FINITE-STATE TECHNIQUES

Finite-state automata (FSAs) are among the simplest computing machines that can be envisaged. They are well understood mathematically, easy to implement and efficient at doing what they do. If an NLP problem can be conveniently solved with a finite-state automaton, then it is probably a good idea to solve it that way. However, FSAs are subject to certain formal limitations that render them ill suited to certain computational linguistic tasks. This chapter provides a fairly comprehensive introduction to these machines and their implementation, indicates possible areas of application and gives some concrete examples of their use, and examines their limitations.

We begin this chapter by presenting finite-state transition networks (FSTNs). An FSTN can be regarded as a neutral description of a language (a set of sequences of symbols), but it can also be interpreted, for instance, as a specification of an FSA to recognize elements of the language or as a specification of an FSA to generate elements of the language. We move on to consider a simple extension to the basic FSTN notation, which then allows networks to be interpreted as finite-state transducers, which are FSAs that can recognize elements of one language, while generating elements of another. We conclude the chapter with a look at what FSAs can and cannot do.

Czestawa Szymanowska leaves the house of her relatives in San Francisco and takes the bus to the airport. Although her relatives left Poland in the 1960s, they continue to speak Polish at home, so the fact that Czestawa can speak hardly a word of English has caused her no problems whatsoever. But she is alone at the airport and she needs to know if she can reroute her return flight via London, so she can visit a cousin who her relatives have told her now lives there. There are long queues at the check-in desks so Czestawa goes over to one of the public terminals in the foyer and types her question in, in Polish – actually only an approximation to Polish, since the terminal has an ASCII keyboard and she is forced to omit diacritics. The machine duly replies, in Polish.

Putting aside for the moment all questions about how such a machine could have understood her query and constructed an answer, let us simply consider the question of how such a machine might infer that her query was expressed in Polish, rather than Spanish, Russian, English or any of the other dozen or so languages that it is equipped to handle.

Here are a couple of strawman solutions. Firstly, different languages employ the letters of the Roman alphabet with different frequencies. The letter 'j', for example, is much more commonly used in Dutch than it is in English. So, one possibility is to compile letter frequency tables for the various languages and then use these to compare against the frequency profiles of pieces of unknown text. The problem with this solution, which might be very satisfactory for books or newspapers, in the present context is that the sample of text is likely to be very small, perhaps only two or three words, and thus will not have any meaningful letter frequency profile.

A second candidate solution is suggested by the fact that in order to be made available for its presumed function, such a machine must, presumably, contain a parser for Spanish, a parser for Arabic, a parser for English, and so on. So, presented with the input in a language of unknown identity, we run the Spanish parser on it; if that fails to arrive at a parse, then we run the Arabic parser, and so on. Eventually, we try the Polish parser and succeed. The obvious problem with this solution, at least in a world of serial machines and relatively slow parsers, is that by the time that it realizes that the query was in Polish, the traveller Czestawa will have missed her plane. It is a case of using a sledgehammer to crack a nut.

Suppose this was what Czestawa typed in:

Czy pasazer jadacy do Warszawy moze jechac przez Londyn?

Now, the authors of this book do not speak a word of Polish, nor a word of Spanish, but if they are told that what Czestawa typed in is either Polish or Spanish, then they can tell right away that it is Polish – told that it was

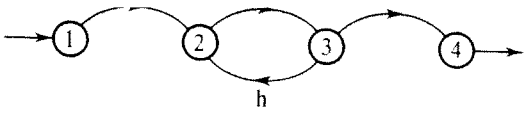

Figure 2.1 A laughing machine.

either Polish or Serbo-Croat, they might be less confident in their judgement. Indeed, if you were to put your hand over all but the first word, they would still guess confidently that they were dealing with Polish, not Spanish. So, the authors are willing to make a decision here based on exposure to less than a half-dozen letters in a language none of whose words are known to them.

The reasoning used is something like this:

- 'Czy' could not be a Spanish word but it could be a Polish word.
- If the first word is Polish rather than Spanish, then the chances are extremely high that the whole utterance will be in Polish, not a mixture of Polish and Spanish.

But how can we be so sure that 'czy' is not a word of Spanish? We cannot read, utter or understand any Spanish, and we have not bothered to look up 'czy' in a Spanish dictionary. Well, we cannot be absolutely sure, of course. But we have naive theories about the possible orthography (written form) of Spanish words and Polish words. These naive graphotactic theories are consistent with 'czy' being a Polish word, but not consistent with it being a Spanish word. If we could articulate theories of this kind, improve them in the light of the way Polish and Spanish graphotactics really work, and encode them in a manner that a computer could understand, then we would be a long way towards solving the problem that Czestawa's interlocutor solves in our scenario above.

However, before we tackle the problem posed by our airport advice unit, let us turn to a much more trivial problem, one which is apparently quite unrelated: getting a computer to laugh. Laughing, for present purposes, consists of displaying the sequences of characters 'ha!', 'haha!', 'hahaha!', and so on. Figure 2.1 is a diagram of an abstract machine, TO-LAUGH-1, that will do just that.

This diagram, a *finite-state transition network (FSTN)* is a conventional illustration of a very simple *finite-state automaton (FSA)* with four *states*, 1, 2, 3 and 4, state 1 being an *initial state*, signalled by the '→' pointing into it, and state 4 being a *final state*, signalled by the '→'

emerging from it. The four nodes of the graph representing these states are linked by four directed *arcs*:

- one, which starts from state 1 and goes to state 2, is labelled 'h';
- another, which starts from state 2 and goes to state 3, is labelled 'a';
- another, which starts from state 3 and goes to state 2, is labelled 'h'; and
- the final one, which is labelled '!', which goes from state 3 to state 4.

In machines of this kind, we will also make use of unlabelled arcs, which are referred to as 'jump' arcs for reasons that will become apparent.

How is this diagram to be understood? Well, the machine must start in an initial state, which can only be state 1 in this instance. If the initial state is also a final state, then it can simply do nothing, but that does not apply in the present case. If the state it finds itself in is not a final state, then it must change state by traversing an arc that starts from the state it is in and changing into the state to which the arc points. In traversing the arc, the machine must follow the instruction that labels the arc. Here, and for the present, we will interpret a label like 'h' as an instruction to print the letter 'h' – on a VDU, a printer, a paper tape or whatever. Since only one arc leaves state 1, our machine has no choices open to it: it has to change state to state 2 and print an 'h'. State 2 is not a final state, and like 1, it has but a single arc leaving it. So, the machine must follow this arc, print an 'a', and find itself in state 3. Now, for the first time, our machine is faced with a choice: it can move to state 4 or back to state 2. If it moves on to state 4, generating a '!', then it will have reached a final state. Now although an FSA, for that is what our diagram depicts, is not *required* to halt when it reaches a final state, unless the latter has no arcs leaving it, it is *permitted* to halt in final states. In this case, however, there is nowhere else to go and the machine will halt, leaving the string 'ha!' on the VDU as a trace of its activity. State 3 has another arc leaving it, labelled with 'h' and leading back to state 2. The machine may choose to traverse this arc. If it does so, then it will find itself faced with exactly the same sequence of arcs, actions and choices as it was when it was last in that state. And so we may see 'haha!' printed, or 'hahahaha!', or any such sequence.

We are interpreting this FSTN as a machine for *producing* laughter and it is important to appreciate that the choice that needs to be made when the machine finds itself in state 3, whether to move to state 4 or back to state 2, is a choice that is outside the machine's own competence. If we were to program a laughing machine based on this automaton, then our program would need to consult some outside authority, such as a random number generator, the time of day or the user, at this point.

But this 'production' or 'generation' interpretation of our FSTN is not the only possible interpretation, nor even the only useful one. Imagine a video game that tells the user jokes. When the user has got the joke, he

or she is supposed to type in 'haha!' or 'hahaha!' or the like. So, we see dialogues like this:

Game: <joke>
User: haha!
Game: You liked my joke. I will tell you another.
Game: <joke>
User: That was awful.
Game: You are not laughing. You did not like that joke. Never mind, perhaps this will amuse you.
Game: <another joke>
User: hohoho!
Game: You are not laughing. You did not like that joke. Never mind, perhaps this will amuse you.
Game: <yet another joke>

Putting on one side reservations we might have about the plausibility of any human user bothering to interact with a program as obviously dim as this one, let us consider how we could press our FSTN into service to check whether the user is 'laughing'. It turns out that we can, simply by changing our interpretation of the labels on the arcs. In the new interpretation, the game prints out its joke and then positions itself in an initial state corresponding to the initial node (1) of the network. It begins to inspect the user's input. If the first letter of that input is an 'h', then it traverses the arc that connects state 1 to state 2 and moves over the letter 'h' in the string that forms the user's input. Then it has to attempt to match the next letter in the string with the letter that labels one of the arcs leaving its current state; if there is a match, then it proceeds as before. If there is another letter in the string, but no match with an arc, then the machine fails to recognize the string. Likewise, if there are no further letters in the string and the machine finds itself in a non-final state. But if there are no further letters and the machine is in a final state, then it has succeeded in recognizing the string and so can print out its little message 'You liked my joke' and so on.

Notice that, in this dialogue, the game has failed to recognize 'hohoho!' as an instance of laughter. The reader should elaborate the machine in Figure 2.1 so as to permit the recognition of 'hohoho!', 'hoho!', 'hehe!', 'hehehehe!' and the like as laughter, but exclude 'hoha!', 'hehoha!' and other mixed cases.

In illustrating two different interpretations of our FSTN as machines, we have shown that the FSTN is actually quite a neutral description of the possible sequences of symbols that count as laughing. Because of this neutrality, we were able to interpret it both as a specification of a machine to *recognize* laughter and as a machine to *generate* laughter. To be

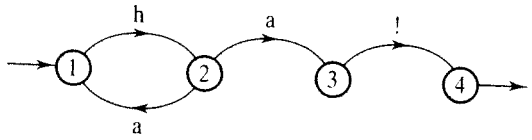

Figure 2.2 Non-deterministic laughing machine.

precise, we should distinguish clearly between the abstract description of laughing (the FSTN) and the different interpretations of it (the FSAs). For instance, there is a conceptual difference between the nodes in the graph and the possible states of a machine that the graph might represent. In practice, however, it is common to blur such distinctions, and so we will talk about FSTNs and the FSAs they could represent interchangeably, only making the distinction when it is important.

Recognition with an FSA so far seems very simple. But suppose we now consider a variant of our network, TO-LAUGH-2, shown in Figure 2.2. There is an important distinction between FSTNs like that in Figure 2.1 and that in Figure 2.2. The first specifies a *deterministic* automaton which, as its name suggests, is one whose behaviour (during recognition) is fully determined by the state it is in and the symbol it is looking at. Our second machine is *not* deterministic because if it finds itself in state 2 and looking at an 'a', then it can either return to state 1 and attempt to find a following 'h', or it can move to state 3 in the hope that the 'a' is the penultimate symbol in the string (since 3 leads to a final state in one transition). As this description is intended to imply, an implementation of a non-deterministic machine can follow false paths. Such an implementation will either need the ability to explore more than one path through the network of arcs simultaneously or else have the ability to backtrack to an earlier choice point when it discovers that it has been on a wild goose chase.

It was mentioned previously that unlabelled arcs can also appear in FSTNs. When the machine is generating, it can traverse such an arc, a 'jump' arc, without generating any symbol. When it is recognizing, it can likewise traverse such an arc without consuming any symbols from the input string. Jump arcs are an important source of non-determinism, because when an ordinary labelled arc leaves a state as well as a jump arc, the machine always has the option of taking the jump, even if the next symbol is the one required by the labelled arc. Figure 2.3 shows another non-deterministic laughing machine, TO-LAUGH-3, where the non-determinism arises at state 3.

The distinction between deterministic and non-deterministic automata is an important one, and we shall return to it from time to time in the course of this book. In the present context, that provided by finite-state machines, the distinction is one of style rather than substance: for every

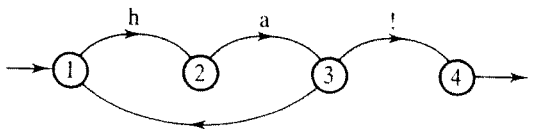

Figure 2.3 Laughing machine with 'jump' arc.

non-deterministic FSTN, there is an equivalent deterministic FSTN that describes exactly the same set of strings of symbols. We will not attempt to prove this here but the reader's attention is drawn to the fact that the deterministic FSTN in Figure 2.1 is exactly equivalent to the non-deterministic FSTN in Figure 2.2 in terms of the strings of symbols that it describes.

At this point, the link between our laughing machine and the Polish tourist in San Francisco airport may be beginning to become apparent. If we can encode our knowledge of possible English/French/Polish... words in FSTNs, there can be a relatively simple mechanism for determining in which languages a given sequence of words might be acceptable. Before we return to Czestawa, however, we will step back from the examples and consider an alternative way of talking about FSTNs. The graphic state diagram used in Figure 2.1 is perspicuous, at least for simple machines, but it does not lend itself to ready communication with computers, certainly not with those that are restricted to character and keyboard input. Nor does it lend itself readily to mathematical or logical analysis. Besides, a single notation provides only one perspective on a class of formal objects. Different notations, although all ultimately equivalent, can provide different perspectives and different ways of thinking about the formal objects, and how to use them and implement them. Accordingly, we will now define an intendedly perspicuous formal language for FSTNs.

2.2 A notation for networks

An FSTN description has three components:

(1) A name for the network.
(2) A collection of declarations.
(3) A collection of arc descriptions.

The first component, the name, is only of mnemonic value when we are discussing FSTNs, since no other component makes reference to it. However, as we shall see in Chapter 3, the name of a network plays an absolutely crucial role in the definition of RTNs. It is introduced here for

consistency and for completeness. The relevant statement of our language simply consists of the word Name followed by a name for the network (any string of one or more characters), followed by a colon. For example:

```
Name TO-LAUGH:
```

The second component, the declarations, consists of a single obligatory initial states declaration and a single obligatory final states declaration. Each declaration consists of a word indicating what is being declared and a list of the terms being declared, separated by commas. Thus, for instance:

```
Final 1, 2, 3
```

declares 1, 2 and 3 to be final states. So, a set of declarations for the FSTN in Figure 2.1 would look like this:

```
Initial 1
Final 4
```

It is often very convenient to be able to employ a single expression to abbreviate some subset of symbols and so we will introduce a type of declaration explicitly for this purpose. An abbreviation declaration consists of the abbreviatory expression, which must be distinct from any symbol, followed by the word abbreviates, followed by a colon and a list of symbols, separated by commas and terminated by a period. The use of this type of declaration will be clearer in the light of some examples:

```
V abbreviates:
  a, e, i, o, u.
C abbreviates:
  b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x, y, z.
```

We will put no orthographic restrictions on symbols, states or abbreviations: any character or string of characters, upper or lowercase, may appear in any role. When other things are equal, we will adhere to a convention of using integers for states, lowercase (strings of) letters for symbols and uppercase (strings of) letters for abbreviations. But this is merely the convention used here – it is not to be thought of as part of our definition of the formal language.

The third major component of our FSTN description consists of a set of one or more arc descriptions, each of which has exactly the same form. An arc description consists of the word From followed by a state, followed by the word to, followed by another state (not necessarily distinct from the first state), followed by the word by, followed by a single item. In this context, an item can be a symbol, an abbreviation (declared elsewhere) or the hash symbol (#, pronounced 'jumping' in this context).

This sounds much more cumbersome than it looks. Here, to illustrate, are the arc descriptions that would be needed to describe Figure 2.1:

```
From 1 to 2 by h
From 2 to 3 by a
From 3 to 2 by h
From 3 to 4 by !
```

And here is the complete description of the FSTN in Figure 2.3:

```
Name TO-LAUGH-3:
  Initial 1
  Final 4
  From 1 to 2 by h
  From 2 to 3 by a
  From 3 to 1 by #
  From 3 to 4 by !.
```

Notice that our choice of symbols so far has always been restricted to single letters and !. But nothing in the theory of FSTNs restricts us in this way, and, in fact, we can define an even simpler laughing machine by using a single multi-character symbol:

```
Name TO-LAUGH-4:
  Initial 1
  Final 2
  From 1 to 1 by ha
  From 1 to 2 by !.
```

This FSTN illustrates another aspect of FSTNs that we have not explicitly mentioned up to this point; namely, the possibility of an arc looping back to the state that it started out from. This is very useful when we wish to allow for unrestricted repetition of a symbol. The following FSTN, ENGLISH-1, uses such loops to good effect in a machine that will produce or recognize simple English sentences.

EXAMPLE: English-1 FSTN

```
Name ENGLISH-1:
  Initial 1
  Final 9
  From 1 to 3 by NP
  From 1 to 2 by DET
  From 2 to 3 by N
```

```
From 3 to 4 by BV
From 4 to 5 by ADV
From 4 to 5 by #
From 5 to 6 by DET
From 5 to 7 by DET
From 5 to 8 by #
From 6 to 6 by MOD
From 6 to 7 by ADJ
From 7 to 9 by N
From 8 to 8 by MOD
From 8 to 9 by ADJ
From 9 to 4 by CNJ
From 9 to 1 by CNJ.

NP abbreviates:
  kim, sandy, lee.
DET abbreviates:
  a, the, her.
N abbreviates:
  consumer, man, woman.
BV abbreviates:
  is, was.
CNJ abbreviates:
  and, or.
ADJ abbreviates:
  happy, stupid.
MOD abbreviates:
  very.
ADV abbreviates:
  often, always, sometimes.
```

This network uses some abbreviations corresponding to basic syntactic categories used in the description of English syntax. What is a syntactic category? Well, at its simplest, a *syntactic category* is the name we give to collections of expressions of the language that have the same distribution. So, for example, 'Sandy' and 'the person you spoke to' are both noun phrases: they may or may not refer to the same person, but regardless of that, they have exactly the same syntactic privileges of occurrence. Construct any English sentence you like using the proper name 'Sandy' and you will be able to construct another perfectly grammatical English sentence by substituting 'the person you spoke to' for 'Sandy' in the sentence you first constructed. The two sentences may mean different things, but they will both be grammatical. Examples of syntactic categories

that linguists commonly make use of are noun phrase (NP), sentence (S), verb phrase (VP), verb (V), and so on. We can also think of more detailed descriptions, such as plural noun phrase, interrogative sentence, passive verb phrase and tensed verb, as examples of more specified syntactic categories. Notice that in our example, the two noun phrases that we claimed to be intersubstitutable were actually both singular noun phrases. Although a singular noun phrase can sometimes be substituted for a plural one (or conversely) while preserving grammaticality, this is not always the case.

For the purposes of our illustrative example, a word is an NP (noun phrase) if it can stand alone as the subject of a sentence. By N we mean common noun, as distinct from proper noun. Determiners (DET) are words that can come before (common) nouns, as in 'the woman', but adjectives (ADJ) like 'stupid' can interpose. The category BV is used here for parts of the verb 'to be' which can be optionally followed by an adverb (ADV) such as 'often'. A word of category MOD is used to modify adjectives, changing 'stupid' into 'very stupid', for instance. Finally, the category CNJ (conjunction) is used for words that can join two sentences into a single larger sentence. Here are some example strings that can be recognized by this network:

> Kim was happy.
>
> Lee is a consumer and often very very stupid.
>
> Sandy was sometimes a stupid man and Kim is always the happy woman.

Our indentation is conventional, not part of the language definition, but adhering to it will make some of the complex networks exhibited in the next chapter a good deal easier to read. It will be convenient to have a name for this formal language in the course of this chapter and the next, and we shall refer to it as *NATR* (*network and transducer representation*).

We have not, as yet, said anything explicitly about what exactly it is that NATR means. For the most part, this is too obvious to need saying, but in the case of abbreviations we have a bit of notation that does not map one to one into our existing graphic representation. However, our abbreviations are exactly what their name suggests. Thus, for example:

```
From 1 to 2 by A.
A abbreviates:
  a, b, c.
```

is wholly synonymous with:

```
From 1 to 2 by a
From 1 to 2 by b
From 1 to 2 by c.
```

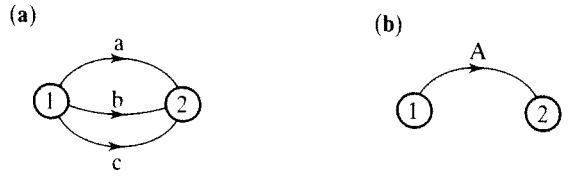

Figure 2.4 Abbreviating multiple arcs.

and thus corresponds to the FSTN fragment shown in Figure 2.4(a). Since fragments like this get to look very messy when more than two or three arcs join a given pair of states, we shall, henceforth, import our abbreviations into the graphic representation whenever it is convenient to do so. So, we may exhibit the fragment of Figure 2.4(b) in place of the unabbreviated fragment in Figure 2.4(a).

Restricting ourselves to the deterministic type of FSTN for the moment, we can introduce another formalism for (partially) representing (deterministic) FSAs used for recognition. This is the state-transition table, a matrix that exhibits the transition function of an FSA in a very clear and readily implemented manner. The vertical axis lists the states and the horizontal axis lists the symbols. The states in the matrix represent the state the machine moves into if it starts in the state given by the vertical coordinate and consumes the symbol given by the horizontal coordinate. Here is the state-transition table that corresponds to Figure 2.1:

	h	a	!
1	2	0	0
2	0	3	0
3	2	0	4
4	0	0	0

A zero indicates that the automaton cannot proceed, and will thus be unable to accept an input string that would have the effect of forcing it into that position in the matrix. So, for instance, the first row of the matrix shows that if the machine is in state 1, the only possibility is a transition to state 2, which requires the consumption of the symbol 'h'. Similarly the third row shows that from state 3 there are two possibilities, corresponding to the symbols 'h' and '!'. Bear in mind that, as it stands, the state-transition table is an incomplete representation of the automaton, since it does not indicate which states are initial and which are final.

Now, at last, we are in a position to directly address the problem with which we started, that of natural language graphotactics (the definition of possible words in terms of permissible letter sequences). One graphotactic constraint that is familiar to most users of English is that the

letter 'q' must be followed by a 'u'. Another example from English is the requirement that a word-initial 'st' only be followed by a vowel, a 'y' or an 'r' (we ignore the word 'sthenic' on the grounds that no self-respecting native speaker of English would dream of using it). Constraints of this kind are easy to code up in an FSTN. ENG-MONOSYL is a first attempt at an FSTN for the orthographic forms of possible one syllable English words.

EXAMPLE: FSTN for possible monosyllabic English words

Name ENG-MONOSYL:
Initial 1, 2
Final 3, 4, 5
From 1 to 2 by C0
From 2 to 3 by V
From 3 to 4 by C8
From 4 to 5 by s
From 1 to 7 by C3
From 7 to 2 by w
From 1 to 6 by C2
From 6 to 2 by l
From 6 to 5 by #
From 1 to 5 by C1
From 5 to 2 by r
From 1 to 8 by s
From 8 to 5 by C4
From 8 to 2 by C5
From 3 to 9 by l
From 3 to 10 by s
From 3 to 11 by C7
From 9 to 4 by C6
From 10 to 4 by C4
From 11 to 4 by th.

V abbreviates:
a, ae, ai, au, e, ea, ee, ei, eu, i, ia, ie, o, oa, oe, oi, oo, ou, ue, ui.
C0 abbreviates:
b, c, ch, d, f, g, h, j, k, l, m, n, p, qu, r, s, sh, t, th, v, w, x
C1 abbreviates:
d, sh, th.
C2 abbreviates:
b, c, f, g, k.
C3 abbreviates:
d, g, h, t, th.
C4 abbreviates:

c, k, p, t.
C5 abbreviates:
c, k, l, m, n, p, pl, qu, t, w.
C6 abbreviates:
b, f, m.
C7 abbreviates:
d, f, l, n, x.
C8 abbreviates:
b, c, ch, ck, d, f, g, h, k, l, m, mp, mph, n, ng, p, que, r, s, sh, th, v, w, x, y, z.

It is important to remember that FSTNs such as the one just presented only attempt to define *possible* English (or Polish or Spanish) words, not *actual* English (or Polish or Spanish) words. In the case of Polish and Spanish, it is rather likely that an FSTN for the actual words could be defined, but it would be a massive undertaking to attempt to construct it by hand. And there are languages, such as Bambara, for whose actual word set it seems that no FSTN could be given. But for many practical purposes, an FSTN for possible words is more use than one for the actual words. Speakers make words up constantly, but their neologisms are always possible words of the language in question. Thus, an English speaker will hardly notice the neologism in 'her secretary nixonized the tapes' but will immediately react to the third item in 'her secretary nxioniezd the tapes.'

We have been discussing graphotactics, but FSTNs are not restricted in their application to this, linguistically rather barren, level of description. For most languages, although not all, it makes sense to postulate a unit of syntactic organization between the letter and the word. This unit is known as a *morpheme* (roughly speaking, the minimal meaningful unit). Here, as elsewhere in this book, we restrict our attention to the written form of languages (In the spoken form, we have individual sounds, known as *phones*, rather than letters of an alphabet, and these sounds typically group into higher level sound units such as syllables.) Rather than attempt to define the notion of a morpheme, we will simply exhibit the component morphemes of some English words:

print-s
print-ed
print-ing
re-print
im-print
slow-ly
in-de-cipher-able

English does not have a rich morphological structure but many languages allow the speaker to express in a single word something that an English

speaker would require a whole sentence to say. For example, the Swahili word 'wametulipa' translates into English as 'they have paid us', while 'unamsumbua' translates as 'you are annoying him.' It is often possible to capture the permissible morpheme sequences of a language, or of part of a language, by means of an FSTN. SWAHILI-1 is an FSTN for a subset of Swahili words.

EXAMPLE: Swahili-1 FSTN

Name SWAHILI-1:
Initial 1
Final 5
From 1 to 2 by SUBJ
From 2 to 3 by TENSE
From 3 to 4 by OBJ
From 4 to 5 by STEM.

SUBJ abbreviates:
ni, u, a, tu, wa.
OBJ abbreviates:
ni, ku, m, tu, wa.
TENSE abbreviates:
ta, na, me, li.
STEM abbreviates:
penda, piga, sumbua, lipa.

This FSTN will recognize each of 100 morphological variants of four Swahili verbs. This is pleasing, but is it useful? The answer is probably no. After all, if we are concerned with Swahili words at this kind of level of detail, as opposed to wondering whether some string of characters was a possible Swahili word, then we are almost certain to want more than the yes or no that an FSTN can provide. To get more information, we need a parser or a transducer, not a recognizer. We will look at finite-state transducers in some detail in a subsequent section of this chapter.

At this point, let us take stock and recapitulate with some practical rules of thumb for the specification of FSAs by FSTNs:

- If we want to get from state *i* to state *j* without moving across a symbol in the input, or putting a symbol on the output, then we connect *i* and *j* with a jump arc.

- If we want to get from state *i* to state *j* while having the option to move across, or print, a symbol *s*, then we connect *i* and *j* with two arcs, one a jump arc and the other labelled with *s*.
- If we are in state *i* and wish to move across, or print, zero or more occurrences of a symbol *s*, then we connect *i* to itself with an arc labelled with *s*.
- If we are in state *i* and wish to move across, or print, one or more occurrences of a symbol *s*, then we connect *i* to a new node *j* with an arc labelled with *s*, and then we connect *j* back to *i* with a jump arc.
- If we are in state *i* and wish to move across, or print, one or more occurrences of a string *S* that can be reached between states *i* and *j*, then we connect *j* back to *i* with a jump arc.
- If the presence of even one occurrence of the string is optional in the latter case, then *i* should also be connected to *j* with a jump arc, pointing in the opposite direction to the one already introduced.

These are simply heuristics: they may not have the intended effect if they interact with other features of the FSTN, most obviously if any of the named states, or intermediate states, are also final states.

Exercise 2.1 Draw out the graphic representation for the ENGLISH-1 FSTN. [*easy*]

Exercise 2.2 Design an FSTN that will recognize well-formed English number names – for example, 'four thousand seven hundred and one', etc. [*easy*]

Exercise 2.3 Design an FSTN that will recognize well-formed time of day expressions – for example, 'five fifteen', 'quarter to four', etc., in either British or American English. [*easy*]

Exercise 2.4 Investigate the problem that the jump arc gives rise to in converting Figure 2.3 into a state-transition table. [*intermediate*]

Exercise 2.5 Devise a variant of the state-transition table that will allow general non-deterministic FSTNs to be represented. Use it to represent the FSTN in Figure 2.2 and the ENGLISH-1 network. [*intermediate*]

Exercise 2.6 Investigate the adequacy of the ENG-MONOSYL FSTN for its intended domain:

(a) What, if any, monosyllabic English words does it fail to accept?

(b) What, if any, impossible (in English) letter sequences does it accept?

Modify it to correct any inadequacies you uncover. [*intermediate*]

Exercise 2.7 Extend the ENG-MONOSYL FSTN to handle English words that may have several syllables. [*intermediate*]

Exercise 2.8 Design a comparable FSTN for a language such as Polish or Spanish, using your own knowledge of the language, a patient informant or a dictionary. [*intermediate project*]

Exercise 2.9 Write a program that will read a (possibly very large) file of words and construct a letter sequence FSTN on the basis of those words. What theoretical issues does such a project pose? How might you evaluate the result? [*hard project*]

2.3 Representing FSTNs in Prolog

It is straightforward to represent FSTNs in Prolog and to write programs to perform various operations with them. Our approach here will be to represent networks declaratively as data structures (facts in the Prolog database) that can be examined and manipulated. We will then write general-purpose recognition and generation programs that will work on any network represented as a data structure in the approved way.

Before we start writing general network traversal programs, we need to consider how we are going to represent transition networks as Prolog data structures. We need to know the following about a network:

- What its initial nodes are.
- What its final nodes are.
- What its arcs are, where each arc is defined by: the departure node, the destination node and the label on the arc.

This can be represented most straightforwardly in Prolog by using simple predicates of the appropriate kind:

```
initial(NODE).
final(NODE).
arc(DEPARTURE_NODE, DESTINATION_NODE, LABEL).
```

Here are some example networks (the ENGLISH-1 example appears in the appendix in the file **fstnarcs.pl**):

```
% SWAHILI-1
%
initial(1).
final(5).
arc(1, 2, subj).
arc(2, 3, tense).
arc(3, 4, obj).
arc(4, 5, stem).

% ENGLISH-1
%
initial(1).
final(9).
arc(1, 3, np).
arc(1, 2, det).
arc(2, 3, n).
arc(3, 4, bv).
arc(4, 5, adv).
arc(4, 5, '#').
arc(5, 6, det).
arc(5, 7, det).
arc(5, 8, '#').
arc(6, 7, adj).
arc(6, 6, mod).
arc(7, 9, n).
arc(8, 9, adj).
arc(8, 8, mod).
arc(9, 4, cnj).
arc(9, 1, cnj).
```

Something is missing from these network definitions (at least, under their intended interpretations); namely, a statement of what such symbols as **subj** abbreviate. We could readily adopt the following style of declaring these abbreviations:

```
abbreviates(subj, [ni, u, a, tu, wa]).
```

But, for reasons of consistency with our treatment of the lexicon in subsequent example programs, we will adopt a more verbose, but equivalent form of declaration:

```
word(CATEGORY, WORD).
```

Here, the CATEGORY argument corresponds to the abbreviation symbol, while WORD corresponds to one of the items subsumed under this

abbreviation. Using this scheme, the abbreviations for ENGLISH-1 are:

```
word(np, kim).
word(np, sandy).
word(np, lee).
word(det, a).
word(det, the).
word(det, her).
word(n, consumer).
word(n, man).
word(n, woman).
word(bv, is).
word(bv, was).
word(cnj, and).
word(cnj, or).
word(adj, happy).
word(adj, stupid).
word(mod, very).
word(adv, often).
word(adv, always).
word(adv, sometimes).
```

Although this is verbose, it is not inefficient: it may involve a Prolog interpreter in just as much work to search down a list checking membership as it does to search through its database looking for the appropriate **word** entry.

Exercise 2.10 Write a program that will transform a database of assertions of the form:

```
abbreviates('A', [b, c, d]).
```

into a database consisting of assertions like:

```
word('A', b).
word('A', c).
word('A', d).
```

[*easy*]

Exercise 2.11 Provide operator definitions that will allow examples such as the following to be treated directly as Prolog clauses:

```
initial a.
final 5.
'Subj' abbreviates: [ni, u, a, tu, wa].
from 1 to 2 by 'Subj'.
```

Note that this is close to the NATR formalism used in the text, but not identical to it due to the single quotes and the use of square brackets and lowercase. Can your operator definitions deal with:

initial a, b, c.
final 5, 6.

[*easy*]

2.4 Traversing FSTNs

The various interpretations of FSTNs as FSAs all involve FSAs that go from state to state in a way that echoes a traversal of the network. We thus turn now to the question of how to set about getting a computer to traverse networks of the kind we have been examining. One way to imagine the traversal of a network is to think of a kangaroo that jumps from node to node and which is constrained to take jumps only where there are arcs. As the kangaroo jumps about, portions of the input string are consumed, if the network is being used for recognition, or portions of the output string are produced, if the network is being used for generation. During recognition, we can think of the input string being displayed in large letters on the wall, with some kind of pointing device indicating which symbols remain to be processed. In a finite-state network, the kangaroo is only allowed to make a jump if:

(1) The arc is labelled by a symbol that is the same as the next symbol in the input.

(2) The arc is labelled by an abbreviation and the next symbol in the input is one of the symbols covered by that abbreviation.

(3) The arc is labelled '#'.

In the first two cases, the kangaroo moves the input pointer forward one word and makes the jump. In the latter, it simply jumps across without changing the input pointer.

The kangaroo model is a useful way to think about the information that needs to be kept during the traversal of a network. If we wish to translate this into computer programs, we must think carefully about what the kangaroo needs to remember and what information is available to it. In doing recognition, our kangaroo needs to keep track of its own current location (which node it is at) and the pointer into the input string. Moreover, it is capable of moving that pointer. In summary, at any moment in the traversal of an FSTN, the state of the computation can be characterized by the following items:

- R1. a node name (the kangaroo's location).
- R2. the remaining input string.

When the network is being used for generation, we will have to replace these items with the following:

- P1. a node name (the kangaroo's location).
- P2. the output string generated so far.

Note that both R1 and R2 influence the future behaviour of the automaton in recognition mode, whereas P2 is irrelevant to future behaviour in generation mode.

Apart from our reference to the distinction between deterministic and non-deterministic FSTNs, we have not really addressed the issue of making choices in this discussion so far. In general, successful traversal of an FSTN involves choice and hence search. Given that R1, R2 characterize the complete current state of the traversal, this is precisely the information that we will need to keep if we wish to record different possibilities that need to be investigated. Following standard terminology, we shall call a collection of items R1, R2 that characterizes a complete (intermediate) state of a traversal a *state*, even though this word is also used to refer to a state of an FSA. (Woods uses the term *configuration* for a complete intermediate state of a parser, and this might be a more appropriate term to use here.) We will show a state as a sequence of two items, as follows:

<2, a h a !>
 ↑ ↑
 | R2. remaining input (symbols to be processed)
 R1. current node

This state represents the intermediate point of a traversal where the kangaroo has got to node 2 and still has to deal with the symbols 'a', 'h', 'a' and '!' (in that order).

When we are traversing a network, we need to keep track of both the *current state*, which expresses where we are now, and *alternative states*, which express other possibilities that could be tried as well as the current one. Consider what happens, for instance, if we are using the network of Figure 2.2 and get into the state just described. In the network of Figure 2.2, there are two arcs leaving state 2: if the next symbol in the input is an

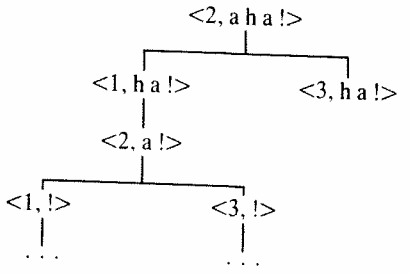

Figure 2.5 Search tree.

'a', then any of them could be traversed, leading to the following possible *next states*:

<1, h a !>
<3, h a !>

In general, we might expect each of these to yield several next states, and each one of these to yield several next states, and so on, giving rise to a whole tree of possibilities, as shown in Figure 2.5.

In practice, the *search tree* arising from this example is not very complex, as most branches quickly lead to states from which no further progress can be made. Nevertheless, only one of the possible states can be investigated at a time – the *current state*. As the tree is generated, the new states need to be remembered somehow so that they can be investigated if necessary – they need to be added to some representation of the *alternative states* there are so far. One way of organizing the search is to keep a 'pool' of alternative states to be investigated. At each stage in the traversal, one of these alternatives is selected and made the current state. The valid next states from this state are worked out and added to the pool. Then another alternative is selected from the pool, and so on. We start off with a state in the set of alternatives for each initial node of the network. For instance, if s1 and s2 were the two initial nodes, s2 and s9 were the two final nodes, and the string was 'a b c d e', the initial set of alternative states would be:

<s1, a b c d e>
<s2, a b c d e>

And the set of legitimate final states would be:

<s2, >
<s9, >

If our machine winds up in either of these two states then it has succeeded in recognizing the string.

Here is an informal description of our algorithm:

(1) Create the set of alternative initial states (the initial pool).
(2) Do the following repeatedly:
  (2.1) Select one of the alternative states (removing it from the pool).
  (2.2) If it is a final state, stop and announce success.
    Otherwise,
    (2.2.1) Calculate the valid next states that could follow from it.
    (2.2.2) Add these to the pool of alternatives.
(3) If the pool of alternatives ever becomes empty, stop and announce failure.

This is obviously not a complete description of what to do, because it does not specify how to decide which alternative to select at each stage. Two main approaches to this problem are *depth-first* and *breadth-first* search. In terms of the search tree, depth-first search follows a path deeper and deeper in the search tree and only moves up the tree to consider an alternative if all the possibilities below its current state have been tried. On the other hand, breadth-first search investigates states roughly in order of their depth in the tree. Intuitively, depth-first search involves preferring to investigate alternatives that have been added to the pool recently, whereas breadth-first search attempts to be fair by avoiding any alternative remaining too long in the pool before being investigated. We will see a lot more of these basic strategies later in this book. Note also that this is a general search algorithm which can be used for any search task that can be characterized as an exploration of states of some kind.

How do we calculate the valid next states that follow from a given state <NODE, INPUT>? For each arc (with label L and destination D) leaving node NODE, we must include states in the set as follows:

(SYM) if L is a symbol and the first symbol of INPUT is the same, include <D, ... rest of INPUT ...>
(ABB) if L is an abbreviation and the first symbol of INPUT is covered by the abbreviation, include <D, ... rest of INPUT ...>
(JMP) if L is '#', include <D, INPUT>

2.5 Traversing FSTNs in Prolog

In this section and some subsequent sections, we will develop a number of programs for performing useful tasks with various kinds of networks.

These programs will have a certain 'family resemblance' but will differ in their details. The task we will consider initially is that of using a network for recognition.

The predicate **recognize(NODE, STRING)** will be true if the given STRING can be accepted by our FSTN with the traversal starting at the given NODE. Firstly, we want to recognize the empty string (a string being a list of words, for present purposes) if we are at a final node:

```
recognize(Node, [ ]) :-
  final(Node).
```

This clause says that **recognize(Node, String)** will succeed when **String** is empty and **Node** is **final**.

Secondly, we need to make provision for a traversal of an arc leading from the node we are at, followed by a continuation of the traversal from that point:

```
recognize(Node_1, String) :-
  arc(Node_1, Node_2, Label),
  traverse(Label, String, NewString),
  recognize(Node_2, NewString).
```

Notice that we call **recognize** recursively, starting it off at the destination node **Node_2** of the chosen arc. The predicate **traverse** is in charge of telling us whether we can traverse that arc and, if so, what change is made to the input string. Given the arc label and the initial string in its first two arguments, **traverse** returns the new value of the string in its third argument.

The definition of **traverse** reflects the three ways in which we can make progress through the network. The first way (SYM) involves crossing an arc labelled with the next word in the string:

```
traverse(Label, [Label | Words], Words) :-
  not(special(Label)).
```

This says that we can traverse the arc if the label is the same as the first word in the input, as long as it is not a 'special' symbol (an abbreviation or '#').

The second way (ABB) involves continuing down a string whose first word is subsumed under the category given by the arc label:

```
traverse(Label, [Word | Words], Words) :-
  word(Label, Word).
```

This says that we can traverse the arc if the label is an abbreviation and the first word on the input is covered by that abbreviation. As in the last case, the new string returned is simply the rest of the string.

The final clause (JMP), for 'jump' arcs, says we can recognize the string from a node if there is a jump to another node. In this case, we return the original string unchanged:

```
traverse('#', String, String).
```

To complete our definition, we need to define the predicate **special**, which detects abbreviations and the jump symbol:

```
special('#').
special(Category) :-
  word(Category, _).
```

Finally, in checking recognition of a whole string, we need to make sure that we start out from an initial node, and we can put this requirement in a simple **test** predicate that calls **recognize**:

```
test(Words) :-
  initial(Node),
  recognize(Node, Words).
```

The predicate **test**, given a list of words as argument, will succeed exactly when the string of words is accepted by the automaton encoded in the **initial**, **final**, **arc** and **word** statements. The definition of **test** completes the code of our recognizer, which also appears in the appendix as **fstnrecg.pl**.

The FSTN recognizer program has been described as if there is no search involved in recognition, but obviously we would like the program to work with non-deterministic automata as well as deterministic ones. When we call **recognize** with a particular string and node, we are entrusting that call with a particular state in the search space that is to be explored. Such a call is in charge of checking whether the state could be a final state, and simply succeeding if so. It is also in charge of computing the possible next states and calling **recognize** again to investigate them. Search arises if there is more than one possible next state from a given state. Our program handles this, thanks to the ability of **recognize** to do different things on backtracking. Thus, a given call may first invoke another **recognize** for one possible next state. If that recursive call does not succeed, it may then be able (by selecting an alternative **traverse** or **arc** clause) to make a call corresponding to an alternative next state. Figure 2.6 shows the **recognize** calls for a simple example using the ENGLISH-1 network: the example sentence, abbreviated in the diagram, is 'kim was often a happy woman'. The tree of possible calls mirrors exactly the search tree.

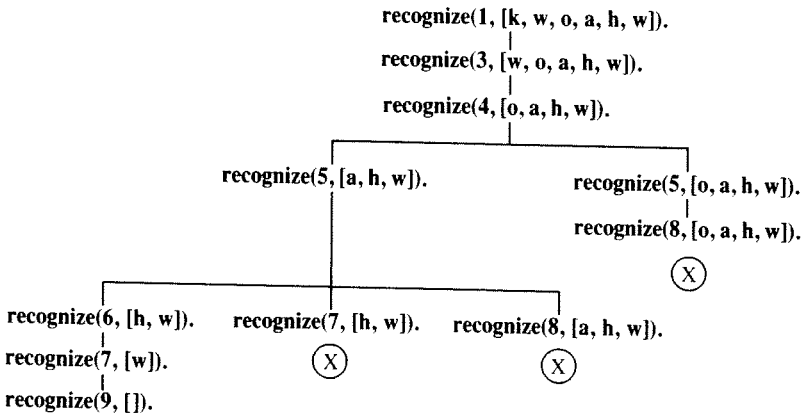

Figure 2.6 Call tree.

Notice that, because of the way Prolog works, only one path down this tree is investigated at a time. So, for instance, the call:

```
recognize(4, [o, a, h, w])
```

firstly calls:

```
recognize(5, [a, h, w])
```

and only if this fails (because this is not a final state and yet cannot be taken further) does it call:

```
recognize(5, [o, a, h, w])
```

This is the characteristic behaviour of depth-first search – a given possibility is followed up as far as possible before any alternative is considered. In this case, Prolog's backtracking facility means that we do not need to explicitly represent the pool of alternative states to be explored: the set of alternatives is represented implicitly in Prolog's memory of places to backtrack to.

Let us now consider what has to be done if we wish to adapt our recognition program for the exhaustive generation of all the sequences that are allowed by a transition network. We still need to search all possible paths through the network, but now we are not constrained to traverse arcs that are compatible with an input string. In fact, we do not really need to make any modifications to the program, since it will generate as it stands, given the goal **test(X), write(X), nl, fail.**, but we can package this up into a **generate** predicate if we wish:

```
generate :-
    test(X),
    write(X),
    nl, fail.
```

This provides us with exhaustive generation, but it cannot be allowed to run to completion if the network allows an infinite number of different strings. Nevertheless, it could still be useful, in principle at least, to run such a system for a while, to get an idea of the range of the strings allowed by a network. If we try generating from our example network for a fragment of English, however, we do not get a representative sample of the strings – the strings generated are as follows:

```
[kim, is, often, a, happy, consumer]
[kim, is, often, a, happy, consumer,
    and, often, a, happy, consumer]
[kim, is, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer]
[kim, is, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer]
[kim, is, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer,
    and, often, a, happy, consumer]
...
```

The trouble is that, once the program has made a decision about an initial set of words, it is happy to go on investigating other decisions that it encounters later. At node 9, the only final node, there is always the option to loop back to node 4 instead of finishing, and it is this decision, rather than any other, that is remade when we ask for alternative solutions. Only if, for some reason, all future possibilities fail to work out will it remake earlier decisions. Because of the way one call of **recognize** has to fail (exhaust all its possibilities) before an alternative can run, the program will always focus on one alternative and further developments from it before it tries other alternatives. This depth-first search behaviour is all right when the set of possibilities to be investigated is finite, but when the set is infinite, as in this case, it can lead to a very narrow exploration of alternatives, or, worse still, an infinite loop where longer and longer alternatives are tried and the program never actually stops with any solutions. Indeed, if we were to rearrange our **arc** database in ENGLISH-1 so that the arcs from node 6 came in the other order, the generation would get into an infinite loop without generating any possibilities.

Our recognition program does just that – it recognizes whether or not a string is accepted by a given FSTN. But although it discovers quite a lot about the string in the course of a successful recognition, this information is lost. All we get by way of a result is a truth value, which is reported by Prolog as yes or no. If we wish to recover the information about the sequence of nodes and arc labels that were involved in the successful recognition, then we need to introduce a further argument to collect this information. Equipped with such an argument, **recognize** ceases to be merely a recognizer and becomes instead a rather primitive form of parser. Here is the modified part of the program. The definition of **traverse** and **special** are as before:

```
parse(Node, [], [Node]) :-
  final(Node).
parse(Node_1, String, [Node_1, Label | Path]) :-
  arc(Node_1, Node_2, Label),
  traverse(Label, String, NewString),
  parse(Node_2, NewString, Path).
test(Words) :-
  initial(Node),
  parse(Node, Words, Path),
  write(Path),
  nl.
```

One way of thinking about what this program (which also appears in the appendix as **fstnpars.pl**) does is simply in terms of it recording the history of its successful arc transitions in a list. But another, equally good, way of thinking about it is as a program that maps one string of symbols (the sentence) into another string of symbols (the parse); in other words, as a program that *transduces* one string into another.

Exercise 2.12 You may have noticed that the Prolog version of the ENGLISH-1 network has the arcs in a different order to that presented earlier. Which of the programs presented does not behave well if the arcs are in the original order? Explain the problem. [*intermediate*]

Exercise 2.13 What would be the effect of adding an arc such as:

```
arc(4, 4, '#').
```

at the beginning of the definition of the ENGLISH-1 network? [*easy*]

Exercise 2.14 The preceding exercise provides a compact example of a more general problem. Define a predicate **check_the_net** that will print an error message if the net suffers from this general problem. Give an informal proof that if network does not provoke an error message from **check_the_net**, then the program given in the text will find an existing traversal if there is one, otherwise it will fail in finite time. [*intermediate*]

Exercise 2.15 Explain the effect of changing the clauses of the definition of **recognize** to read as follows:

```
recognize(Node, []) :-
  final(Node), !.
recognize(Node_1, String) :-
  arc(Node_1, Node_2, Label),
  traverse(Label, String, NewString), !,
  recognize(Node_2, NewString).
```

Would all the example networks retain their existing interpretation given this definition of **recognize**? What class of FSTNs would be unaffected by this change? [*easy*]

Exercise 2.16 Work out a way of representing state-transition tables in Prolog and write an FSTN recognizer that uses this representation. [*hard*]

Exercise 2.17 Modify the example network and the definition of **recognize** so that they dispense with **word** and use **arc** and **abbreviates** with the syntax exemplified here:

```
arc(from, 1, to, 2, by, subj).
abbreviates(subj, [ni, u, a, tu, wa]).
```

[*easy*]

Exercise 2.18 Write a version of **recognize** that conducts the search breadth first instead. In breadth-first search, each possible state is taken in turn and the new states that immediately arise from it are derived. Then, all these new states are taken in turn and the next states immediately arising from them are derived. The idea is to spend time evenly in the different parts of the search space. For this program, you will need to represent a state explicitly by a data structure – for instance, a term of the form **state(Node, String)**, of which a particular realization might be **state(3, [is, happy])** – and manipulate lists of these terms, representing sets of alternatives. Here are some predicates that you may need to define:

```
next_states(State, List)
% List is the list of possible next states that could follow from State.
next_states_list(List_1, List_2)
% List_1 is a list of states and List_2 is the result of concatenating together
% all the lists of next states arising from all the states in List_1.
```

You will probably need to use a built-in predicate, like **findall**, **setof** or **bagof**, that enables you to construct all the solutions of a Prolog goal in a list. [*hard*]

2.6 Finite-state transducers

An FSA of the kind we have discussed, used to analyze some existing input, is a recognizer, not a parser or a transducer, so all it can do is decide on the well-formedness of a string. If it can reach the end of the string in a final state, then the string is well formed; if it cannot reach the end of the string, or if cannot reach the end of the string and simultaneously be in a final state, then the string is not well formed. That is all the information it provides. To get more information, we need a parser or a transducer, not a recognizer.

Although we can interpret an FSTN as a machine that produces a simple yes or no output for any input, with some small extensions to the notation, we can interpret one as a *finite-state transducer (FST)*. An FST is a more interesting kind of FSA that allows a string of output symbols to be produced as an input string is recognized.

One way to think of an FST is as a special kind of FSA that inspects two (or more) symbols at a time and proceeds accordingly. To put some intuitive flesh on this rather barren statement, we describe a family of hypothetical children's games. Imagine a long, straight pavement made up of coloured, square paving stones set two abreast. At any given time, apart from when a player actually makes a move forwards, the player must have his or her left foot on a left-side paving stone and his or her right foot on the right-side paving stone that is immediately adjacent to it. And, at any given time, the player must be in one of a (finite) number of states. For simplicity, we will assume just two states: that of having your hand in your pocket, and that of not having it there. Each variant of the game consists of a collection of rules, of which the following are examples:

- If your left foot is on a red paving stone and your right foot is on a green paving stone and your hand is in your pocket, then you can advance to the next pair of stones and keep your hand in your pocket.
- If your left foot is on a blue paving stone and your right foot is on a blue paving stone and your hand is not in your pocket, then you can advance to the next pair of stones and put your hand in your pocket.

Given such a collection of rules, the player's goal is to move from the beginning of the pavement to the end, without making any move that is not expressly permitted by the rules. A child playing this game will be implementing an FST.

As we have just presented it, an FST is a kind of inspector that runs along checking if two strings of symbols stand in an appropriate correspondence. As such, it seems very little different, and hardly more useful, than an FSA running as a recognizer. But just as an FSTN can be interpreted both as a string recognizer and a string generator, so a network representing an FST can be interpreted, for instance, as both a string correspondence checker and as a machine that reads one string while writing another. We will use the term FST to refer to any such finite-state machine that makes use of multiple tapes, even though it is the latter interpretation that is of most interest and utility in the natural language domain. How, then, are we to conceive of an FST applying to a linguistic domain? Here is an example of the kind of rule that we might use in a linguistic application:

- If your current English word is 'fish' and you are in state 13, then you may print 'poisson' and advance to the next English word and shift into state 7.

But before we get overambitious and attempt English–French machine translation with an FST, let us step back and equip ourselves with a notation for talking about FSTs.

An FST can be specified by an FSTN whose arcs are labelled with pairs of symbols, as opposed to single symbols, and as before we will blur the distinction between the abstract notation, which describes sets of pairs of strings, and the possible machines that it might specify. To specify an FST, then, the simplest thing to do is just to augment the NATR language we already have for defining FSTNs. Let us imagine that the symbols we are dealing with are being read from, or printed on to, tapes. There is no need to make any changes in our formalism for expressing initial and final states. But we do need to allow our arc description statements to specify pairs of symbols as labels. We will use expressions like A_a, where A is a tape 1 symbol and a is a tape 2 symbol, for such symbol pairs. In fact, there is no need to restrict ourselves to two tapes. Both our syntax for transducer symbol declarations and our notation for symbol pairs are designed to generalize to the n-tape situation, for n greater than 2. Hence, A_a_α could be used for a symbol triple in a three-tape situation, for example. However, in what follows, we will restrict ourselves to two-tape machines.

An arc description statement for an FST will appear thus:

```
From 1 to 2 by A_a
```

What about cases where we wish to read, or write, a symbol on one tape, but do nothing on the other? Here, we can use our jumping symbol; so, #_a and A_# will be well-formed pairs. (Thus, the hash remains a special symbol, part of our language for talking about FSTs. If you want to have the FST manipulate the real hash symbol, then it will need to be set off in quotes.)

This minor generalization from single symbols to pairs – or, more generally, n-tuples – of symbols is all we really need to define FSTs. Here is

an example of an abbreviation for an FST:

DIGIT abbreviates:
one_un, two_deux, three_trois, four_quatre, five_cinq, six_six,
seven_sept, eight_huit, nine_neuf, ten_dix.

Having introduced these additions to our existing FSTN language, we can consider ENG_FRE-1, which is an example of a complete FST specified in this augmented NATR notation.

EXAMPLE: Eng_fre-1 FST

Name ENG_FRE-1:
Initial 1
Final 5
From 1 to 2 by WHERE
From 2 to 3 by BV
From 3 to 4 by DET
From 4 to 5 by NOUN.

WHERE abbreviates:
where_ou.
BV abbreviates:
is_est.
DET abbreviates:
the_#.
NOUN abbreviates:
exit_la sortie, policeman_le gendarme, shop_la boutique, toilet_la toilette.

The little FST described in ENG_FRE-1 would be completely trivial were it not for one wrinkle: in French the form of the determiner or definite article varies with the gender of the noun it accompanies, whereas English shows no such variation. This FST gets round the problem by letting the English 'the' map into nothing, and then requiring English nouns to map into the relevant French noun preceded by a determiner marked for the appropriate gender. Thus, la sortie and le gendarme, for example, are single symbols as far as this FST is concerned. This is not a very elegant solution to the problem since it obfuscates the correspondence that holds between the English 'the' and French 'la/le'.

An alternative FST for this English–French translation task, one that is arguably more perspicuous, despite the introduction of an additional state and two additional arcs, is given in ENG_FRE-2.

EXAMPLE: Eng_fre-2 FST

Name ENG_FRE-2:
Initial 1
Final 5
From 1 to 2 by WHERE
From 2 to 3 by BV
From 3 to 4 by DET-FEMN
From 4 to 5 by N-FEMN
From 3 to 6 by DET-MASC
From 6 to 5 by N-MASC.

WHERE abbreviates:
where_ou.
BV abbreviates:
is_est.
DET-FEMN abbreviates:
the_la.
DET-MASC abbreviates:
the_le.
N-FEMN abbreviates:
exit_sortie, shop_boutique, toilet_toilette.
N-MASC abbreviates:
policeman_gendarme.

Notice in ENG_FRE-2 how the gender distinction has been, in effect, encoded into the states: if we traverse the net via state 4, then we must have a feminine determiner and a feminine noun, whereas if we go via state 6, then we must have a masculine determiner and a masculine noun. There are no other possibilities.

Notice also that if we use this transducer to translate from French to English, then it will operate deterministically – it will never need to make a choice. But if we use it to translate from English to French, its operation will not be deterministic: in translating 'Where is the policeman?' it will be faced with a choice when it reaches the determiner. It can either traverse the DET-MASC arc or the DET-FEMN arc and it has no basis for deciding which. If it goes the DET-FEMN route, then it will fail when it reaches the N-FEMN arc, since policeman has no French counterpart in N-FEMN. So, any algorithm that we devise to employ FSTs that exhibit such non-determinism will need to incorporate either an ability to backtrack or an ability to explore multiple arcs in parallel. It is worth noting that our earlier and uglier English–French FST was deterministic in both directions.

The examples of FSTs that we have just been playing with are misleading in that nobody nowadays would dream of attempting to do serious English–French machine translation with an FST, for reasons that will begin to emerge in the final section of this chapter and which should be self-evident by the time you have reached the end of this book. But FSTs are potentially well suited to providing efficient solutions to certain small self-contained areas of linguistic analysis. Examples that spring to mind are the translation or interpretation of number names and time of day expressions (although not in all languages), text to speech transduction in languages with fairly well-behaved orthographies and the inflectional analysis of word forms. We will look briefly at the latter here, returning to our earlier Swahili example, reconstructed as an FST mapping between the Swahili morphemes and reasonably perspicuous representations of their syntactic and semantic content (SWAHILI-2).

EXAMPLE: SWAHILI-2 FST

```
Name SWAHILI-2:
  Initial 10
  Final 90
  From 10 to 21 by Subj_ni
  From 10 to 22 by Subj_u
  From 10 to 23 by Subj_a
  From 10 to 24 by Subj_tu
  From 10 to 25 by Subj_wa
  From 21 to 31 by 1ST
  From 22 to 31 by 2ND
  From 23 to 31 by 3RD
  From 24 to 32 by 1ST
  From 25 to 32 by 3RD
  From 31 to 40 by SING
  From 32 to 40 by PLUR
  From 40 to 50 by TENSE
  From 50 to 61 by Obj_ni
  From 50 to 62 by Obj_ku
  From 50 to 63 by Obj_m
  From 50 to 64 by Obj_tu
  From 50 to 65 by Obj_wa
  From 61 to 71 by 1ST
  From 62 to 71 by 2ND
  From 63 to 71 by 3RD
  From 64 to 72 by 1ST
  From 65 to 72 by 3RD
  From 71 to 80 by SING
  From 72 to 80 by PLUR
  From 80 to 90 by STEM.

1ST abbreviates:
  1st_#.
2ND abbreviates:
  2nd_#.
3RD abbreviates:
  3rd_#.
SING abbreviates:
  Sing_#.
PLUR abbreviates:
  Plur_#.
TENSE abbreviates:
  Future_ta, Present_na, Perfect_me, Past_li.
STEM abbreviates:
  LIKE_penda, BEAT_piga, ANNOY_sumbua, PAY_lipa.
```

This network will map a Swahili expression like 'wa-me-ni-sumbua' (on the second tape) into Subj-3rd-Plur-Perfect-Obj-1st-Sing-ANNOY (on the first tape), or conversely, and it can be seen why this might well be a useful thing to do in a system designed to analyze or synthesize inflected Swahili words. Notice that the FST given is deterministic when we use it to map from Swahili into our analysis expressions, but not when we go in the opposite direction. Notice also that we have been cheating in our discussion so far: real Swahili words come to us as 'wamenisumbua' not as 'wa-me-ni-sumbua' with all the morpheme breaks conveniently marked. But we can solve this problem with the FST machinery that we already have: we simply need to transduce 'w a m e n i s u m b u a' into 'wa me ni sumbua'.

Exercise 2.19 Draw a diagram of the SWAHILI-2 network. [*easy*]

Exercise 2.20 Design an equivalent FST that will run deterministically from the analysis expressions to the Swahili words. [*intermediate*]

Exercise 2.21 Elaborate the ENG_FRE-2 FST to cover a small but useful subset of the phrases that you might need in a foreign airport. [*easy*]

Exercise 2.22 Design an FST that will transduce well-formed English number names, such as 'twenty two', into the corresponding Arabic numerals – that is, '22'. [*easy*]

Exercise 2.23 Design an FST that will transduce well-formed English time-of-day expressions into their French counterparts, or those of some other language. Thus, for example, it should map 'seven minutes past noon' into 'midi sept'. [*intermediate*]

Exercise 2.24 Design an FST that will transduce well-formed English number names into well-formed French number names, or those of some other language. Thus, for example, it should map 'ninety four' into 'quatre vingt quatorze'. [*easy*]

Exercise 2.25 Design an FST to transduce 'w a m e n i s u m b u a' into 'wa me ni sumbua', etc., for all the Swahili data. [*easy*]

Exercise 2.26 Imagine two parallel sequences of phonemes and phones set side by side so that each phoneme is paired with a phone (ignoring null elements, for the sake of simplicity). An FST can crawl down this sequence of phoneme–phone pairs obeying rules such as the following:

- If your current phoneme is /d/ and your current phone is [t] and you are in state 17, then you may advance to the next pair and shift into state 11.

In this example, state 17 might well be the state that the automaton gets into as it emerges from an unvoiced consonant, say. If the FST is able to reach the end of the sequence of pairs in an approved final state, then the phone sequence is a well-formed realization of the phoneme sequence according to the rules that the FST embodies (and conversely). We can reformulate the inspection rule as one of transduction:

- If your current phoneme is /d/ and you are in state 17, then you may print [t] and advance to the next phoneme and shift into state 11.

Devise a small FST incorporating such phoneme–phone matching rules and show how it can handle some standard phonological problems. [*intermediate, requiring knowledge of phonology*]

Exercise 2.27 Design an FST that will translate three-character strings consisting of any letter of the alphabet, a mid-string marker character, say '^', and any letter of the alphabet into its mirror image. So, for instance, the string 'a^h' would be mapped into the string 'h^a'. Roughly how many states would you need for a similar FST that reversed strings like 'ak^yp' consisting of four alphabetic characters with a mid-string marker in the middle? [*intermediate*]

Exercise 2.28 Deterministic FSTs, like deterministic FSTNs, can be (partially) represented by state-transition tables. The difference is that whereas the state-transition table for an FSTN is a two-dimensional object, that for a two-tape transducer is a three-dimensional object, with one dimension for the states and one dimension for each of the symbol sets associated with the tapes. This is illustrated here with a two-dimensional slice of the three-dimensional matrix for the FST that we used as an example in the text:

est	1	2	3	4	5
where	0	0	0	0	0
is	0	3	0	0	0
the	0	0	0	0	0
exit	0	0	0	0	0
policeman	0	0	0	0	0
shop	0	0	0	0	0
toilet	0	0	0	0	0

This can be interpreted as saying that if you are in state 2 and the next word on your input tape is 'est', then you can print 'is' on your output tape and change into state 3. Otherwise, you are stuck. The full matrix would require five more of these two-dimensional slices, one for each word from the symbols vocabulary of the second tape. (An alternative to this three-dimensional approach would be to use a two-dimensional table with one dimension for the states and the other dimension for symbol pairs.) Write a program that will compile deterministic two-tape FST descriptions given in our formal language into three-dimensional state-transition matrices. Now write a program that will use these matrices, and information about initial and final states, to translate from the language used on one tape to that used on the other. What happens if your compiler is given a non-deterministic FST to compile? [*hard*]

2.7 Implementing FSTs in Prolog

An FST is just like a finite-state recognizer except that it deals with two tapes. Thus, we can amend our FSTN traversal programs to deal with FSTs, mainly by changing the arc statements to specify a pair of symbols (one from each tape) instead of just one. Where a transition is allowed by an abbreviation, however, the label can just be a single symbol, as before. Here is an example, the ENG_FRE-2 transducer, which appears in **fstrans.pl** in the appendix:

```
initial(1).
final(5).
```

```
arc(1, 2, 'WHERE').
arc(2, 3, 'BE').
arc(3, 4, 'FDET').
arc(4, 5, 'FNOUN').
arc(3, 6, 'MDET').
arc(6, 5, 'MNOUN').

word('WHERE', [where, ou]).
word('BE', [is, est]).
word('FDET', [the, la]).
word( 'MDET', [the, le]).
word('FNOUN', [exit, sortie]).
word('FNOUN', [shop, boutique]).
word('FNOUN', [toilet, toilette]).
word('MNOUN', [policeman, gendarme]).
```

Notice that abbreviations now stand for possible pairs of words, rather than simply possible single words. Given such **arc** statements, our **transduce** predicate can be defined along very similar lines to our earlier **recognize** and **parse** predicates. The only additional complication is that we need a modified version of **traverse**, **traverse2**, which takes *two* input tape arguments and produces *two* modified tape arguments. The definition of **traverse2** needs to consider the three possible jump situations: left tape but not right tape, right tape but not left tape and both tapes.

```
transduce(Node, [], []) :-
  final(Node).
transduce(Node_1, String1, String2) :-
  arc(Node_1, Node_2, Label),
  traverse2(Label, String1, NewString1, String2, NewString2),
  transduce(Node_2, NewString1, NewString2).

traverse2([Word1, Word2], [Word1 | RestString1], RestString1,
        [Word2 | RestString2], RestString2) :-
  not(special(Word1)), not(special(Word2)).
traverse2(Abbrev, String1, NewString1, String2, NewString2) :-
  word(Abbrev, NewLabel),
  traverse2(NewLabel, String1, NewString1, String2, NewString2).
traverse2(['#', Word2], String1, String1, [Word2 | RestString2], RestString2).
traverse2([Word1, '#'], [Word1 | RestString1], RestString1, String2, String2).
traverse2('#', String1, String1, String2, String2).
```

Notice that this definition of **transduce** is wholly reversible: it makes no difference to the program if we think of it as transducing from its second argument to its third or conversely. Indeed, we may even wish to use it to match inputs on both tapes or to generate output on both tapes.

As before, we need an additional predicate to ensure that we start out from an initial node, and we can conveniently use this predicate to print the transduced string out, as well:

```
test(String_A) :-
  initial(Node),
  transduce(Node, String_A, String_B),
  write(String_B),
  nl.
```

Exercise 2.29 Using pencil and paper, work out how you expect the transducer to respond to the queries:

```
test([where, is, the, shop])
test([where, is, the, policeman])
```

before submitting these queries to the program itself, which is repeated in **fstrans.pl** in the appendix. Can the program be used to translate simple French questions into English? [*easy*]

Exercise 2.30 As it stands, our program will not handle cases where an abbreviation indicates more than one word appearing on one of the tapes – for instance, the abbreviation NOUN in ENG_FRE-1. Modify both the way that **word** assertions are represented and the **transduce** program so that examples like ENG_FRE-1 can be run. [*intermediate*]

Exercise 2.31 Implement an FST program that employs state-transition tables. [*hard*]

Exercise 2.32 Write a program to translate deterministic FSTs as they are represented in the text above into equivalent state-transition tables. [*hard*]

2.8 Limitations of finite-state machines

One recent natural language application of finite-state devices has been in the recognition of inflected words in languages that make heavy use of inflectional morphology. Such a language is Finnish: nouns have 2000 odd

distinct forms and verbs some 12 000. To the relief of Finnish newborns, these forms mostly result from simple regular concatenation of morphemes, which linguists refer to as *agglutination*, but there is enough irregularity to make an accurate description non-trivial. For Finnish, we can define a grammar for inflected words which is essentially just an FSTN. Each morpheme, including the stem, is associated in the lexicon with:

- its (underlying) form,
- its syntactic/semantic properties, and
- a pointer to the items that may follow it.

An (underlying) inflected word form is well formed, roughly speaking, if and only if it consists of the concatenation of a sequence of forms, beginning with a root and ending with some conventional marker, arrived at by following pointers through the lexicon.

The interest of the work we have just described, from the perspective of computational linguistics, is that FSTNs and FSTs, especially if deterministic, allow for extremely efficient implementations. FSTs also have the advantage of being bidirectional: once we have the FST, we can get from, say, phone to phoneme just as easily as we can get from phoneme to phone. This makes FSTs attractive for a number of applications, ranging from spelling checking to machine translation in restricted domains. But finite-state devices have their limitations for NLP purposes, and we conclude this chapter by taking a look at some of them.

In evaluating such devices, there are two relevant notions of adequacy that need to be kept in mind, which we shall call 'mathematical' and 'notational', respectively.

Mathematical adequacy is concerned with whether the formal objects characterized by the notation, under the intended semantics, have the properties manifested in the real-world objects that the notation and its interpretation is intended to model. Although FSTNs can recognize non-finite languages, languages that contain an infinite set of strings, there are many non-finite languages that they cannot recognize. Thus, it is easy to build an FSTN to recognize the language of strings consisting of any number n of *a*s (known as the language a^n) or the language of strings consisting of any number m of *a*s followed by any number n of *b*s (the language, $a^m b^n$), but quite impossible to build an FSTN to recognize the language of strings consisting of some number n of *a*s followed by the same number of *b*s (the language $a^n b^n$), unless an upper bound is put on the size of n, in which case we cease to have a non-finite language. This is a failure of mathematical adequacy. Strings of the general form $a^n b^n$ arise in a language when the language permits one string to be embedded inside another and puts no limit on such embedding. A lexicon of the sort described in the opening paragraph of this section can only define sets of strings that are finite-state languages: a natural language whose word-set involved constructions like $a^n b^n$ would show that kind of account of the lexicon to be mathematically inadequate as a theory of natural language lexicons in general.

Notational adequacy is to do with how elegantly the notation describes the real-world objects. In general, a short description is preferable to a longer one, repetition and long windedness increasing the possibility of errors in the use of the notation. An ideal notation allows one to exploit the similarities between different structures and state general properties when they exist. Looking at the SWAHILI-2 network we can detect some notational inadequacy by observing the similarity between the arcs following the rewriting of the symbol Subj and those following the rewriting of the symbol Obj. For instance, the arc with label Subj_ni is immediately followed by one labelled 1ST, as is the arc with label Obj_ni. The notation does not allow us to 'factor out' the similarities between these sets of arcs. Similarly, if we wished to describe English sentences of the form subject–verb–object using a FSTN, we would find that we would have to separately describe possible subjects and objects, even though almost all phrases that can appear as one can equally appear as the other. We will return to this problem in Chapter 3.

Exercise 2.33 Readers unfamiliar with the fact that no FSTN can recognize the language $a^n b^n$ are encouraged to attempt to provide an informal proof of it. [*hard*]

SUMMARY

- FSTNs are the simplest approach to NLP.
- FSTNs are of very little use by themselves.
- FSTs map one string of symbols into another.
- FSTs can be used for sublanguage translation.
- FSTs can be used for morphological processing.
- FSAs are easy to implement.
- FSAs do not suffice for NLP.