**th**ick	aɪ	a, 'I'	97, 73	p**ie**, m**y**
s	s	115	**s**it, ki**ss**, **ps**ych	oɪ	o, 'I'	111, 73	t**oy**, qu**oi**t
ʃ	'S'	83	**sh**ip, qui**ch**e	u	u	117	f**oo**d, bl**ue**
h	h	104	**h**ot	ju	j, u	106, 117	n**ew**, c**ue**, **you**
v	v	118	**v**et, gi**ve**	əʊ	'@', 'U'	64, 85	g**o**, t**oe**, t**oa**d
ð	'D'	68	**th**is	aʊ	a, 'U'	97, 85	c**ow**, l**ou**d
z	z	122	**z**oo, si**ze**	ɪə	'I', '@'	73, 64	p**ier**, p**eer**, f**ear**
ʒ	'Z'	90	rou**ge**	eə	e, '@'	101, 64	p**air**, p**ear**, c**are**
m	m	109	**m**et	ɜ	3	51	v**er**se, f**ur**, f**ir**
n	n	110	**n**et, **kn**ot	ʊə	'U', '@'	85, 64	t**our**
ŋ	'N'	78	si**ng**, thi**nk**	ɔ	'O'	79	**or**, l**aw**, t**au**t
l	l	108	**l**et, te**ll**	ɑ	'A'	65	t**ar**
r	r	114	**r**ot, **wr**ite				
w	w	119	**w**et				
j	j	106	**y**et				

Prolog if they are to be used as constants, because capitals in Prolog represent variables. When we want to use capital letters as constants we have to put them in single quotes. Thus, ʃəʊ is encoded as the list ['S','@','U']. Note that 0 (zero), used to encode IPA ɒ, and 3 (three), representing IPA ɜ, do not need to be put in quotes, because numerals are constants anyway.

Listing 5.1. A Prolog implementation of NFSA1

```
/* NFSA1.PL    A nondeterministic finite-state automaton to        */
/*             recognize English-like monosyllabic phoneme strings */

accept(String):- move(s1,String).

move(State,Symbol):-
       transition(State,Symbol,end).
move(StateA,[Symbol|Rest]):-
       transition(State1,Symbol,StateB),
       move(StateB,Rest).

/* Enumerate all acceptable strings */

loop:- accept(A), write(A), nl, fail.

transition(s1,s,s2).
transition(s1,p,s3).
transition(s1,t,s3).
transition(s1,k,s3).
transition(s1,b,s3).
transition(s1,d,s3).
transition(s1,g,s3).
transition(s1,f,s3).
transition(s1,'T',s3).
transition(s1,'S',s3).
transition(s1,p,s4).
transition(s1,t,s4).
transition(s1,k,s4).
transition(s1,b,s4).
transition(s1,d,s4).
transition(s1,g,s4).
transition(s1,f,s4).
transition(s1,v,s4).
transition(s1,s,s4).
transition(s1,'T',s4).
transition(s1,'D',s4).
transition(s1,'S',s4).
transition(s1,h,s4).
transition(s1,'C',s4).
transition(s1,'J',s4).
transition(s1,r,s4).
transition(s1,l,s4).
transition(s1,w,s4).
transition(s1,j,s4).
transition(s1,m,s4).
transition(s1,n,s4).
transition(s2,p,s3).
transition(s2,t,s3).
transition(s2,k,s3).
transition(s2,p,s4).
```

```
transition(s2,t,s4).
transition(s2,k,s4).
transition(s2,m,s4).
transition(s2,n,s4).
transition(s2,l,s4).
transition(s2,w,s4).
transition(s2,f,s4).
transition(s3,r,s4).
transition(s3,l,s4).
transition(s3,w,s4).
transition(s3,j,s6).
transition(s4,'I',s5).
transition(s4,e,s5).
transition(s4,'&',s5).
transition(s4,o,s5).
transition(s4,'U',s5).
transition(s4,'@',s7).
transition(s4,a,s7).
transition(s4,'I',s8).
transition(s4,e,s8).
transition(s4,'&',s8).
transition(s4,'V',s8).
transition(s4,0,s8).
transition(s4,'U',s8).
transition(s4,3,s9).
transition(s4,'A',s9).
transition(s4,'O',s9).
transition(s4,u,s9).
transition(s4,i,s9).
transition(s4,'@',s9).
transition(s5,'@',s9).
transition(s5,'I',s9).
transition(s6,u,s9).
transition(s7,'U',s9).
transition(s8,l,s10).
transition(s8,m,s10).
transition(s8,n,s10).
transition(s8,'N',s10).
transition(s8,b,s11).
transition(s8,d,s11).
transition(s8,g,s11).
transition(s8,'D',s11).
transition(s8,v,s11).
transition(s8,l,s11).
transition(s8,b,s12).
transition(s8,'J',s12).
transition(s8,g,s12).
transition(s8,v,s12).
transition(s8,'D',s12).
transition(s8,z,s12).
transition(s8,p,s14).
transition(s8,t,s14).
```



```
transition(s8,k,s14).
transition(s8,f,s14).
transition(s8,d,s14).
transition(s8,l,s14).
transition(s8,n,s14).
transition(s8,'C',s15).
transition(s8,f,s15).
transition(s8,s,s15).
transition(s8,'S',s15).
transition(s8,'T',s16).
/* State 9 is an end state if there are no more letters left */
transition(s9,[],end).
transition(s9,l,s10).
transition(s9,m,s10).
transition(s9,n,s10).
transition(s9,b,s11).
transition(s9,d,s11).
transition(s9,g,s11).
transition(s9,v,s11).
transition(s9,'D',s11).
transition(s9,l,s11).
transition(s9,b,s12).
transition(s9,'J',s12).
transition(s9,g,s12).
transition(s9,v,s12).
transition(s9,'D',s12).
transition(s9,z,s12).
transition(s9,'Z',s12).
transition(s9,p,s14).
transition(s9,t,s14).
transition(s9,k,s14).
transition(s9,f,s14).
transition(s9,m,s14).
transition(s9,p,s15).
transition(s9,k,s15).
transition(s9,f,s15).
transition(s9,'C',s15).
transition(s9,s,s15).
transition(s9,'S',s15).
transition(s9,'T',s16).
/* State 10 is an end state if there are no more letters left */
transition(s10,[],end).
transition(s10,b,s11).
transition(s10,d,s11).
transition(s10,v,s11).
transition(s10,'J',s12).
transition(s10,v,s12).
transition(s10,z,s12).
transition(s10,p,s14).
transition(s10,t,s14).
transition(s10,k,s14).
transition(s10,f,s14).
```

```
transition(s10,p,s15).
transition(s10,k,s15).
transition(s10,'C',s15).
transition(s10,f,s15).
transition(s10,s,s15).
transition(s10,'S',s15).
/* State 11 is an end state if there are no more letters left */
transition(s11,[],end).
transition(s11,z,s12).
/* State 12 is an end state if there are no more letters left */
transition(s12,[],end).
transition(s12,d,s13).
/* State 13 is an end state if there are no more letters left */
transition(s13,[],end).
/* State 14 is an end state if there are no more letters left */
transition(s14,[],end).
transition(s14,s,s15).
transition(s14,'T',s16).
/* State 15 is an end state if there are no more letters left */
transition(s15,[],end).
transition(s15,t,s16).
transition(s15,'T',s16).
/* State 16 is an end state if there are no more letters left */
transition(s16,[],end).
transition(s16,s,s13).
```

The program is very simple, even though it is quite long. It has two parts: first, there are four statements or clauses, each of which has a left hand part and a right hand part separated by ':–', which means *if*. 'A :– B.' can be read as 'A if B', or 'A is true if B is true', or 'in order to prove A, prove B'. Note that each clause ends with a full stop. So the first statement, 'accept(String):- move(s1,String).', means 'accept a string (held in the variable String) if there is a move from state 1 (s1) that takes you through the String'. So that is a definition of acceptance or acceptability.

Then, there are two clauses in lines 6 to 10 that define a move through the automaton. The first defines the special case of a final move; the second defines non-final moves. The definition of a final move is:

```
move(State,Symbol):- transition(State,Symbol,end).
```

(Line breaks, such as those after the ': – ' in line 6 or 8 of listing 5.1, are ignored by the Prolog interpreter.) This clause can be read as 'there is a move from a State past a Symbol if there is a transition from State to the end via that Symbol'. The largest part of the program is a long list of statements about what kinds of transitions are permitted in this automaton: these statements define the entire content of the automaton NFSA1. States are represented s1 ... s16, though *any* sixteen distinct symbols would do.

Consider the transitions from state 9 to state 11. On figure 5.3 I drew a single transition arrow from node 9 to node 11 with a set of six letters {b d g v ð l} on the one arrow. But in listing 5.1 there are six different transitions from state 9 to state 11, each mentioning a single symbol:

```
transition(s9,b,s11).
transition(s9,d,s11).
transition(s9,g,s11).
transition(s9,v,s11).
transition(s9,'D',s11).
transition(s9,l,s11).
```

So that is why there are six conditions, six statements in the group of rules for transitions from state 9 to 11, any one of which is a legal transition. It is as if we interpret figure 5.5 (a) as (b), and similarly for all the other transitions with multiple labels. With only one symbol per transition arrow, we can dispense with set braces.

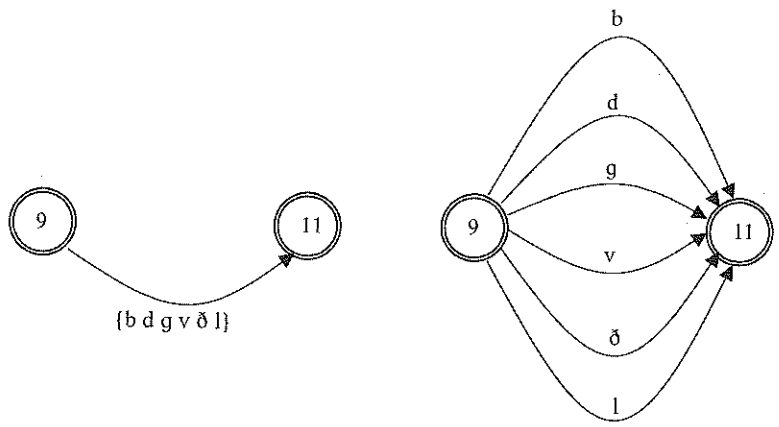


FIGURE 5.5
Interpretation of multiple labels

Digression

Alternatively, we could use the predicate member, which is built in to many versions of Prolog, to check whether the symbol is in the set of symbols on a transition arrow. For example, instead of the following six lines:

```
transition(s9,p,s15).
transition(s9,k,s15).
transition(s9,f,s15).
transition(s9,'C',s15).
transition(s9,s,s15).
transition(s9,'S',s15).
```

we could write just:

```
transition(s9,X,s15):- member(X,[p,k,f,'C',s,'S']).
```

If it is not built in to your version of Prolog, the following definition of `member` is standard:

```
member(H,[H|_]).
member(X,[_|T]):- member(X,T).
```

However, this approach makes the code run a little more slowly, and does not generalize as easily to other varieties of finite-state automata, such as those we will consider below.

Referring to figure 5.3, the following are examples of possibly final transitions: (1) from s6 to s9 via `u`, (2) from s10 to s11 via `b`, and (3) from s8 to s15 via `t∫`. In the program, however, these are treated like non-final transitions, because whether a state is final or not depends not only on its own status, but also on the fact that there are no more symbols left in the string. The following clauses sanction the three transitions mentioned above:

```
transition(s6,u,s9).
transition(s10,b,s11).
transition(s8,'C',s15).
```

The fact that states 9, 11 and 15 (to name but three) are final states if there are no more letters left is encoded by the following clauses:

```
transition(s9,[],end).
transition(s11,[],end).
transition(s15,[],end).
```

Effectively, it is as if we had defined an additional state, called `end`, as in figure 5.6. Transitions from s9, s11, s15 and others to state `end` are permitted if all that remains of the string is nothing.

The second statement in the definition of `move` defines non-final moves:

```
move(StateA,[Symbol|Rest]):-
        transition(StateA,Symbol,StateB),
        move(StateB,Rest).
```

It is a recursive definition, because it includes `move` again. It can be read as follows: 'there is a `move` from any state, `StateA` through a string that starts with a `Symbol` and continues with the `Rest` of the string *if*: (1) there is a transition from `StateA` to some `StateB` via that `Symbol`, and (2) there is a `move` from `StateB` through the `Rest` of the string'. Briefly, that means you can go from state A to the end, if you can go from state A to state B and then from state B to the end. Note that the second argument of the first mention of `move` is a list, `[Symbol|Rest]`. There are two notations for lists (of e.g. characters) in Prolog. We have already

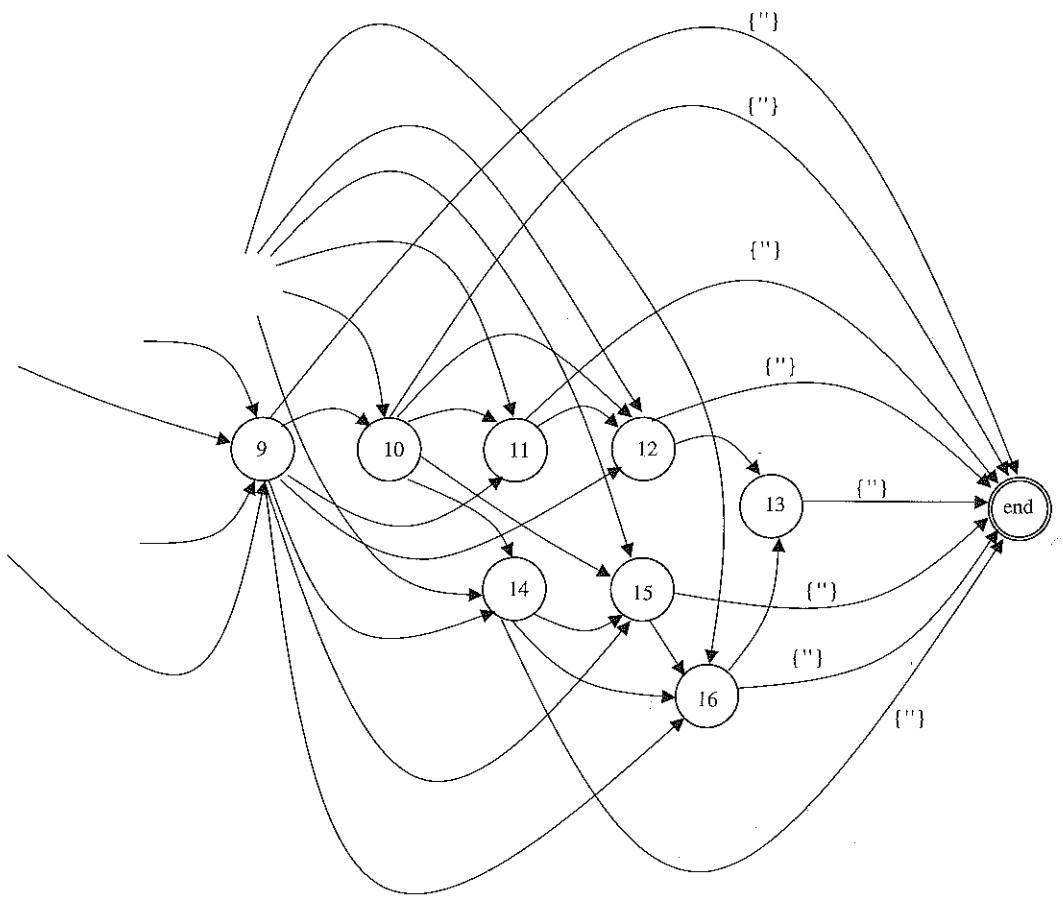


FIGURE 5.6
Treatment of final states in the Prolog implementation of NFSA1

seen the notation in which you explicitly state the items in the list, separated by commas, for example `[p,e,t]`. We can also describe a list by its first element(s) (the head) and the rest (the tail). The tail of a list is a list containing all the items in the list apart from the head. Thus, the head of `[p,e,t]` is `p` (not `[p]`, note) and its tail is `[e,t]`. Another way of writing a list uses the symbol '|' to separate the head from the tail. For example, instead of `[p,e,t]` we could write `[p|[e,t]]`. (We could also write it as `[p,e|[t]]` or even `[p,e,t|[]]`.)

The fourth clause is a loop to generate all the strings that the automaton accepts. I will discuss it further below.

Student: Can I just ask about the brackets? I am not quite sure what is the significance of ordinary parentheses and square brackets.
Parentheses are only used in this program to show the arguments of a predicate (i.e. function). Square brackets indicate a *list* of objects: in this

case we are using lists of letters to represent strings. The vertical bar is a notation that separates the beginning of a list from the rest of a list.

Student: So in the second clause of move, *the first symbol of the list is salient, and the rest is sort of unspecified?*
Yes, the `Rest` is dealt with in later steps. In each state we are only concerned with the first symbol of the part of the string we are working on. The rest of it is still to be processed. We can't look ahead and process letters in the string that are further on: we need to set them aside for processing by the later parts of the network.

Student: I mean, at state 1, at the beginning, if the letter is for instance m *(let's say the work is* milk*), at this state if* m *is accepted, the rest is going to be processed in state 4?*
And the `Rest` will be `['I',l,k]`, in state 4. Then when we look at the first symbol, `'I'`, we can go to state 5 or 8. Now the `Rest` is just `[l,k]`. Suppose we go to state 5 . . .

Student: It is illegal.
Yes, it is illegal, because in state 5 there are no transitions that read the letter l. The rest of the string must begin with `'@'` or `'I'`. The machine will have to backtrack. But if the input string had been `[m,'I','@']` ('mere'), the rest of the string from state 5 would be `['@']`, and so in that case we could go to state 9, which is a possible end state.

So as we go through the string we can read letters off the beginning of the list, one by one, and as we do so we move from one state to the next. There are several end states, and if in those states the string that is passed to this state has no more letters left, it is the empty list, that is, an acceptable final move. And because that is an acceptable final move that isn't conditional upon any other moves, the program will finish at that point: it will terminate successfully.

Student: What does it actually do if you give it an illegal string. Suppose you get to state 9 and the first symbol in Rest is an 'I'?
Since Prolog works non-deterministically and this is a non-deterministic automaton, it will backtrack and attempt to find other routes through the network. If it reaches a point at which every route has been tried that it can legally get to, but it can get no further than that, the program will fail at that point. Each time Prolog processes a clause, it actually returns the answer either 'yes' or 'no', meaning 'yes, that is provable', or 'no, I can't prove that with these rules'. If it gets to an end state with no letters left, it has proved that the input string is acceptable, so the result will be 'yes'. But if after thrashing around and backtracking here, there and everywhere, and exploring the network without finding any way through it, it will say 'no'. Those are the two possible results of the predicate `accept`.

5.5 Prolog's processing strategy and the treatment of variables

I've been talking about this machine as an acceptor of strings: you put a string in, it follows some transitions through the machine and ends up with a decision about whether the string is acceptable or not. Interpreting the program as an acceptor, a recognizer, is actually only one view of the program's behaviour. Given a string, the machine will behave as an acceptor. But suppose we provide not a string but no information, just a variable, as the input to the machine. What it will then do is work its way through the transition network, and since a variable will match *any* of the symbols on the transition network, it will be able to trace any path through the network that it chooses. In doing so it will have followed a particular choice of letters on the arrows.

To understand this, let's think of an example in a different domain. In algebra you have symbols that stand in place of a whole set of other symbols that could have occurred in those places: we use x and y instead of actual numbers, for instance. In an expression like $x + y = 10$, x can be 1 and y 9, or x can be 2 and y 8. Or x could be 1024 and y, -1014. x can be any number: it stands for a range of possibilities. We can do that for strings as well: you can have variables like x and y representing not numbers but possible letters, without specifying which particular letters. You can have variables for any kind of object, representing a lack of any more specific information about that kind of object. In listing 5.1, the definition of a move does not refer to any particular symbol, but describes the string in question using the variables `Symbol`, to refer to the first letter of the string, and `Rest`. Within a clause, the value of a variable is the same on each mention, as illustrated by the lines in Figure 5.7. In this figure, solid lines are used to indicate that two instances of a variable have the same value, and dashed lines are used to indicate that two variables with different names but in the same position (e.g. the first and second arguments of the predicate `move`) have the same values, too.

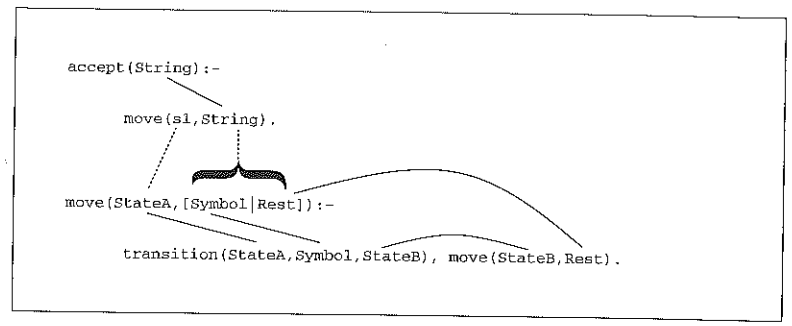


FIGURE 5.7
Sharing of values of variables within a clause is determined by name (solid lines) and between clauses is determined by argument position (dashed lines)

Now, if we ask the Prolog interpreter to prove '`accept([s,p,u,n]).`', it will give the variable `String` the value `[s,p,u,n]`: this value is passed on to the `move` predicate, and the interpreter next attempts to prove:

```
move(s1,[s,p,u,n]).
```

Recall that there are two clauses in the definition of `move`: the interpreter tries each one in turn. It tries the first clause first:

```
move(State,Symbol):- transition(State,Symbol,end).
```

It can do this by assigning the constant `s1` to the variable `State` and the list `[s,p,u,n]` to `Symbol`, leading it next to attempt to prove:

```
    transition(State,Symbol,end).
i.e. transition(s1,[s,p,u,n],end).
```

The attempt will be fruitless, however, as nowhere in all the many definitions of the legal transitions of this machine is there one from `s1` to `end`. Nor are there any transitions listed with more than one letter as the second argument. Thus, this clause cannot be proved, and the interpreter must backtrack.

But there is a second clause to the definition of a move, and this is now tried out. To prove '`move(StateA,[Symbol|Rest])`' according to the second clause, the interpreter must first prove '`transition(StateA, Symbol,StateB)`', and then prove '`move (State B, Rest)`.' To do this, `s1` must be assigned to `StateA`, and `[s,p,u,n]` to `[Symbol|Rest]`. Bearing in mind what I said earlier about the structure of lists, that means that `Symbol` is instantiated to `s` and `Rest` is instantiated to `[p,u,n]`. So, the interpreter must prove '`transition(s1, s,StateB)`'. Note that `StateB` is a variable, so it can in principle be set to anything (because Prolog variables are not limited to a specific data type, as C variables are). Among the many definitions of the transitions, there are two with `s1` as the first argument and s as the second argument:

```
transition(s1,s,s2).
transition(s1,s,s4).
```

Once again, these are considered in turn. The first means that `StateB` must be set to `s2`. OK so far. That means that the interpreter must then attempt to prove '`move(StateB,Rest)`', that is, '`move(s2,[p,u,n])`'.

If we repeat this logic over again, we will soon come round to attempting to prove that '`transition(s2,p,StateB)`', and thence '`move(s3,[u,n])`'. However, the latter will not pan out, as there are no transitions out of s3 with u as the first symbol. So we backtrack to the second possibility for proving '`transition(s2,p,StateB)`', which is '`transition(s2,p,s4)`', which leads on to '`move(s4,[u,n])`'. This is more profitable, as the program contains the statement '`transition (s4,u,s9)`'. So, on we go with '`move(s9,[n])`' as our new goal. Because of the definition '`transition(s9,n,s10)`', we next attempt the goal '`move(s10,[])`'. We have now used up all the letters in the input string as we moved from one state to the next. Fortunately, state 10 is an end state: the program contains the clause '`transition(s10,[],end)`'. This matches the first clause of `move`, and does not ask for anything else to be proved. The search for a proof is now complete, as the following text box spells out.

Prolog's (eventual) proof of `accept([s,p,u,n])`.

1. `transition(s10,[],end)`.
2. Therefore, `move(s10,[])`.
3. `transition(s9,n,s10)`.
4. Therefore, `move(s9,[n])`.
5. `transition(s4,u,s9)`.
6. Therefore, `move(s4,[u,n])`.
7. `transition(s2,p,s4)`.
8. Therefore, `move(s2,[p,u,n])`.
9. `transition(s1,s,s2)`.
10. Therefore, `move(s1,[s,p,u,n])`.
11. Therefore, `accept([s,p,u,n])`. Q. E. D.

Exercise 5.1. Trying it out

If you have SWI-Prolog installed on your computer, you should be able to start the Prolog interpreter in Windows and consult (i.e. load) `nfsa1.pl` just by double-clicking on its icon, in whatever folder you have put it. (Otherwise, double-clicking on the `nfsa1.pl` file icon may cause Windows to present the 'Open With' dialogue box, which says: 'Click the program you want to use to open the file "Nfsa1.pl". ...'. You'll have to respond by selecting your Prolog interpreter.)

Alternatively, you can launch your Prolog interpreter (for example, by double-clicking on its icon, or clicking on the Start menu, selecting 'Programs ▶', and then your Prolog interpreter's icon. If you don't see one, you probably haven't got one, and you will need to install one before going any further.

If you launch Prolog in this way, you'll need to consult `nfsa1.pl` manually, by clicking on the Prolog window and typing:

```
[nfsa1].
```

after the Prolog prompt, '?- '. If that doesn't work, you may have to give the full file name, e.g.

```
['nfsa1.pl'].
```

or the full directory name. (In SWI-Prolog, the Microsoft convention of using '\' in pathnames is not observed: instead, you must use '/'.) Thus, to load C:\SLP\`nfsa1.pl`, you type:

```
['C:/SLP/nfsa1.pl'].
```

Or you may have to use the built-in predicate `consult`:

```
consult('nfsa1.pl').
```

Some versions of Prolog for Windows (including SWI-Prolog) give you pull down menus for dealing with files. Whatever brand of Prolog interpreter you have, and however you consult the program file, you should then be able to try it out. Try the query:

```
accept([s,t,r,'I','N']).
```

Prolog should reply just:

```
Yes
?-
```

and is now ready for you to type in the next query.

Exercise 5.2
See if it will accept the pseudo-English words mentioned in section 5.2: sprɳkst, splɔnd, strɔlkt, traɪtθ and blem. What about the un-English word tloimv? What about sgrɪnt and sprɪmθ?
Hint: Remember to encode ŋ, ɔ, ʌ, θ, u, and ɪ using capital letters in single quotes, i.e. 'N', 'O', 'V', 'T','U' and 'I' respectively (see table 5.1).
Tip: In SWI-Prolog, you can recall earlier queries by using the 'up arrow' key. You can then modify an earlier query using the 'left arrow' and 'right arrow' keys, delete and backspace, and the other keys on the keyboard. Prolog will not process your query until you hit the Enter/Return key.

Exercise 5.3
If you forget to put the quotes round a capital letter, and type e.g.

```
accept([s,p,l,O,n,d]).
```

instead of:

```
accept([s,p,l,'O',n,d]).
```

Prolog will take the O to be a variable name. What happens? After it gives its response, press the semicolon key. What happens? Press it 11 more times, or until Prolog responds No. What's going on?

5.6 Generating strings

If we don't provide a string to the automaton, but just provide a variable, a variable will match *any* list of letters. So if we give a variable as the argument of `accept`, the machine can take any path it likes through the network from beginning to end. Since all we have given it to work on is a variable, it will always be able to get from any of the start states to any of the end states.

Instead of invoking this program by entering

```
accept([s,t,r,'I','N']).
```

where we give a particular string and ask 'is this sequence of letters acceptable?', we can enter

```
accept(X).
```

which means 'What X is acceptable?' Because X is unspecified, the machine will be able to follow any path from the start state to the end state as an instance of X. In following such a path, it will follow a particular set of transitions, each labelled with a particular symbol. So when the machine gets to the end state it will have picked some list of letters, and that list will be assigned to X. The first string it will accept in this way is [`s, p, r, 'I', @`], in fact. That is just the first path that it happens to follow through the

network, because the first transition in the program out of s1 is to s2 via s, the first transition out of s2 is to s3 via p, the first transition out of s3 is to r via s4, and so on. In this way, the automaton can *generate* a string.

Exercise 5.4
Try it.

However, getting *one* result in this way perhaps isn't very satisfactory. For instance, we might want to generate *all* the strings acceptable to the automaton. So after the Prolog interpreter gives an answer, you can type ';', and that tells it to backtrack and consider another possible outcome. When it is forced to backtrack it goes back and generates another answer. First, the last transition will be reconsidered, which was the transition from state 9 to the end:

```
transition(s9,[],end).
```

The next transition out of state 9 in the program listing is:

```
transition(s9,[l],s10).
```

so the next solution it generates is:

```
X = [s, p, r, 'I', @, l]
```

Then if you enter ';' again it will go off and find another solution, and another. If you want to determine the full set of strings that an automaton generates you could be typing semi-colons all night, as in fact it generates 564498 strings. (Yes, I *have* generated them all.) So I have also provided a little predicate called `loop` (see listing 5.1 for the definition). If you enter

```
loop.
```

that calls:

```
accept(A), write(A), nl, fail.
```

meaning, 'accept a variable, type the contents of that variable out, start a new line and then fail'. The enforced failure at the end makes Prolog backtrack, and it will carry on looking for alternative solutions. There are no alternative solutions to `write(A)` or `nl`, but there are many, many alternative solutions to `accept(A)`. So Prolog will backtrack through the whole search space and it will generate all acceptable strings. (You may have to use Ctl-C to interrupt the program, or even close the Prolog window!)

Consequently, the program `nsfa1.pl`, like the abstract finite-state machine in figure 5.3, is completely non-committal about whether it is an accepting device or a generating device. It depends on what queries you give the interpreter to work on.

You can concoct slightly more exotic queries (as in exercise 5.3) where you give an incompletely specified string, such as `[s,t,r,X,'N']` and the solutions to that will provide various values of X. You can get it to generate either a single value of X that is phonotactically acceptable, or if you were to keep typing semicolons or write a bit of code like the `loop` predicate, you could get it to generate all possible values of X, insofar as the string is well formed according to the machine.

5.7 Three possibly useful applications of that idea

1. Alliteration. Suppose we want to find words that begin with the same consonant cluster as 'scrunch'. The following query will do the trick:

```
X = [s,k,r|_], accept(X).
```

The underscore symbol means 'the anonymous variable', that is, a variable whose contents we are uninterested in examining. (What happens if we just ask `'accept([s,k,r|_]).'`?)

2. Riming. This is a bit difficult. Suppose we want a rime for 'munch'. The following will give some results, but not all:

```
X = [_,'V',n,'C'], accept(X).
```

Repeatedly replying to each solution by typing semicolons will generate every solution with one letter before '–unch'. But it will not yield, say, 'scrunch'. The best ways to get rimes with more than one letter before '–unch' is simply to make two further queries:

```
X = [_,_,'V',n,'C'], accept(X).
```

and

```
X = [_,_,_,'V',n,'C'], accept(X).
```

(There is an alternative that will generate all the rimes from one query, but it is very inefficient and slow.)

3. Palindromes. In the previous examples, a *template* list was constructed and submitted to `accept`. Other templates are possible. For instance, we can exploit the fact that every mention of a variable shares a value within a clause to make templates that are symmetrical. Words with this structure are palindromes. (In this case, they are phonemic palindromes, because NFSA1 cannot spell.) The only patterns of palindromes found in monosyllabic words are:

```
[C,V,C],      [C1,C2,V,C2,C1],          [C1,C2,C3,V,C3,C2,C1],
[C,V,V,C],    [C1,C2,V,V,C2,C1], and    [C1,C2,C3,V,V,C3,C2,C1].
```

The first solution to

```
X = [C1,C2,C3,V,C3,C2,C1], accept(X).
```

is

```
X = [s, p, l, 'I', l, p, s]
```

All the solutions with two V's in the middle are uninteresting, as the only identical sequence of V's acceptable to NFSA1 is 'I','I', which is not really English.

Student: Is X allowed to be zero?

Do you mean an empty symbol? Not in NFSA1, because there are no empty symbols in the definitions of transitions, except in end states, and all strings end in those empty strings.

5.8 Another approach to describing finite-state machines

We have been talking about these machines on several different levels. I started off with pictures of networks with state nodes, arrows and labels. We should not confuse a picture of a network with the abstract machine that it depicts. The machine itself is not a network, but an abstract computing device. I shall not get into a discussion about it, but there is a huge literature on the algebraic structure and properties of abstract automata. (If you are really keen, see, e.g., Hopcroft et al. 2000.) For example, you may come across definitions like this:

A finite-state automaton A is a quintuple $(Q, \Sigma, q_0, F, \delta)$ where Q is a finite set of states $q_0, q_1, \ldots, q_N$, Σ is a finite alphabet of input symbols, q_0 is the start state, F is the set of final states, $F \subseteq Q$, and $\delta \subseteq Q \times \Sigma \times Q$, the transition function.

That definition should be taken outside and shot.

So pictures and algebraic structures are two levels of representation. A third level of representation is that of particular computer programs written in particular programming languages, the implementations of a finite-state machine. There are many ways to implement a specific, abstract finite-state machine, and many programming languages in which you could do it.

Table 5.2 gives a fourth representation of finite-state machine, a symbol–state table, or state transition table, as they are sometimes known. A state transition table is a two-dimensional table with the list of the symbols that may occur in the alphabet that the machine is capable of accepting listed along the top row, and the numbers of the states listed down the left-hand column. The entries in the table say what state to go to if the symbol at the head of a column occurs in the input. We start at the start state, state 1, which is the first line. If the first symbol in the string we are processing is a 'b', we look in the column of state numbers underneath 'b'. In row 1, in the 'b' column is the number 3, which means we then go to state 3. If the first letter of the input was 'v', we would go to state 4, and so on. Suppose it is 'v' and we go to state 4. We now look at the entries in line 4. If the next symbol is 'i' we must go to state 9. If the next symbol is 'h', we look at row 9, column 'h'. That cell contains 0, meaning that that is an illegal transition: 'vih' is an illegal string! So the entries in the table encode the transitions from one state to another state when viewing a particular symbol in the input. So the 'from' state number is the line number of the table, the

Table 5.2: Symbol–state table of DFSA1 (see figure 5.4). Final states are in bold italics.

	1	2	3	4	5a	5b	5c	6	7a	7b	8a	8b	***9***	***10***	***11***	***12***	***14***	***15***	***16***
P	3	3	0	0			0	0					14	14					0
k	3	3	0	0			0	0					14	14					0
b	3	3	0	0			0	0					11	11					0
d	3	3	0	0			0	0					11	11					0
g	3	3	0	0			0	0					11	0					0
U	4	4	0	0			0	0					15	15					0
dz	4	4	0	0			0	0					12	12					0
F	3	3	4	0			0	0					14	14					0
θ	3	3	0	0			0	0					16	16					0
S	2	2	0	0			0	0					15	15					0
J	3	3	0	0			0	0					15	15					0
Y	4	4	0	0			0	0					0	0					0
ð	4	4	0	0			0	0					11	12					0
Z	4	4	0	0			0	0					12	0					0
4	4	0	4	4	0	4	4	4	4	4	4	4	9	5b	5a	8b	7b	7a	9	5a	5c	8b	8b	9
3	0	0	4	4	0	4	0	4	0	4	0	4	0	9	5b	5a	8b	7b	7a	9	9	5a	5c	8b	8b	9
2	0	0	4	4	0	4	0	4	0	4	0	4	0	9	5b	5a	8b	7b	7a	9	9	5a	5c	8b	8b	9
1	4	0	4	4	0	4	4	4	4	4	4	4	9	5b	5a	8b	7b	7a	9	9	9	5a	5c	8b	8b	9

'next' state numbers are entries in the table, and each column corresponds to the symbol that must be read in order to make a transition from one state to the next. Also, note that I have indicated which states are final, by writing those state numbers in bold italics. As you will see, I haven't filled in all the numbers for the entire network. I leave that as an exercise for anyone who wants to implement DFSA1 in this way for themselves. Although some lines are complete, note that line 6 actually only has one legal state transition in it, which means 'if you see a "u", and you are in state 6, you can go to state 9'. So another approach to the implementation of finite-state machines is to have such a table in a computer file, and then all you need is a little program that moves from one state to the next according to the entries in the table and the next symbol in the string. The program obviously would not care about what that table represents: it will work with any such table, so that is a nice general-purpose implementation of finite-state machines. The machine follows the moves given to it by the table, and the table could be changed in order to model different languages, or to use different symbols. (To represent non-deterministic machines in this way, it must be possible to have more than one number in each cell. Then, a way of picking one of them must be added, as well as a way of keeping track of which one was selected, so that on backtracking the other choices may be pursued.)

Well, we could stop at this point because those are the main things to learn about finite-state machines. But for the rest of the chapter I shall cover some other possibilities and some particular ideas for extensions, applications and so on.

5.9 Self-loops

In the automata we have looked at so far there aren't any self-loops. A self-loop is a transition from a state to itself. In figure 5.2 there is a loop from state 4 to itself. If the machine is in state 4 and sees the letter *a*, instead of going to state 5 it stays in state 4. The letter *a* is acceptable at that state but it doesn't advance the state of the machine. You might ask 'what is the use of that?' In the context of the previous discussion it may not appear to be very useful, but there is a general purpose device called a searcher, a finite-state machine with just two states (figure 5.8). A searcher is a machine that looks for a particular symbol, or perhaps a particular short sequence of symbols. Suppose, for instance, we have a string and it is however long it is, and want to look to see if it contains a '£' sign. We might even have a file of such strings: we might be searching through the files on your disk, looking for all of the files that have something to do with money. (A file is just a long string of characters.) What we need is a searcher. State 1 is labelled 'Σ–{£}': Σ is the alphabet of the machine (the entire ASCII character set, for instance), and '–' means set difference, so 'Σ–{£}' means 'all symbols apart from £'. In state 1, therefore, if the first symbol of the input *isn't* a £ sign, the machine stays in that state. It then

examines the next symbol in the input. It can go to the second state if the first symbol is what it is that you are looking for, the pound sign. After it has found a £ sign we don't care what else it sees, so that could be anything: it might contain the £ sign, or it might not contain the £ sign, it could be any symbol in the alphabet, so state 2 is labelled 'Σ'. (Since 'Σ' is being used as the name of a set, rather than as a symbol that might occur in the machine's input, it is not enclosed in set braces, {...}.) State 2 is an end state.

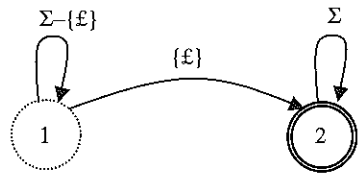


FIGURE 5.8
A machine that searches for money

The machine will only get to state 2 if the input contains the £ sign. Therefore, the machine will only accept strings containing £.

Exercise 5.5
Think of an easy way to extend the machine in figure 5.8 in order to make it search for money in many currencies, e.g. dollars, yen or euros. What problem might arise in searching for prices in pence?

grep

Readers who are familiar with the Unix operating system are probably aware of *grep*, a program that searches through one or more files for a specific string or search pattern, using a finite-state method. It's like a more general version of the method presented here for searching for a currency symbol. MS-DOS has a similar but more restricted command, called 'find'. In the MS-DOS prompt window (i.e. command prompt window, in more recent versions of Windows), typing `FIND 'string' filename` causes the stated file(s) to be searched for all instances of the stated string.

Searchers can also be used in finite-state machines that model the application of phonological rules. When you apply a phonological rule of the form $cad \rightarrow cbd$ (i.e. $a \rightarrow b \,/\, c\_d$) to a string x you have to see whether the expression on the left-hand side of the rule, *cad*, matches the string x that you are applying the rule to. You can use a searcher for that (figure 5.9). Note that in state 2, if the next symbol is not *a*, you go back to the beginning of the search again. That is, a *c* that is not followed by an *a* does not get you very far. Nor does *ca* if *d* does not immediately follow (state 3).

So a searcher implements a pattern-matching operation. The larger machines that we looked at earlier on are also pattern-matching machines in that the set of strings that they will accept matches the patterns of well-formed syllable phonotactics of this English-like language. Weird and

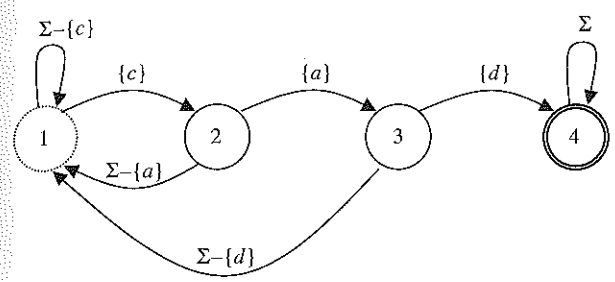


FIGURE 5.9
A schema for machines that search for phonological rule environments of the form *cad*. (For each rule, specific symbols must be substituted in place *c*, *a* and *d*.)

wonderful strings that violate what is acceptable by NFSA1 or DFSA1 are not recognized or accepted: they fail to match the machine's patterns.

5.10 Finite-state transducers (FSTs)

There is an interesting and useful generalization of finite-state machines in which the transition labels consist not just of single symbols, but of pairs of symbols. A finite-state machine of this kind is called a finite-state transducer, and works with two strings at a time. A transition is acceptable if one element of the label is the first symbol of one string and the other element of the label is the first symbol of the other string. In this way, correspondences between the symbols of one string and symbols of the other string can be related to another in sequence. The two strings that are processed by a finite-state transducer could have various interpretations or uses. For instance we might regard one of the strings as an input string, and the other as an output. Alternatively, we could regard two strings as the input, and the machine would then compute an alignment or set of correspondences between the two strings.

Let's look at some specific examples of this to show the use. First, we will consider a machine that relates orthographic representations (i.e. words written in normal spelling) to their phonemic transcriptions. It uses paired transition labels, joined by a colon, such as ph:f, th:θ, th:ð, sh:ʃ, c:k, ck:k, oo:ʊ, oo:u and x:ks. A symbol written by itself, for example s, abbreviates the same symbol on both sides of the relation, for example s:s. On either side of the semi-colon there may be single symbols or short sequences of symbols, to allow for the fact that two orthographic units can map onto one phoneme (e.g. sh:ʃ), or one orthographic unit may map onto two phonemes (notably x:ks). Figure 5.10 illustrates such a transducer, NFST1. It is based on NFSA1, and works with English-like monosyllables.

A Prolog implementation of NFST1 is given in the file `nfst1.pl`. The beginning of that program is given in listing 5.2. There are a few differences between `nfst1.pl` and `nfsa1.pl`. First, the definitions of `accept`, `move` and `loop` are altered so that they work with two strings simultaneously: an orthographic string and a phonemic string. Second, the representation of strings is different from that in listing 5.1. Instead of strings of constants, such as `[s,t,r,'I','N']`, this program represents a string as a list of ASCII character codes. For example, `"S@U"`

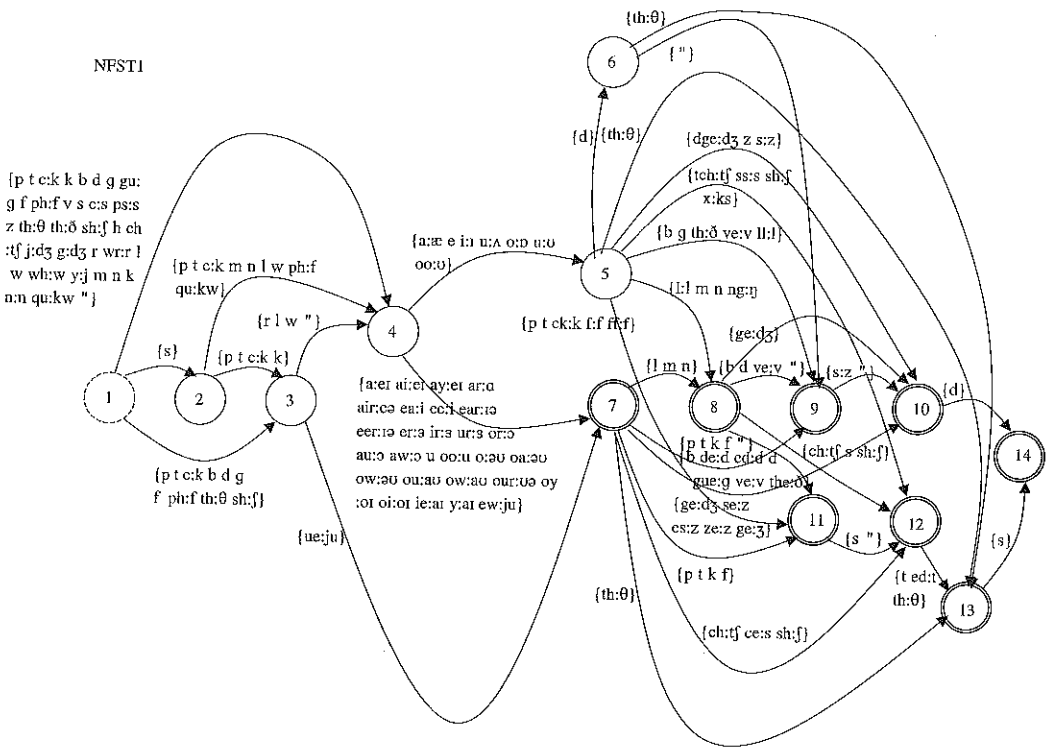


FIGURE 5.10
NFST1: a non-deterministic finite-state transducer for computing grapheme–phoneme relations in English monosyllables

(i.e. ʃəʊ) is encoded as the list `[83, 64, 85]`, not `['S','@','U']`. (This has several implementational advantages that I shall not discuss here.) Fortunately, this encoding is not as opaque as it may at first appear, because Prolog provides two mechanisms that help us use it easily. First, there is a special notation for lists of ASCII characters: a string of letters enclosed in double quotes is automatically represented as a list of ASCII codes by Prolog. Thus, if we write `"S@U"`, it will be automatically translated by Prolog into `[83, 64, 85]`. The empty string, `""`, is translated to the empty list `[]`. Second, the built-in predicate `name` converts (either way) between Prolog constants and lists of ASCII codes. Thus, the query:

```
?- name(X, [83, 64, 85]).
```

yields the answer:

```
X = 'S@U'
```

and

```
?- name('sgrint',X).
```

gives:

```
X = [115, 103, 114, 105, 110, 116]
```


This predicate is used to make the ASCII strings that may be generated by the machine more readable, so they can be printed out to the screen or to a file. (Note that `name` is now called `atom-codes` in the international standard definition of Prolog.)

Listing 5.2. Part of a Prolog implementation of NFST1

```
/* NFST1.PL    A nondeterministic finite-state transducer          */
/*             to relate English-like phoneme strings to spellings */

accept(OrthString,PhonString):-
       move(s1,OrthString,PhonString,[],[]),
       name(Orth,OrthString), name(Phon,PhonString),
       write(Orth), write(' '), write(Phon), nl.

move(State1,Orth,Phon,[],[]):-
       transition(State1,Orth:Phon,end).
move(State1,Orth,Phon,OrthRem,PhonRem):-
       transition(State1,OrthSym:PhonSym,State2),
       append(OrthSym,OrthRest,Orth),
       append(PhonSym,PhonRest,Phon),
       move(State2,OrthRest,PhonRest,OrthRem,PhonRem).

/* Enumerate all acceptable strings */

loop:- accept(A,B), fail.

transition(s1,"s":"s",s2).
transition(s1,"p":"p",s3).
transition(s1,"t":"t",s3).
transition(s1,"c":"k",s3).
transition(s1,"b":"b",s3).
transition(s1,"d":"d",s3).
transition(s1,"g":"g",s3).
transition(s1,"f":"f",s3).
transition(s1,"ph":"f",s3).
transition(s1,"th":"T",s3).
transition(s1,"sh":"S",s3).
```

The treatment of empty transitions is a little different, as there are two circumstances to consider. First there is the case in which the state is an end state, and there are no more letters left. For example:

```
transition(s8,"":"",end).
```

Second, there is the case of an empty transition to the next state, though there are more letters remaining (a mechanism that helps to keep the machine a bit simpler). For example:

```
transition(s8,"":"",s9).
```

Now suppose that you provide only one of the input strings to `accept`, the one that corresponds to the symbols before the colon, a graphemic input string. For example:

```
?- accept("squeak",_).
```

The graphemic input string variable `OrthString` is instantiated with a particular sequence of symbols (in this case `"squeak"`, i.e. `[115, 107, 119, 105, 107]`), and `PhonString` is just an uninstantiated variable. Provided that `OrthString` is graphemically well formed (according to the definition of the machine), as the machine progresses through the network from one state to the next, it will successfully read off the graphemic side of the transition labels. As it does so it also can generate a record of the correspondences between the graphemes and the phonemes as it goes along. The sequence on the phonemic side is generated by the machine at the end as the result (just as was the case when we non-deterministically generated strings in section 5.6). The machine provides a method for mapping from orthography to phonemic transcriptions. Thus, Prolog's reply to the preceding query is to print out:

```
squeak skwik
```

that is, the paired orthographic and phonemic strings. (If you use a named variable, e.g. X, instead of the anonymous variable _, it will give the ASCII codes of each string too.)

It works the other way round too: by providing just a phonemic string, by keeping track of the transitions that the machine goes through and writing down the *first* symbol of each pair in the transition label, you can get a possible graphemic string corresponding to a given phonemic input. By responding to Prolog's output by entering a semicolon, additional solutions can be generated. For example:

```
?- accept(X,"f0ks").
focks f0ks

X = [102, 111, 99, 107, 115] ;
fox f0ks

X = [102, 111, 120] ;
phocks f0ks

X = [112, 104, 111, 99, 107, 115] ;
phox f0ks
```

Well, it doesn't necessarily give the right answer first time! But in that way we can find homophones.

Once again the machine is completely noncommittal as to whether it is mapping from graphemes to phonemes, or phonemes to graphemes. A third possibility, of course, is that you provide both a graphemic string and a phonemic string. In that case you can only get through the network if the

grapheme to phoneme correspondences encoded in the transition work are accepted. So that is a way of asking the machine to determine whether the particular phonemic transcription is a valid phonemic transcription of the graphemic transcription, and vice versa, whether the graphemic string is a valid spelling of the phonemic transcription. The fourth possibility is when you specify neither of the two strings. You input a variable for both the graphemic string and the phonemic string. That will non-deterministically generate a correspondence between the spelling and pronunciation of a syllable. By backtracking, you can generate all spelling–sound correspondences for all the syllables of the language.

There is a caveat, though: the power of the machine is limited by how much of the string it can look at at each transition. The examples that we have had so far are correspondences between single symbols in one string and single symbols in the other, where a 'single symbol' might actually be ornate: it might be a digraph or a trigraph, but it is effectively interpreted by the machine as a single symbol. There is no look-ahead mechanism in a finite-state machine, so you can't peek ahead to see whether or not there is some letter coming up later in the string. However, there is a way to encode a kind of look-ahead, which is to actually make the symbols on the transitions longer sequences of letters. For instance, in dealing with spelling to phoneme relations we must consider the behaviour of 'magic e' (the letter *e* after a vowel and a consonant that makes the vowel long) in the English orthography. For example: s͟i͟t vs. s͟i͟t͟e, c͟a͟n vs. c͟a͟n͟e, p͟a͟s͟t vs. p͟a͟s͟t͟e, r͟o͟t vs. r͟o͟t͟e. The pronunciation of a vowel letter depends on whether or not there is an 'e' after the next consonant, further along the string. NFST1 does not deal with most cases of 'magic e' (apart from –ce, –de, –se and –ze in state 7), though it does accept an 'e' after certain consonants (i.e. –dge, –ve, –ge, –gue and –the) that has nothing to do with vowel length. The only way in which you can build those kinds of correspondences into a deterministic finite-state machine without adding a new processing mechanism is to use transition labels that are three or four letters long at a time, and relate three- or four-letter sequences to three or four phoneme symbols at a time, for example ast:/ast/, but aste:/eɪst/. So you overcome the fact that you can't compute non-local dependencies in a string by making them local, by using longer chunks. The bigger the chunks get, however, the greater the number of transitions you need to have, since there is a very large number of mappings between sequences of three or four letters and sequences of three or four phoneme symbols.

Student: You can get into trouble with that kind of thing though can't you? The letters 'a, m, e' correspond phonemically to /eɪm/, but suppose you give the machine the string 'c, a, m, e, r, a'? How is it going to process that? Is it going to first map the c into a /k/? Then it might hit an 'a' and think it is /æ/. But if it processes 'a, m, e' as a single unit, corresponding to /eɪm/, it will go wrong.

The trouble with orthography is that sometimes these doublets and triplets correspond to phonological units and sometimes they do not.

You are right. It doesn't present any *general* computational problems, but it does present problems for the implementation of that particular task. By making the chunks of strings that are processed on each step larger, we introduce non-determinacy into the network, because the number of possible transitions to the next states becomes larger the more letters that you have on each transaction. There are various solutions about what to do with the sequence of letters a, m, e. We could just accept the non-determinism, and then there will be no need to use such long substrings as transition labels. Alternatively, you can attempt to prioritize the choice of which transition to follow next. The usual scheme for doing that is to try to match the longest substrings first, on the theory that they are more specific, special cases. You make the shorter substring matches of a lower priority. But that is actually adding an ordering scheme to the basic finite-state mechanism. We would stay within the bounds of finite-state languages, but that would be a new, non-standard variant of the finite-state architecture. I raised this example to show one of the limitations of finite-state machines. It is a limitation that can be overcome, but it is an intrinsic limitation, in as much as things become more complicated the more of the string you try to scan in any one chunk.

5.11 Using finite-state transducers to relate speech to phonemes

The examples in the previous section are about relations between one kind of alphabetic string and another kind of alphabetic string, grapheme strings and phoneme strings. Or instead of grapheme to phoneme mappings, we could map graphemes to allophones directly if you wanted to, or from morphophonemes to allophones, or from virtually any alphabetic representation to any other alphabetic representation that corresponds to the first in a certain way. (The kinds of correspondences that can be computed using finite-state transducers are known as regular relations.) But the symbols in the transition labels do not have to be letters of the alphabet (*any* alphabet: Roman, phonetic, Arabic, Thai, etc.). *Any* finite set of symbols will do. One instance of special interest in speech processing is that we can treat a set of acoustic parameters, such as a vector of LPC predictor coefficients, taken together as *one symbol*. For example, frame 200 of `joe_coeffs.dat` is a vector of 14 LPC coefficients for (part of) the vowel /ɑ/ of 'father'. We could treat them as a single symbol, a set of features, if you like, as in phonology, and write:

$$[\alpha] = \begin{bmatrix} a_1: & 2.693137 \\ a_2: & -3.15723 \\ a_3: & 2.153815 \\ a_4: & -0.46244 \\ a_5: & -0.62918 \\ a_6: & 0.194162 \\ a_7: & 0.696667 \\ a_8: & -1.27494 \\ a_9: & 1.502201 \\ a_{10}: & -1.37626 \\ a_{11}: & 1.135756 \\ a_{12}: & -0.52237 \\ a_{13}: & -0.11574 \\ a_{14}: & 0.135437 \end{bmatrix}$$

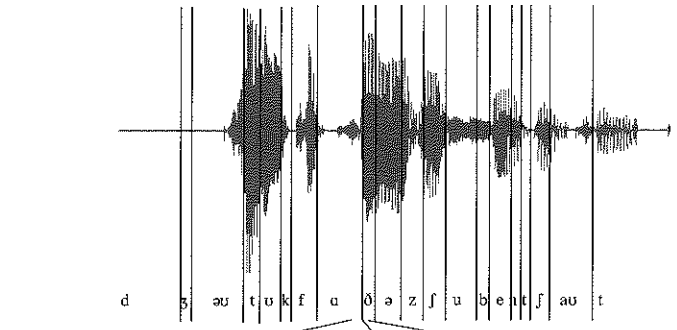
The number of possible values and combinations of all the different coefficients is very large, so the alphabet of these complex symbols is certainly enormous, but it *is* finite. So, consider a transducer in which the symbols on one side of the transition labels are LPC vectors and on the other side phoneme symbols. Each pairing of a phoneme symbol with an analysis vector represents a phonemic *labelling* of that analysis vector. For example:

```
"A":[2.693137   -3.15723   2.153815   -0.46244   -0.62918
    0.194162   0.696667   -1.27494   1.502201   -1.37626
    1.135756   -0.52237   -0.11574   0.135437]
```

So how can we find out what the correspondences are? Well, the first step is to record a speech database and encode speech into the desired parameters. Then, you segment the speech into phonemes, and provide phonemic labels for each segment, as in the upper part of figure 5.11. (This is usually a painstaking, long, manual task, possibly requiring many person-months of work.)

Then for every 5 or 10 ms frame in every segment you associate that phoneme label with that frame. (This needs to be automated to be practical.) This means that a stretch of speech in the database that is a complete vowel, say, will consist of a certain number of frames, say 30 frames, and each of those frames will have the same vowel label, as in the lower part of figure 5.11. That part of the figure shows a segment of speech from the 204th 5 ms frame of `joe.dat` (towards the end of the /ɑ/ of 'father') to the 212th frame, shortly after the start of the /ð/. Below each frame number is a phoneme label for that 5 ms interval, and 14 LPC coefficients, a_1 to a_{14}. Time is in the horizontal dimension, and analysis features are in the vertical dimension.

From such a database, we can construct a set of transitions in which the alphabet on the speech side is an alphabet analysis of vectors, and the alphabet on the linguistic side is an alphabet of phoneme labels of the



Label:	ɑ	ɑ	ɑ	ɑ	ð	ð	ð	ð
Frame: 204	205	206	207	208	209	210	211	212
LPC vector:								
2.693137	2.610945	2.597558	2.565878	2.319291	2.349247	2.321899	1.863409	1.481066
−3.15723	−3.07347	−2.94453	−2.91241	−2.11807	−2.11037	−2.02477	−0.63155	0.010306
2.153815	2.379279	1.995438	2.108499	1.147475	0.910962	0.814372	−0.56747	−0.66528
−0.46244	−0.9316	−0.35989	−0.8136	−0.3249	0.256971	0.287573	0.335704	0.133916
−0.62918	−0.35334	−0.83202	0.027484	0.184156	−0.79563	−0.68009	0.429717	0.126124
0.194162	0.306683	0.485961	−0.43509	−0.80504	0.373898	0.156924	−0.89356	−0.28647
0.696667	0.268792	0.402462	1.037625	0.898168	−0.0097	0.262798	0.252545	0.115398
−1.27494	−0.76344	−1.07576	−1.31662	−0.49597	−0.10334	−0.3322	0.661088	0.046145
1.502201	1.145245	1.177903	1.283225	0.157776	0.202759	0.226109	−0.7357	0.109533
−1.37626	−1.13322	−0.6209	−1.15597	0.013327	−0.37699	−0.28064	−0.03584	−0.32924
135756	0.999281	0.119541	1.134817	0.012791	0.677501	0.512114	0.689633	0.540491
−0.52237	−0.65051	0.418474	−0.68714	0.354996	−0.35548	−0.15489	−0.0299	−0.0326
−0.11574	0.121429	−0.70822	0.124007	−0.60873	−0.16315	−0.20147	−0.65855	−0.46588
0.135437	0.046852	0.305646	0.003265	0.24165	0.122075	0.072926	0.305114	0.200167

FIGURE 5.11
A phonemic labelling of a speech file

ɑ:	ɑ:	ð:	ð:	ð:
2.565878	2.319291	2.349247	2.321899	1.863409
−2.91241	−2.11807	−2.11037	−2.02477	−0.63155
2.108499	1.147475	0.910962	0.814372	−0.56747
−0.8136	−0.3249	0.256971	0.287573	0.335704
0.027484	0.184156	−0.79563	−0.68009	0.429717
−0.43509	−0.80504	0.373898	0.156924	−0.89356
1.037625	0.898168	−0.0097	0.262798	0.252545
−1.31662	−0.49597	−0.10334	−0.3322	0.661088
1.283225	0.157776	0.202759	0.226109	−0.7357
−1.15597	0.013327	−0.37699	−0.28064	−0.03584
1.134817	0.012791	0.677501	0.512114	0.689633
−0.68714	0.354996	−0.35548	−0.15489	−0.0299
0.124007	−0.60873	−0.16315	−0.20147	−0.65855
0.003265	0.24165	0.122075	0.072926	0.305114

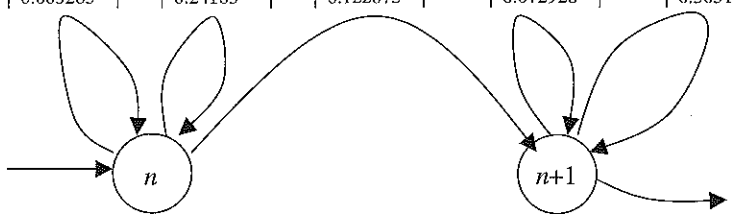


FIGURE 5.12
Part of a finite-state transducer that relates phoneme labels to LPC vectors

usual kind. The transition labels are pairings of phoneme label with vectors of analysis features. We can disregard the frame numbers. For transitions from one vector to the next within a phoneme, we use self-loops, but for transitions from one phoneme to the next, we employ two separate states, as in figure 5.12. So the sequences of state transitions that the machine will accept are sequences of phoneme–frame correspondences.

Now, suppose we want to use such a machine to produce transcriptions on the basis of the speech analysis vectors. The basic idea is the same as when we were discussing grapheme-to-phoneme conversion earlier on. Then, if we didn't specify the graphemes but just provided the phonemes, NFST1 could compute the graphemes for us as it goes through the transition network. The same applies here: if we don't specify the phoneme labels but the set of correspondences between phoneme labels and frames is known and encoded in the machine, by presenting the machine with a sequence of analysis vectors we can recover the corresponding sequence of phoneme labels. Now consider a machine for transcribing vowel-consonant sequences that only recognizes one particular vowel-consonant sequence, /ɑð/. The machine might only have two states, as in figure 5.12, with a very large number of self-loops from the state 1 to itself, each of which represents a possible vowel frame. In state 2, the consonant frames will be likewise represented by self-loops consisting of consonant symbols paired with acoustic parameter vectors. There will also be some particular vectors that have been observed at the transition from a vowel to a consonant. Provided we have multiple different tokens of /ɑð/, there will be more than one transition from state 1 to state 2, all representing the change from /ɑ/ to /ð/. For consistency let's give it the consonant label. There will be a great many self-loops in state 1, even if the machine only recognizes one vowel phoneme. And even if it only recognizes one consonant in state 2, there will be an awful lot of self-loops there, one for each distinct observed frame. If we enlarge the set of vowels and consonants that this machine recognizes, there will be an even greater number of self-loops, so the number of transitions in the machine will be very large. But the structure of the machine itself is extraordinarily simple: while the machine is seeing '/ɑ/-type' frames it stays in state 1, and associates each vowel frame with the label /ɑ/. Only if/when a '/ð/-type' frame is input can it make a transition from state 1 to state 2. Then if the frames after that continue to be of type /ð/, it will continue to generate a sequence of /ð/ labels. If the device works in the way that it is intended to, when given a sequence of frames as the input it will generate the corresponding sequence of phoneme symbols. A long sequence of *identical* symbols have to be contracted into a single label, so that the sequence of frame labels /ɑɑɑɑɑɑɑðððððð/ will be abbreviated as /ɑð/. So given the acoustic parameters of a speech signal, we could produce a hypothesis about what phonemes were input to the machine. It would be a rudimentary kind of speech recognition device; well, a kind of phonemic labelling device, at least.

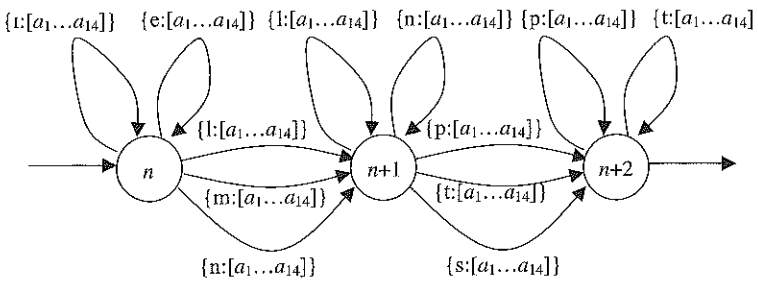


FIGURE 5.13
Schema for a finite-state transducer for VCC sequences

Student: Would a two-state machine like this handle VCC sequences?
No, but it would not take much to extend it to longer sequences. You'd have a state for vowels, a state for the consonants that can follow the vowel, and then a state for the second consonant in the sequence, as in figure 5.13.
Student: So the number of states in the machine is not tied to the number of distinct phonemic symbols that it could output.
That's right. I guess you could pursue that logic even further and say you only need one state. But the reason for having separate states is to ensure that only certain patterns of vowels and consonants are acceptable.

Now the question is, does it work? Well, yes it does, and no it doesn't. It works after a fashion, but it doesn't work as well as we might like it to. For the machine in figure 5.12 I picked a particular vowel and a particular consonant. But we are unlikely to want a machine that is so limited. We are more likely to want a machine that can recognize different vowels and consonants. So in state *n*, as well as the /ɑ/-to-frame correspondences, we are going to have some /e/-to-frame correspondences, and correspondences for other vowels. Likewise, in state *n*+1 as well as /ð/-to-frame correspondences we are going to have /p/-to-frame correspondences, and other consonants, like in figure 5.13. Now the problem is going to come that it may so happen that for some frame of the input that is spoken to the machine, that frame may actually be more like an /ʌ/ than an /ɑ/. That could be the case, for instance, at a vowel-to-consonant boundary where the spectrum of the vowel is changing. Or even if it is not, even if it is just coincidence, a particular frame at a particular point in the input may happen to physically be more like what had been stored away as an /ʌ/ frame, than an /ɑ/ frame. What are we going to do about that? It might only cause a glitch at one point, in which case the output might contain a sequence like /ɑɑɑɑɑʌɑɑɑ/. That could be a remediable problem, because most of the symbols in the output are /ɑ/s and only one of them is an /ʌ/, but the question is, how do you cope with that? What we need is some kind of confidence measure that tells us 'well, it is mostly /ɑ/'. We could count the number of times a letter is continuously repeated in the output, and if there are more /ɑ/'s than /ʌ/'s we might decide that it is an /ɑ/. But what if a changing sound really *was*

spoken? Maybe what the person said was 'eye', pronounced [ɑʌəi]. The whole business becomes a lot more tricky when we start talking about confidence measures, and asking what was the most likely input given a sequence of observations. Was it an /a/, a /t/, an /e/ or what? We have pushed this kind of machine to the limits of what it can achieve. In order to make this approach work, we need to add a probabilistic dimension to the machine, so that we can make judgements such as 'well, it is probably an /a/, and this sequence of frames is probably such and such a phoneme.' That is a topic for chapter 7, but I thought that I would raise it here as an indication that yes, the technique does work. It will slavishly compute sequences of symbols given sequences of frames in the input, but whether the sequences of symbols that it returns are quite what we are expecting or hoping to get is a different matter.

Student: Can you say 'if the frame is less than 5 or 10 milliseconds, it is really too short, so it doesn't count'?
Well, there are a variety of novel and imaginative ways in which we could try to resolve these problems, to try to get the device to perform how we want it to. These problems have taxed the minds of people working in speech recognition for years and years and a large number of bright ideas have been tried. Some improve the situation and some don't. But those considerations really take us beyond the scope of this chapter.
Student: As it stands, could it cope with double articulation? For example, when someone says 'apt', there is a short interval when the lips are closed for the /p/ and the tongue tip is raised for the /t/. Those two articulations actually overlap.
Yes, provided that the correspondences between symbols and analysis frames that are encoded in the transitions of the machines were included in the speech database, there is absolutely no reason why not. The machine does not care about the *linguistic plausibility* of the sequence of frames that it accepts. All that finite-state transducers do is to compute correspondences between descriptions on two different levels.

5.12 Finite-state phonology

Because of that fact, a number of people have proposed that finite-state transducers can be used in phonological modelling, which brings us back to the topic I started this chapter with. We can use finite-state transducers to compute the transitions between lexical (morphophonemic) representations and phonetic representations, as in standard generative phonology for instance. One of the attractive computational properties of finite-state machines is that you can cascade them: you can take the output of one machine and put it into the input of another machine. That is exactly the kind of thing that linguists want to do in standard generative phonology, where you compute the output of one rule, and take it as the input to another rule. So for each phonological rule you can build a little transducer,

and then to represent the cascade of rewriting rules you can combine the transducers together. To cascade two transducers you can't just take the end states of one machine and join them to the start states of the next machine: that is not applying the rules in order, that is simply processing the first part of a string with one machine, and the rest of the string with a second machine. In order to cascade transducers you have to merge them in a way that I am not going to go into. It involves combining the sets of states and adding or removing transitions to collapse the two machines into one. There are some well-defined techniques for cascading two transducers because of which you can take a set of generative phonological rules and the order in which they are applied, build a transducer for each rule, and then cascade the whole set of transducers into a single large transducer, in which the effects of all of the rules have been worked out and combined together.

Let's consider an example, based on the standard phonological analysis of the alternation between *revise* ([ɹɪvʌɪz]) and *revision* ([ɹɪvɪʒən]). According to a commonly repeated generative phonological analysis (Chomsky and Halle 1968; Halle and Mohanan 1985), the short [ɪ] in the second syllable of *revision* is a shortened version of an underlying long /ii/, shortened because two more vowels follow ('trisyllabic shortening'). The diphthong in the second syllable of *revise* is the default realization of underlying long /ii/. Conventional (but simplified) rewrite rules for these two relationships are given in (5.1).

(5.1) a. Trisyllabic shortening $V \rightarrow \emptyset \; / \; V - (C)\,V\,(C)\,V$
b. Vowel shift $i \rightarrow a \; / \; - \; i$

(5.1) uses a combination of the environment symbols '/' and '—' to abbreviate the full forms in (5.2).

(5.2) a. Trisyllabic shortening $V\,V\,(C)\,V\,(C)\,V \rightarrow V\,(C)\,V\,(C)\,V$
b. Vowel shift $i\,i \rightarrow a\,i$

According to this analysis, (5.2a) has to be applied to the underlying form before (5.2b), because trisyllabic shortening bleeds vowel shift. Applied the other way round, the output would be wrong:

(5.3) a. Correct rule ordering:

Underlying form	/riviiz/	/riviiz+iən/
Trisyllabic shortening	Not applicable	rivizіən
Vowel shift	rivaiz	Not applicable
(Other rules)	. . .	. . .
Surface form	[ɹɪvʌɪz]	[ɹɪvɪʒən]

b. Incorrect rule ordering:

Underlying form	/riviiz/	/riviiz+iən/
Vowel shift	rivaiz	rivaiz+iən
Trisyllabic shortening	Not applicable	rivaziən
(Other rules)	. . .	. . .
Surface form	[ɹɪvʌɪz]	*[ɹɪvaʒən]

Let's reconstruct this analysis using finite-state transducers, and get rid of the need for rule ordering. Recall how we encoded in figure 5.9 a searcher for the left-hand side *cad* of rules of the form $a \rightarrow b \; / \; c\text{—}d$. Figure 5.14 extends this by encoding the rewrite part of the rule, $a \rightarrow b$, as the

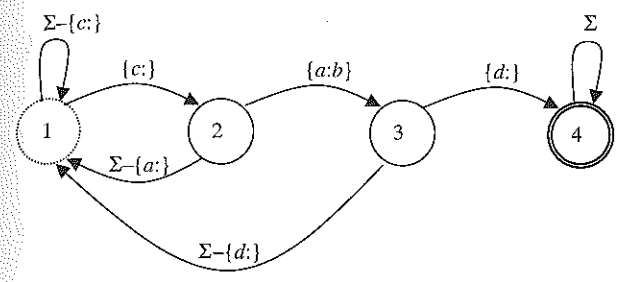


FIGURE 5.14
A schema for finite-state transducers that search for and map phonological rule environments of the form *cad* onto *cbd*, thus implementing rewrite rules of the general form $a \rightarrow b \; / \; c - d$

correspondence *a*:*b*. Note that the absence of a symbol on either side of the colon, for example *c*:, matches any symbol: we are not interested in what *c* or *d* map onto. Figures 5.15 and 5.16 give specific instances of how

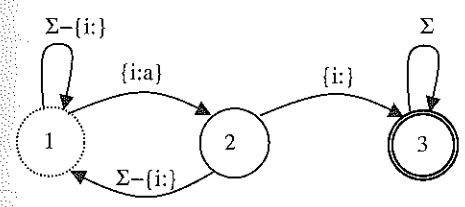


FIGURE 5.15
A transducer that encodes the vowel shift rule $i \rightarrow a \; / \; - \; i$

this transducer may be tailored to encode trisyllabic shortening and vowel shift. Note that, as in the conventional rule formalism, we use V to denote any vowel and C to denote any consonant. That is, {V:"} abbreviates {i:", e:", a:", o:", u:", ə:", ʌ:", ɒ:" . . .}.

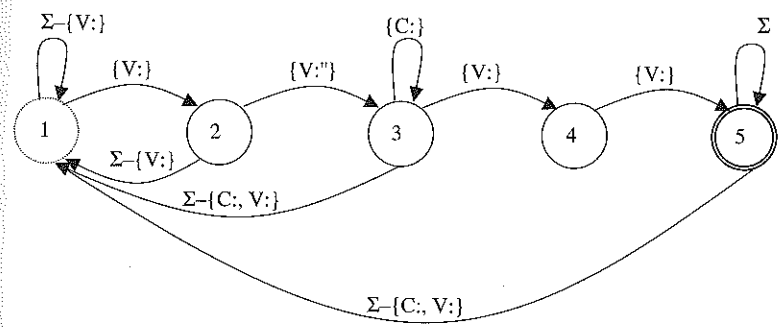


FIGURE 5.16
A transducer that encodes the trisyllabic shortening rule $V \rightarrow \emptyset / V - (C)\,V\,(C)\,V$

The crucial difference between the two rule orders is that in the correct order, the two rules cannot *both* occur, whereas in the incorrect order, both rules incorrectly apply, to derive *[ɹɪvaʒən]. Thus the rules are in an *exclusive or* relationship: one or the other rule can apply, but not both. We

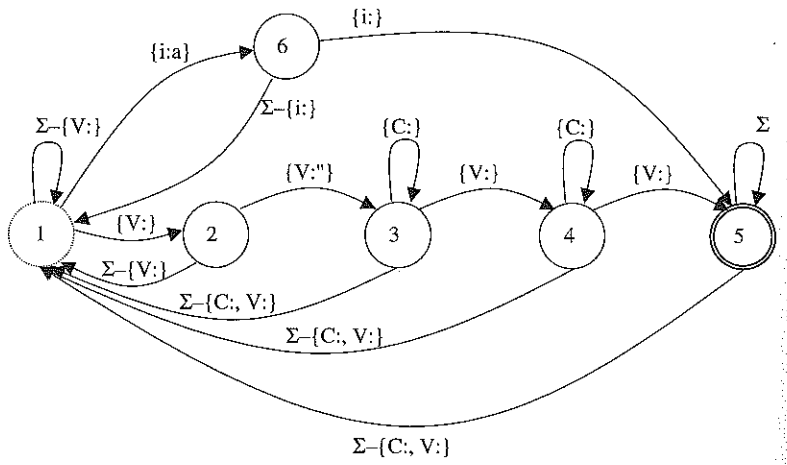


FIGURE 5.17
Result of combining the transducers in figure 5.15 and figure 5.16

therefore combine the two transducers into a single machine (figure 5.17) by establishing two separate paths from state 1 to state 5, such that either the state sequence of one machine or that of the other, but not both, may be followed.

The idea that standard phonological rules can be expressed as finite-state transducers has been very influential in computational phonology. The theoretical potential of this approach was first noted by Johnson (1972) and pursued in detail by Kaplan and Kay (1994) (a paper known since 1981 from a conference presentation), Koskenniemi (1983) and Karttunen (1983). The latter describes Kimmo, a system for automatically compiling two-level rules into finite-state transducers. More recent implementations of 'finite-state toolboxes' include PC-KIMMO (Antworth 1990) and the more general-purpose FIRE Lite toolkit (Watson 1999).

This work demonstrates (given appropriate caveats about the manner of rule application) that we can dispense with intermediate levels of representation and rule ordering. As a consequence, this approach to computational phonology is called Two-Level Phonology: as the name suggests, it employs only two levels of phonological representation, the lexical and surface levels.

I introduced transducers as a generalization of the finite-state machines that work with pairs of symbols rather than single symbols. Once you take that step, the floodgates are open: as well as working with pairs of symbols, you could compute correspondences between *any* number of symbols. Down that path lies a method for the computational implementation of autosegmental phonology, where you have to keep track of several parallel tiers. This possibility was first informally proposed by Kay (1987), and was further explored by Kornai (1991) and Wiebe (1992).

Kaplan and Kay (1994) acknowledge that cyclic rule application is a problem for finite-state approaches to phonology, because one of the conditions that have to be placed on SPE rules in order to implement them as finite-state devices is that a rule cannot reapply to its own input

(Johnson 1972). If that condition is not observed, SPE rules may have the power of context-sensitive grammars or even unrestricted rewriting systems. But if you impose the condition that a rule can't reapply to its own outcome at some later step in the derivation, the grammar is finite state. Kaplan and Kay argue that cyclicity is a contentious issue. There are certainly some unresolved issues as to whether or not cyclicity is dispensable. If it is always avoidable, then they are right, and most of phonology can be reduced to finite-state relations, but if they are wrong, that places a limitation on the circumstances in which finite-state methods are appropriate.

5.13 Finite-state syntactic processing

The literature on finite-state approaches to computational phonology is now quite large, so I have been rather selective about the references I have given. Before finishing, though, I want to mention one other example. I have been talking about phonetics, phonology and orthography, but the first use of these machines was in syntax. Figure 5.18 is an example of a finite-state machine for a subset of English expressions that might be used

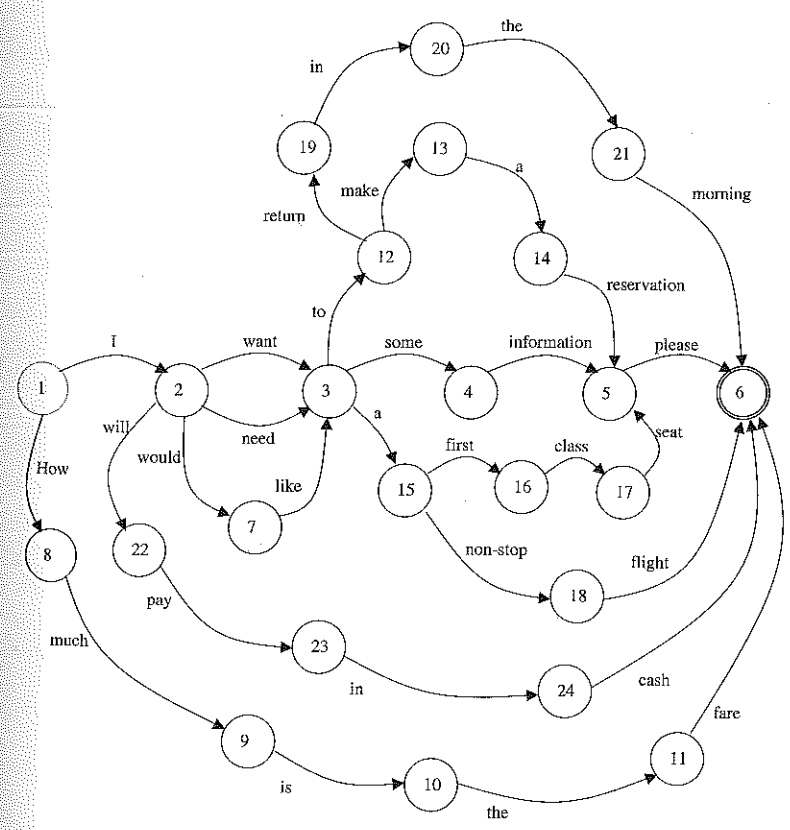


FIGURE 5.18
An example of a finite-state automaton for a small language (after Rabiner and Levinson 1981)

when booking an airline ticket. It is a simple kind of grammar that accepts certain sequences of words and not others. For certain kinds of applications that only use very simple languages, finite-state grammars are quite appropriate. In many circumstances where the range of messages that you want to recognize or generate is rather restricted, a finite-state machine might be more appropriate than a complex type of computational linguistic device, such as a full-blown linguistic parser of some sort. That is the case in many speech technology applications, where we are usually not so concerned with including all of the wonderful and elaborate sorts of linguistic constructions that one finds in generative grammar. You do not often find parasitic gaps in most people's requests for information, so a query like 'which files did you discard without reading?' may cause problems for such systems.

In chapter 2 of *Syntactic Structures*, Chomsky (1957) discusses three models of linguistic description. The first that he considers is a finite-state approach to syntax. He criticizes it, and shows a range of linguistic constructions that it can't handle. So it is ironic that in working linguistic systems in real life one finds finite-state methods being used more and more commonly, and often more successfully than more sophisticated kinds of linguistic parsers, which often just fall over, even though they are more theoretically respectable.

Student: But in all fairness what Chomsky was trying to say was that finite-state machines were not a general solution to syntactic problems. If you want to argue that they represent specific solutions to limited problems he might not argue with that.
That is true, you are dead right. But I think it is interesting historically that the wheel has turned full circle.
Student: Also the other argument he offered is that finite-state machines will produce a lot of junk that is not grammatical.
That is true too, but that is true of almost any linguistic theory. And if you over-constrain a grammar just a little too much, you can prevent a parser from accepting perfectly grammatical sentences. That is just as reprehensible as overgenerating, and can be far more annoying to a user, in practice!

But there is another issue, too, which is that the kinds of sentences with unbounded centre embedding that are widely cited as evidence that natural languages cannot be analysed with finite-state machines is easily addressed: they do not occur! To illustrate the importance of this, consider the following sentences:

(5.4) The malt lay in the house that Jack built.

(5.5) The malt that the rat ate lay in the house that Jack built.

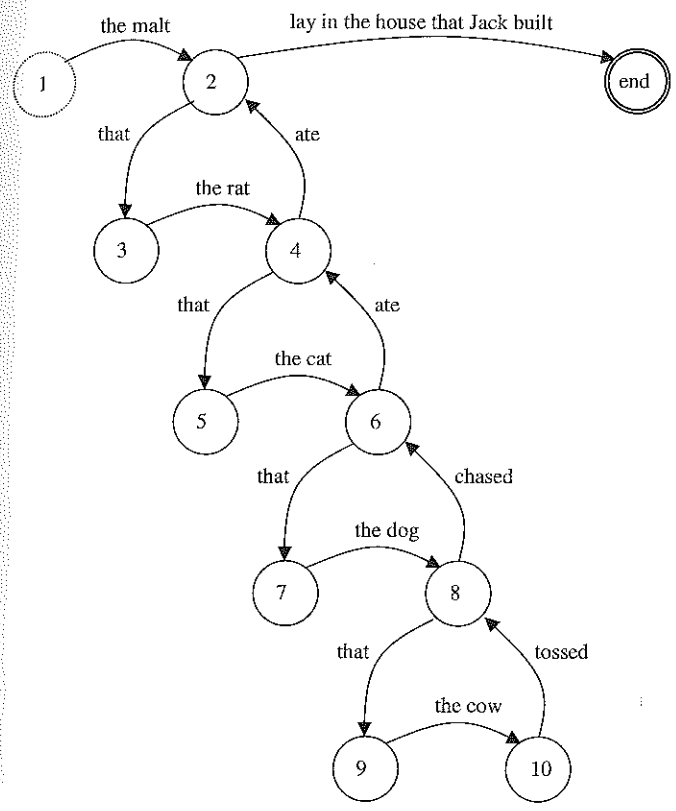


FIGURE 5.19
Centre embedding in a state transition network

(5.6) The malt that the rat that the cat ate ate lay in the house that Jack built.

(5.7) The malt that the rat that the cat that the dog chased ate ate lay in the house that Jack built.

(5.8) The malt that the rat that the cat that the dog that the cow tossed chased ate ate lay in the house that Jack built.

The syntactic structure of (5.4) to (5.8) can be expressed using a state transition network of a kind similar to that used in finite-state automata, as in figure 5.19. But if there is no limit to the depth of the centre embedding, this network is not a finite-state machine, as we cannot set a finite limit to how many additional nodes lie below state 10. But is that actually a problem? The degree of centre embedding exhibited in examples (5.6) to (5.8) is rare to non-existent. So a network with no further embedding below node 10 would actually be capable of accepting or generating (5.4) to (5.8) with a finite number of states. With this limitation, it would be a finite-state automaton. We shall return to this issue at the end of chapter 7.

Chapter summary

In this chapter we examined a simple but very versatile computational device: finite-state machines. We saw how they could be used to generate or accept strings of symbols (sequences of letters or words). The twin labels used in finite-state transducers allow a variety of mappings between levels of analysis to be modelled too, making them suitable for grapheme–phoneme conversion, or for associating labels with speech signals. The theoretical presentations were illustrated with working computational implementations written in the Prolog programming language.

Further exercises

Exercise 5.6.

Extend `nfst1.pl` to deal with 'magic *e*' in the following words: ape, ate, cake, babe, made, age, ace, haze, came, cane, pale, haste, eke, theme, pipe, site, pike, jibe, glide, ice, size, rime, fine, tile, hope, rote, coke, robe, code, doge, hose, home, tone, sole, dupe, jute, puke, cube, rude, luge, fume, tune, rule.

Exercise 5.7.

Adapt figure 5.19 into a finite-state transducer by labelling the transitions with part-of-speech categories. Implement the result as a Prolog program in the style of `nfst1.pl`.

Further reading

There are several good textbooks on Prolog: for example, Clocksin and Mellish 2003, Pereira and Shieber 1987 or Sterling and Shapiro 1994 are all highly recommended. Many textbooks on formal language theory and the foundations of computer science have some discussion of finite-state machines. Few attain the gold standard set by Hopcroft et al. 2000; for clarity of presentation, however, Jurafsky and Martin 2000: 35–52 deserves special commendation. For more on finite-state *phonology*, see the references in section 5.12. For applications of finite-state methods to morphological analysis and syntactic parsing, see the papers in Kornai 1999.

CHAPTER

6 Introduction to speech recognition techniques

CHAPTER PREVIEW

KEY TERMS

pattern-matching
decision tree
dynamic time warping
vector quantization
variability

In this chapter we examine a selection of techniques that have been used in speech recognition systems. We examine one important pattern-matching technique, dynamic time warping, in some depth.