Further reading

For technical introductions to FSTNs and discussion of some of their formal properties, see Aho and Ullman (1972, Sections 2.2 and 2.3), Aho and Ullman (1977, Chapter 3), Hopcroft and Ullman (1979, Chapter 2) or Gersting (1982, Chapters 7 and 8). Relevant accounts of directed graphs and graph traversal algorithms are to be found in Aho *et al.*(1982, Chapter 6) and Sedgewick (1983, Chapters 29–30). Technical presentations of FSTs are available in Lynch and Pierson (1968), Lewis and Stearns (1968), and Aho and Ullman (1972, Sections 3.1 and 3.3). Useful background reading on NLP uses of transition networks and pattern matching is provided by Winograd (1983, Chapter 2). Some researchers have continued to take an interest in finite-state devices for syntactic parsing: Blank (1985), Brodda (1986), Ejerhed (1982), Ejerhed and Church (1983), Langendoen and Langsam (1984), and Pulman (1986), for example.

The formal properties of the morphology of Bambara, mentioned earlier, are the subject of Culy (1985). Other work relevant to the question of the adequacy of finite-state machinery for phonological and morphological processing purposes includes Langendoen (1981), Carden (1983), Church (1983b, pp. 116–17), and Gazdar and Pullum (1985). Further details of the morphology of Swahili are to be found in Perrott (1951). ENG-MONOSYL is loosely based on the English phonotactics to be found in Whorf (1940).

The applicability of FSTs in phonology was noticed almost immediately. Johnson (1972) proved that a phonology that permitted only simultaneous rule application, as opposed to iterative derivational application, was equivalent to an FST. A decade later, Kaplan and Kay presented a paper in which they showed how the iteratively applied rules of standard generative phonology could, individually, be algorithmically compiled into FSTs. The closest thing to a published account of this work can be found in Kay (1983, pp. 100–4). Kaplan and Kay's work directly influenced that of Koskenniemi (1983a, 1983b, 1984) which, in turn, led to that of Karttunen and his students on the KIMMO system (Gajek *et al.* 1983, Khan *et al.* 1983, and Karttunen 1983). The Koskenniemi-style approach is described and evaluated in Gazdar (1985). Finite-state phonology or morphology fragments exist for Arabic (Kay 1987), Japanese (Alam 1983), Rumanian (Khan 1983), French (Lun 1983), German (Görz and Paulus 1988), Semitic (Kataja and Koskenniemi 1988), Spanish (Meya 1987), Old Church Slavonic (Lindstedt 1984), Swedish (Blåberg 1985), Turkish (Hankamer 1986) and English (Karttunen and Wittenburg 1983, Russell *et al.* 1986, Black *et al.* 1987). Some interesting complexity results relevant to this approach are presented in Barton *et al.*(1987, Chapters 5 and 6), to which Koskenniemi and Church (1988) replies. Further theoretical developments are reported in Bear (1988), Carson (1988), and Reape and Thompson (1988). The potential role of finite-state parsing in speech recognition is the subject of Church (1983a), while Gibbon (1987) proposes its use for the processing of tone systems.

Good introductions to the notion of search spaces and search techniques are given by Raphael (1976), Nilsson (1980, Chapter 1), Barr and Feigenbaum (1981, Chapter II), Rich (1983, Chapters 2–4), Charniak and McDermott (1985, Chapter 5), Bratko (1986, Chapters 11–13) and Winston (1984, Chapter 4). Pearl (1984) provides a definitive reference on the topic.

CHAPTER 3

Recursive and Augmented Transition Networks

3.1 Recursive transition networks **64**
3.2 Modelling recursion in English grammar **68**
3.3 Representing RTNs in Prolog **72**
3.4 Traversing RTNs **73**
3.5 Implementing RTN traversal in Prolog **79**
3.6 Pushdown transducers **82**
3.7 Implementing pushdown transducers in Prolog **87**
3.8 Advantages and limitations of RTNs **89**
3.9 Augmented transition networks **91**
3.10 Some reflections on ATNs **95**

We saw in the last chapter how FSTNs and FSTs can capture patterns of language and be used for responding to and producing language. We also implied, however, that FSTNs cannot capture, or cannot capture elegantly, some aspects of natural languages that we would like to be able to handle in NLP. For instance, FSTNs are not mathematically adequate for the description of certain kinds of embedded structures. There are also questions of notational adequacy, to which we return in this chapter, as we investigate how the network metaphor can be taken further to allow for the description of recursive structures in natural languages. Recursive transition networks and pushdown transducers are introduced as the recursive analogue of FSTNs and FSTs. The development of network-based machines for language processing has culminated in the design of various kinds of augmented transition networks, which essentially provide us with specialized programming languages for writing language processing applications.

3.1 Recursive transition networks

If you have access to computer mail facilities, you may occasionally get annoyed with people who besiege you with irrelevant messages and seem to expect a sensible reply each time. You may have speculated that, for people whose messages are particularly predictable and uninteresting, the process of producing replies might be automated in some way. Such an automated reply generator would not have to be very sophisticated, but it would have to respond robustly to any of a large number of potential natural language inputs, and the responses would have to be fairly plausible. Thus, such a system might be designed to respond to particular patterns in the input and produce responses that are related to them. For instance, a general rule like:

> I have discovered a new bug in … .
> ⇒ Yes, I found that one last week.
> But I do not have time to fix … just now.

would enable the system to respond to a number of different inputs. So if the input was 'I have discovered a new bug in *the operating system*,' the response would be 'Yes, … but I do not have time to fix *the operating system* just now.' A simple program along these lines, called ELIZA, was written by Joseph Weizenbaum in the 1960s. ELIZA was an attempt to reproduce some of the conversational abilities of a non-directive psychologist. It is reputed to have fooled some of its conversational partners into thinking that they were interacting with another human being.

Given the machinery of FSTs developed in the last chapter, a strategy for implementing our reply generator immediately suggests itself. We simply encode the different stimulus–response rules in one giant FST that transduces possible input sentences into possible outputs. Once we start to build an FST with the kind of complexity required by a system like ELIZA, however, we encounter certain problems. To be able to copy arbitrary parts of the input into the output – for instance, the name of the computer program in the foregoing rule – we need to be able to specify an abbreviation that stands for any pair of identical legal symbols. Thus:

```
EQUAL abbreviates:
  a_a, about_about, acorn_acorn, … .
```

for each possible word in the English language. While it would be tedious to specify such an abbreviation in our current notation, it would be simple to make our recognition programs deal with this identity mapping specially.

Less tractable are problems to do with producing patterns, which have the right degree of specificity without having the size of the FST explode drastically. Let us concentrate for the moment purely on the

problem of recognizing instances of patterns for which certain responses are plausible. We will come back to the general transduction issues later. Consider, for instance, the problem for someone who wishes to write an ELIZA program that will give the following responses:

> everyone hates me
> ⇒ who do you think hates you?
> everyone dismisses me when i ask a question
> ⇒ who do you think dismisses you?

but will not produce the following:

> everyone with me is so stupid
> ⇒ who do you think with you?
> everyone except me is happy
> ⇒ who do you think except you?

The desired responses seem to be appropriate when the input is of the form 'everyone … me …', but only when the first slot is filled with an English verb, such as 'hates' or 'dismisses'. We can account for the fact that the second two responses are inappropriate by the fact that 'with' and 'except' are not verbs. In an FSTN to recognize this pattern, this regularity appears as follows:

```
Name EVERYONE-ME:
  Initial 1
  Final 4
  From 1 to 2 by everyone
  From 2 to 3 by VERB
  From 3 to 4 by me
  From 4 to 4 by ANYWORD.

VERB abbreviates:
  hates, dismisses, likes, … .
ANYWORD abbreviates:
  a, about, acorn, … .
```

where ANYWORD is an abbreviation that covers every English word. Now let us try to generalize this. For, actually, in the pattern 'everyone … me …', we ought to allow for the first slot to be filled with more than one word, as the following examples show:

> everyone will punish me
> ⇒ who do you think will punish you?
> everyone has forgotten me
> ⇒ who do you think has forgotten you?

The obvious solution is to include a whole bunch of connected arcs between 2 and 3, instead of the single verb arc. This will encode the notion of 'valid sequence of verbs'. A fairly simple-minded version might involve the following arcs in the bunch:

From 2 to 21 by has
From 21 to 3 by V-PERF
From 2 to 22 by will
From 22 to 3 by V-BASE
From 2 to 3 by V-PRES

where:

V-PERF abbreviates:
  forgotten, punished, loved, ...
V-BASE abbreviates:
  forget, punish, love, ...
V-PRES abbreviates:
  forgets, punishes, loves ... .

This seems quite reasonable, but consider what happens when we want to allow our program to respond to other patterns such as the following:

everyone will love you
  ⇒ who do you think will love me?

Because the pattern for 'everyone ... you ...' also involves sequences of verbs, when we encode it as a network we will have to go through the construction of a bunch of arcs that looks just like the ones we have just developed for the first pattern. Indeed, we may need to have a copy of these arcs in many places in a complete ELIZA system. The FSTN notation is not being as useful here as it might be. Quite apart from forcing us to develop networks that are quite large for relatively uncomplicated patterns, it has not given us any way to think about the 'verb sequence' bunch of arcs as a single object. In particular, if we subsequently decide to refine our notion of valid sequence of verbs, we will have to make changes in all the places where this bundle of arcs appears in the networks. In our ELIZA program, we are thus encountering in a much more serious way the problem of notational inadequacy, which we noted briefly in connection with our transducer SWAHILI-2 at the end of Chapter 2.

A fundamental advance is made by using *recursive transition networks (RTNs)* instead of FSTNs. Basically, RTNs are just like FSTNs except that they introduce the extra concept of a named subnetwork. That is, it is possible for an arc to name a subnetwork to be traversed, instead of a specific word (or class of words) that is to appear. The idea is that if we have a commonly used bunch of arcs, we can express this abstraction by

making it into a self-contained, named network. This network can then be referenced *by its name* in a network that needs it, rather than having to appear expanded out in every place. Note that we have not used the names of networks at all so far. In contrast to the practice in this book, some authors name a subnetwork by the initial node that the traversal is to start from. This allows the network writer to choose one of several initial nodes as being appropriate for a particular use of a network, but has the disadvantage that node names have to be globally accessible.

Just as an FSTN can be regarded as a specification of an FSA, so an RTN can be regarded as a specification of a machine, a *pushdown automaton (PA)*. Informally, in an RTN, to traverse an arc that is labelled with a subnetwork name instead of a word, it is necessary to traverse the subnetwork named, but remembering where to resume when that has been done. A pushdown automaton is an FSA that is equipped with an extra memory, a *stack*, that can be used for this purpose. We will examine the role of a stack in RTN traversal later in this chapter. For an RTN, network traversal is defined partially in terms of itself. This is the reason for the word 'recursive' in recursive transition network. We will see in a later section that, although this informal definition looks dangerously circular, it is possible to make computational sense of it.

Here is how our earlier finite-state ELIZA network fragments could be reconceived in RTN terms:

Name EVERYONE-ME:
  Initial 1
  Final 4
  From 1 to 2 by everyone
  From 2 to 3 by VERB-GROUP
  From 3 to 4 by me
  From 4 to 4 by ANYWORD.
Name VERB-GROUP:
  Initial 1
  Final 2
  From 1 to 11 by has
  From 11 to 2 by V-PERF
  From 1 to 12 by will
  From 12 to 2 by V-BASE
  From 1 to 2 by V-PRES.

ANYWORD abbreviates:
  a, about, acorn, ...
V-PERF abbreviates:
  forgotten, punished, loved, ...
V-BASE abbreviates:
  forget, punish, love, ...
V-PRES abbreviates:
  forgets, punishes, loves ... .

Having the VERB-GROUP network mentioned in patterns like EVERYONE-ME is an asset because it allows several words to appear between the 'everyone' and the 'me', but eliminates the problems otherwise caused by strings like:

> everyone who Mayumi saw me with ...

Note that we could absorb our convention for abbreviations into the more general notion of named subnetworks. But, in computational practice, we want a better way of dealing with abbreviations than thinking of them as networks, whether explicitly or implicitly present.

In RTNs, as we have defined them, a (sub)network is only referred to outside of itself by its name, and therefore it does not matter if a node name that is used in one network is also used in another. For instance, there is a node '1' in each of the two networks shown.

Exercise 3.1 Produce an improved VERB-GROUP subnetwork, which accepts strings like 'has been seen', 'will not eat', 'is still waiting', 'will have been seen', and so on. [*easy*]

3.2 Modelling recursion in English grammar

The model of English usage suggested by our computer mail answerer is a very piecemeal and *ad hoc* one. It suggests that often the way we use language is simply by responding to particular, special-purpose patterns with stereotyped responses. Although this may sometimes be true, it gives us little information about how people understand and respond to sentences that they have never seen before or the basic principles that lie behind efficient language use. In this section, we will begin to develop a more general RTN for a small fragment of English. This will be unsatisfactory in a number of ways, but will be useful as a running example in the discussions to follow. To keep things simple, we will ignore the intricacies of verb groups suggested by our previous examples.

To start with, we need some abbreviations corresponding to some basic lexical categories used in the description of English syntax. We have seen N, NP and DET in the ENGLISH-1 network of Chapter 2. The symbol V is used here to stand for English verbs and the symbol WH for English relative pronouns like 'who' and 'which'. Secondly, we need to define some larger networks in terms of these categories.

EXAMPLE: RTN for a fragment of English

```
Name S:
  Initial 0
  Final 2
  From 0 to 1 by NP
  From 1 to 2 by VP.
Name NP:
  Initial 0
  Final 2
  From 0 to 1 by DET
  From 1 to 2 by N
  From 2 to 3 by WH
  From 3 to 2 by VP.
Name VP:
  Initial 0
  Final 1, 2
  From 0 to 1 by V
  From 1 to 2 by NP
  From 1 to 3 by that
  From 3 to 2 by S.

N abbreviates:
  woman, house, table, mouse, man, ... .
NP abbreviates:
  Mayumi, Maria, Washington, John, Mary, ... .
DET abbreviates:
  a, the, that, ... .
V abbreviates:
  sees, hits, sings, lacks, saw, ... .
WH abbreviates:
  who, which, that.
```

In this RTN, S is the network for English sentences. A sentence can be recognized by finding first a noun phrase (NP) and then a verb phrase (VP). The noun phrase at the beginning of a sentence – for example, a phrase like 'Mayumi' or 'the woman' – is the subject of the sentence. A noun phrase can include a relative pronoun (WH) introducing a qualifying verb phrase, which is a rather simple type of relative clause, as in 'the man who sings'. The verb phrase is sometimes known as the predicate of the sentence and this contains a verb and possibly a noun phrase (the object of

the verb). It is also possible for certain verbs to be followed by the word 'that' and a sentential complement, as in 'thinks that Maria sings'. Thus, the above S network will recognize sentences like:

Mayumi sees the house.

Maria sings.

The table hits Washington.

Mayumi sees that Maria sings.

The table that lacks a leg hits Washington.

and also various other, less natural sounding, sequences of words. Note finally that the arc labels now have a dual status: they may simply stand for a set of items in the lexicon, as previously, they may just name a subnetwork, or they may do both. Thus, we can have items in the lexicon, such as 'Mayumi', listed as members of the category NP, or we can have an NP network, or we can have both.

This fragment demonstrates that English syntax is fundamentally recursive. For instance, it is easy to construct an English sentence that contains an English sentence, an English sentence that contains an English sentence that contains an English sentence ... and so on for as long as we like. But, of course, such sentences will become increasingly hard to understand as they get longer:

Mayumi says that Maria is a genius.

Mayumi says that Mayumi says that Maria is a genius.

Mayumi says that Mayumi says that Mayumi says that Maria is a genius.

The recursiveness of English, and most other natural languages, means that RTNs are a natural tool for expressing its regularities.

In our network, one way of traversing the S network involves traversing the VP network, which in turn involves traversing the S network again as a subtask. Once we have networks that display 'genuine' recursion in this way, we can see that RTNs are indeed more powerful than FSTNs with an abbreviatory convention for word categories. We could imagine a reference in an arc to a subnetwork as a shorthand for having a copy of that network in that position instead of the arc. This would explain why traversing the arc involves traversing the whole subnetwork. This shorthand model works nicely for examples like the VERB-GROUP subnetwork, but will not work when the definition of a phrase (directly or indirectly) includes itself. When this happens, the model would claim, in effect, that the finitely stated RTN is shorthand for an *infinite* network (Figure 3.1).

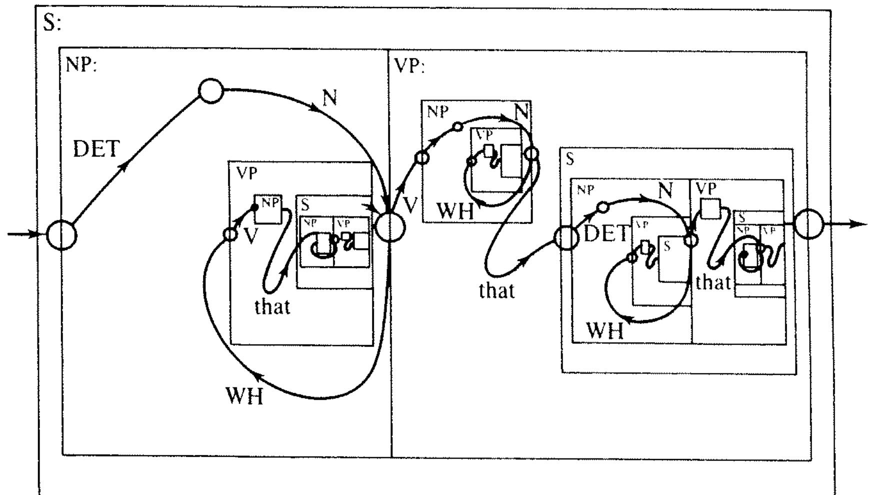


Figure 3.1 Expanded RTN for English.

Exercise 3.2 Write an FSTN that accepts all the 'Mayumi says' sentences in the infinite sequence described in the text. Can you write an FSTN that accepts all the sentences in the following sequence:

The man chants.

The man who the woman sees chants.

The man who the man who the woman sees sees chants.

but which does not accept 'unbalanced' strings like:

*The man who the man who the woman sees sees

*The man who the man who the woman sees sees sees chants

Note the use of '*' to signal ungrammaticality. [*intermediate*]

Exercise 3.3 Extend the example RTN so that it accepts the following sentences:

Lee knew Kim appointed the chairman.

Lee presented the prize to the author.

The chairman died.

but so that it does not accept the following:

*Lee knew Kim to the author

*Lee presented

*Lee died Kim appointed the chairman

[*easy*]

3.3 Representing RTNs in Prolog

Since an RTN consists of a set of networks, as opposed to the single monolithic FSTN, the data structure we use to represent the RTN needs to have a way of indicating which bits belong to which subnetwork. If we think of each subnetwork having a name, then we can add a further argument to each predicate of our existing FSTN data structure and use this argument to indicate which component of the network the clause belongs to. This leads to a predicate of the form:

```
arc(DEPARTURE_NODE, DESTINATION_NODE, LABEL,
    SUBNETWORK).
```

The way this works can be readily seen from our simple RTN for English sentences, which comprises three subnetworks, one for S, one for NP and one for VP (the code for these networks is repeated as **rtnarcs.pl** in the appendix):

```
% 0 is initial state of the S network:
initial(0, s).
% 2 is final state of the S network:
final(2, s).
% The S network contains an arc from 0 to 1 with label NP:
arc(0, 1, np, s).
% The S network contains an arc from 1 to 2 with label VP:
arc(1, 2, vp, s).
initial(0, np).
final(2, np).
arc(0, 1, det, np).
arc(1, 2, n, np).
arc(2, 3, wh, np).
arc(3, 2, vp, np).
initial(0, vp).
final(1, vp).
final(2, vp).
arc(0, 1, v, vp).
arc(1, 2, np, vp).
arc(1, 3, that, vp).
arc(3, 2, s, vp).
```

Notice how each subnetwork has its own **initial** and **final** statements in addition to the set of arcs that define it. Also observe how the same integer can represent different nodes; thus, each subnetwork has a node **2**, but the 2-node in the S network is wholly unrelated to the 2-node in the VP network.

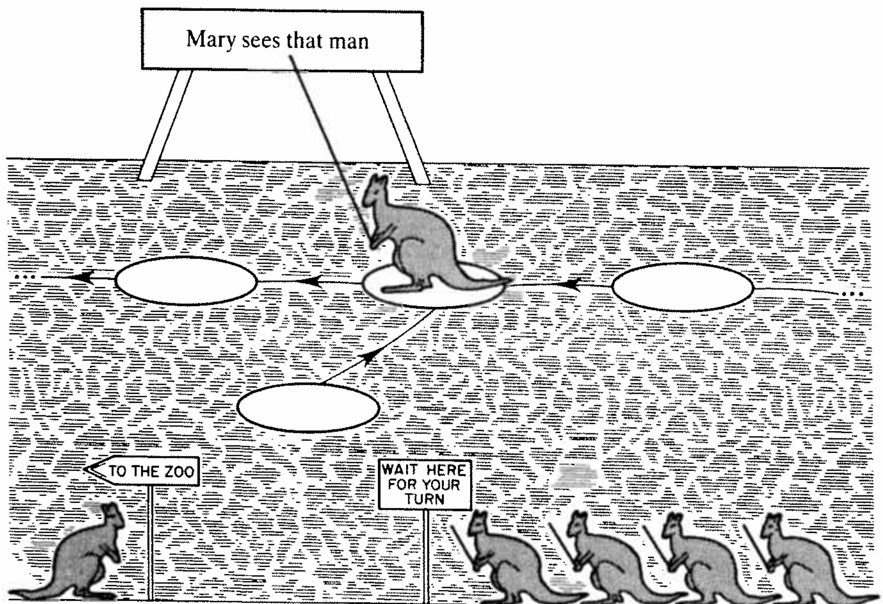


Figure 3.2 Kangaroos traversing an RTN.

3.4 Traversing RTNs

Although introducing named subnetworks seems conceptually like a small change to make to FSTNs, it does introduce significant complexity in the procedure for traversing a network. The basic problem is that, to traverse one arc, it may be necessary to traverse a whole subnetwork. While that subnetwork is being traversed, the position of the original arc must be remembered, so that the traversal can resume there afterwards. Our kangaroo model of network traversal, introduced in Chapter 2, needs to be embellished somewhat to account for this.

In a recursive network, a kangaroo may come across a subnetwork name on an arc, instead of a word. In this case, we can think of it as recruiting another kangaroo (from a large labour pool) to traverse the subnetwork for it. As this second kangaroo jumps through the subnetwork, it is allowed to move the pointing device along the input string. It is also allowed to recruit other kangaroos to traverse sub-subnetworks for it (and the levels of delegation may go arbitrarily deep). When any kangaroo has finished, it reports success to the kangaroo that recruited it, who waits patiently until this happens. So, when the second kangaroo has finished jumping through its subnetwork it reports back to the first kangaroo. The first kangaroo wakes up again, noticing that the pointer has been moved

on, finishes jumping along the arc mentioning the subnetwork and continues from the new node as before.

In the kangaroo model, at any time, there is one kangaroo jumping through a network and a line of other kangaroos, each remembering a place in a network and waiting to resume work once its subordinate has finished, as illustrated in Figure 3.2. The first of the waiting kangaroos (the one who joined the line last) is waiting for the currently active kangaroo, the second waiting kangaroo is waiting for the first, the third waiting kangaroo is waiting for the second, and so on. If the active kangaroo gets to the end of its network, it wakes up the first waiting kangaroo and then rejoins the labour pool. This first waiting kangaroo leaves the line and resumes work. When it finishes, it wakes up the first kangaroo in the line (who was previously the second) and itself goes back to the labour pool. On the other hand, if the active kangaroo finds a subnetwork that needs traversing, it recruits a new kangaroo from the labour pool (*not* the waiting line), sets this kangaroo off on the subnetwork and itself joins the waiting line, at the front, waiting for the subcontracted work to be complete.

The main innovation here is the line of waiting kangaroos. How are we to represent this computationally? In fact, the line of waiting kangaroos corresponds to the computational device of a *pushdown stack*. A pushdown stack is a device for storing information that can be manipulated by two basic operations:

(1) A piece of information can be PUSHed (added) on to the stack. This corresponds to a kangaroo joining the line at the front.

(2) The most recently added piece of information can be POPped (removed) from the stack. This corresponds to the first kangaroo in the line being woken up, leaving the line and resuming work.

When we need to indicate the contents of a pushdown stack, we will simply separate the items with colons, the first (most recently added) item being on the left. We will denote the empty stack by empty space. So here is how we would denote the stack that results from 'pushing' the symbols a, b and c on to the empty stack in that order:

c : b : a :

While a kangaroo is waiting in the line, it only needs to remember its place in a network, so that it can resume there afterwards. All other information can be picked up when it starts work again – the networks themselves will not have changed, and the pointer will have changed in ways completely outside its control. Thus, the pieces of information to be stored in the pushdown stack are simply positions in the networks. We can represent a position in a network by the name of the node where the kangaroo will resume work (the destination of the arc that requires the traversal of the subnetwork). In general, if we allow the same node name to be used in different networks, we require the name of the network containing the node as well. Where it is necessary to distinguish which network a node belongs to, we will qualify the name in the following fashion:

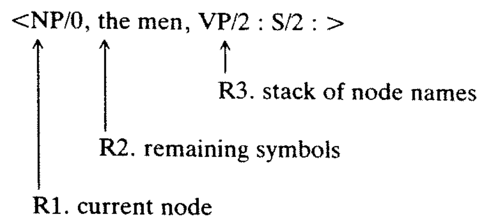
S/3

which denotes the node named 3 in the network S.

If we wish to extend our previous account of network traversal to RTNs, the notion of 'state' will have to be extended now to include a third component:

- R3. a pushdown stack of positions (node names).

Here, then, is a fully specified example state describing a point in an RTN traversal (recognition, not generation):



This state represents the intermediate state of a traversal where the remaining words to be considered are 'the men' and the active kangaroo is sitting on node 0 in the network NP. The line of waiting kangaroos contains kangaroos waiting to continue moving from node 2 in network VP (at the front of the line) and node 2 in network S (at the back of the line).

Given this modified notion of 'state', the basic search algorithm for network traversal does not need to be different from the finite-state case. This time each of the initial set of states needs to have an empty stack. Thus, since 0 is the only initial node in the top-level network for English sentences, the only initial state for traversing this network would be:

<S/0, ..., >

where '...' is the input string. What are the valid final states that indicate successful traversal? To be successful, we must have reached a final node in the top-level network, we must have an empty line of waiting kangaroos and we must have used all the symbols in the input. Thus, if we were using our networks for English sentences, the only possible final state would be:

<S/2, , >

How do we calculate the valid next states that follow from a given state <NODE, INPUT, STACK>. For each arc (with label L and destination

D) leaving node NODE, we must include states in the set according to the following rules. We illustrate these with examples from the foregoing example networks:

(POP) if NODE is a final node,
the top of STACK is H and
the rest of STACK is T,
include <H, INPUT, T>
e.g., from <NP/2, sings, S/1 : > we can get
<S/1, sings, >

(SYM′) if L is a symbol and
the first symbol of INPUT is the same,
include <D, ... rest of INPUT ..., STACK>
e.g., from <VP/1, that John, S/2 : > we can get
<VP/3, John, S/2 : >

(ABB′) if L is an abbreviation
which covers the first symbol of INPUT,
include <D, ... rest of INPUT ..., STACK>
e.g., from <NP/1, man sings, S/1: > we can get
<NP/2, sings, S/1 : >, since 'man' is an N

(PUSH) if L is a network name,
for each initial node INIT of that network
include <INIT, INPUT, D : STACK>
e.g., from <VP/1, the man, S/2 : > we can get
<NP/0, the man, VP/2 : S/2: >

(JMP′) if L is #, include <D, INPUT, STACK>

Of these principles, two are new and correspond to entering and leaving subnetworks (PUSH and POP). The others are essentially the same as in the finite-state network case.

Here is an example of recognition using our initial English networks and the sentence 'Mary sees that man.' We will show the stages of the recognition by a sequence of 'snapshots', each time showing the set of alternative states under consideration. Initially, we just have one state:

<S/0, Mary sees that man, >

Removing this from the set, only the PUSH rule is applicable and so we get:

<NP/0, Mary sees that man, S/1 : >

Selecting this, only the ABB′ rule applies (the first word is an NP):

<NP/2, sees that man, S/1 : >

From NP/2, the only arc requires a WH word, and 'sees' is not of this category. Since NP/2 is a final node, the only option is then to POP to:

<S/1, sees that man, >

From S/1, there is no choice but to PUSH, looking for a VP:

<VP/0, sees that man, S/2 : >

Once again, there is no choice but to recognize the V and proceed (by ABB′) to VP/1:

<VP/1, that man, S/2 : >

From VP/1, there are now three possibilities. Since VP/1 is a final node, we can POP back to the S network. Alternatively, we can proceed to VP/3 by consuming the word 'that' (SYM′). Finally, we can try to PUSH for an NP. Thus, we have three possible next states:

<S/2, that man, >
<VP/3, man, S/2 : >
<NP/0, that man, VP/2 : S/2 : >

We now have to choose between these, removing one from the set and leaving the other two behind, to be reconsidered later if other possibilities do not work out. Let us say that we choose the first of them. Since there are no arcs leaving S/2, only the POP rule could be appropriate. This does not apply, however, as the stack is empty. So we cannot progress further from this state. Since the state is not a final state, we simply remove it from the list, leaving:

<VP/3, man, S/2 : >
<NP/0, that man, VP/2 : S/2 : >

Again, we have to choose which state to investigate next. Let us again choose the first one. From VP/3, we have no option but to PUSH for an S:

<S/0, man, VP/2 : S/2 : >
<NP/0, that man, VP/2 : S/2 : >

Again, selecting the first state in the list, we are forced to PUSH and get:

<NP/0, man, S/1 : VP/2 : S/2 : >
<NP/0, that man, VP/2 : S/2 : >

If we this time select the second state to investigate, we can proceed to:

<NP/0, man, S/1 : VP/2 : S/2 : >
<NP/1, man, VP/2 : S/2 : >

(because 'that' is a DET). Similarly, we can progress from this state to a similar state at NP/2:

<NP/0, man, S/1 : VP/2 : S/2 : >
<NP/2, , VP/2 : S/2 : >

If we continue selecting the second state in the set, we can POP to:

<NP/0, man, S/1 : VP/2 : S/2 : >
<VP/2, , S/2 : >

and then POP again to:

<NP/0, man, S/1 : VP/2 : S/2 : >
<S/2, , >

The second state in our list is now a successful final state, and so we can halt, having successfully recognized the string.

Here is an example of random generation from the networks, giving the intermediate states and the rule used at each stage. As in the FSTN case, for generation, we keep the list of words output so far, rather than the list of words to be processed in a state. The rules for generating next states have to be altered slightly to cater for this change. Since the generation is random, we only need to consider one alternative at each point and can discard the others.

<S/0, , >
PUSH
<NP/0, , S/1 : >
ABB′ (NP)
<NP/2, John, S/1 : >
POP
<S/1, John, >
PUSH
<VP/0, John, S/2 : >
ABB′ (V)

<VP/1, John saw, S/2 : >
PUSH
<NP/0, John saw, VP/2 : S/2 : >
ABB′ (DET)
<NP/1, John saw the, VP/2 : S/2 : >
ABB′ (N)
<NP/2, John saw the mouse, VP/2 : S/2 : >
POP
<VP/2, John saw the mouse, S/2 : >
POP
<S/2, John saw the mouse, >
SUCCESS!

Exercise 3.4 Why does it not matter for the final outcome which choices we make at which points in the recognition example that we walked through in the text? Does this apply in general? [*intermediate*]

Exercise 3.5 Write down a revised set of next state rules for RTNs, oriented towards generation, rather than recognition. [*easy*]

3.5 Implementing RTN traversal in Prolog

We will now present a modified version for RTNs (appearing as **rtnrecg.pl** in the appendix) of the FSTN recognizer program. The program harnesses Prolog's backtracking ability to perform a depth-first search of the search space, as before. In addition, it uses recursion in Prolog to implement recursive network traversal and hence is able to dispense with an explicit pushdown stack.

The RTN recognizer, on several occasions, uses pairs of arguments as difference lists to represent strings. Although initially somewhat unintuitive, difference lists are an efficient way of manipulating lists of symbols in Prolog, and they play a crucial role in the interpretation of the definite clause grammar formalism that we will be examining in detail in chapter 4. Readers unfamiliar with difference lists should probably take time out at this point to review the coverage of the topic in a Prolog text. Each main predicate responsible for recognition is given two arguments concerned with sequences of words (lists). The first of the pair of arguments represents an initial list, some initial portion of which is to be recognized. The second of the pair represents the end portion of the first list which is 'left

over' after the recognition. When such a predicate is invoked during recognition, its task is to find possible initial portions of the first list that can be recognized, 'consume' the relevant words from the list and return what remains as the left-over list.

As might be expected, implementing RTN traversal is slightly more complex than implementing FSTN traversal since, in the former, we need to keep track of which network we are in, and we need some way of dropping down into subnetworks and of climbing back out again. As before, we shall split the recognition task beween two predicates, **traverse/3** and **recognize/4**. The former is a suitably modified version of our previous FSTN **traverse** predicate while the latter is similar to the two-argument **recognize** predicate, but with the addition of an extra argument to keep track of the name of the current network and an extra argument to keep track of any string left over after the traversal:

```
% recognize(Net, Node, WordList, Left)
%   Net: the name of the network to be traversed
%   Node: the name of the node you wish to start from
%   WordList: the list of words you want to test
%   Left: the list of words left over after the traversal
```

Notice that we have chosen to use two arguments to represent the initial location – the network and the node within it – rather than combining them into a single object, as would be directly suggested by our Net/Node notation. As with the FSTN recognizer, the predicate **recognize** has two clauses. Firstly, if we have reached the final node of the (sub)net, we can successfully recognize without consuming any of the string:

```
recognize(Net, Node, X, X) :-
  final(Node, Net).
```

Alternatively, we want to recognize a string X if:

(1) We can consume part of the string, traversing an arc from the current node, and leaving string Y.

(2) We can recognize the remainder Y, continuing from where that arc ended up.

The string Z left over from such a recognition is then whatever was left over by step (2):

```
recognize(Net, Node_1, X, Z) :-
  arc(Node_1, Node_2, Label, Net),
  traverse(Label, X, Y),
  recognize(Net, Node_2, Y, Z).
```


Thus, **recognize** delegates part of the job to **traverse**, and it is to the latter that we turn next:

```
% traverse(Label, WordList, Left)
%   Label: an arc label specifying a kind of test
%   WordList: the list of words you want to test
%   Left: the list of words left over after the traversal of the arc
```

There are four possibilities for success, depending on the kind of test the arc label specifies on the input string. First of all (SYM′), if the label is not a special symbol, we can consume one word from the string if it is the same as the arc label, returning the rest of the string:

```
traverse(Word, [Word | X], X) :-
  not(special(Word)).
```

Second (ABB′), if the label specifies an abbreviation, we can consume one word from the string if it is an instance of the appropriate category, returning the rest of the string:

```
traverse(Category, [Word | X], X) :-
  word(Category, Word).
```

Third (PUSH/POP), if the arc label names a subnet, we can traverse the arc by traversing the named network:

```
traverse(Net, String, Left) :-
  initial(Node, Net),
  recognize(Net, Node, String, Left).
```

Notice that when we recursively traverse a subnetwork, we do not need explicitly to remember where to return to afterwards. This is because if the recursive call to **recognize** succeeds, Prolog will automatically continue with whatever remains to be done after that (succeeding the **traverse** goal and then continuing in the **recognize** clause that called it). So the PUSH/POP operations are conveniently implemented by Prolog's call/return mechanism. The final clause for **traverse** (JMP′) handles the case where the label on the arc is the jump symbol. In this case, we can succeed without consuming any string:

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

Between them, the mutually recursive definitions of the predicates **traverse** and **recognize** handle the problem of RTN traversal in a compact and elegant manner.

3.6 Pushdown transducers

We saw in the last chapter how an FST is simply an FSA that deals with two tapes. Similarly, we can construct a *pushdown transducer (PT)* as a pushdown automaton that deals with two tapes. To specify a PT, it suffices to augment the RTN notation so that labels on arcs can denote pairs of symbols, just as in the finite-state case. In general, a label in a PT (or, to be precise, an RTN that is interpreted as a PT) may specify any of the following:

- a symbol to be found on each tape – for example, fish_poisson – where either or both symbols may be #.
- an abbreviation for a set of such symbol pairs.
- the name of another PT to be traversed.

With the extra power of RTNs, we can write interesting transducers quite easily. For instance, an ELIZA PT can be constructed from the previous RTN translated into a transducer.

EXAMPLE: Eliza PT

```
Name ELIZA:
  Initial 1
  Final 2
  From 1 to 2 by EVERYONE-ME
  From 1 to 2 by ...
  ... many more arcs ... .
Name EVERYONE-ME:
  Initial 1
  Final 4
  From 1 to 2 by everyone_'who do you think'
  From 2 to 3 by VERB-GROUP
  From 3 to 4 by me_you
  From 4 to 4 by EQUAL.
Name VERB-GROUP:
  Initial 1
  Final 2
  From 1 to 11 by has_has
  From 11 to 2 by V-PERF.
  ... .
EQUAL abbreviates:
  a_a, about_about, acorn_acorn, ... .
V-PERF abbreviates:
  forgotten_forgotten, punished_punished, hated_hated, loved_loved, ... .
```

It is similarly easy to produce a simple PT for limited English–French translation. Apart from the alterations to deal with gender in noun phrases, this PT is a direct translation of our previous RTN for English sentences.

EXAMPLE: PT for limited English–French translation

```
Name S:
  Initial 0
  Final 2
  From 0 to 1 by NP
  From 1 to 2 by VP.
Name NP:
  Initial 0
  Final 2
  From 0 to 1 by DET-FEMN
  From 1 to 2 by N-FEMN
  From 0 to 4 by DET-MASC
  From 4 to 2 by N-MASC
  From 2 to 3 by WH
  From 3 to 2 by VP.
Name VP:
  Initial 0
  Final 1, 2
  From 0 to 1 by V
  From 1 to 2 by NP
  From 1 to 3 by that_que
  From 3 to 2 by S.
N-MASC abbreviates:
  man_homme, horse_cheval, ... .
N-FEMN abbreviates:
  house_maison, table_table, ... .
```

NP abbreviates:
  John_Jean, Mary_Marie, Jean_Jeanne, ...

DET-MASC abbreviates:
  a_un, the_le, this_ce, ...

DET-FEMN abbreviates:
  a_une, the_la, this_cette, ...

V abbreviates:
  sees_voit, hits_frappe, sings_chante, lacks_manque, ...

WH abbreviates:
  who_qui, which_qui, that_qui.

Note that, when used to produce French output from English input, this system is doing little more than word-by-word translation, and it is easy to find examples where this will not work for French or other languages. On the other hand, the fact that the transducer incorporates a notion of what is a grammatical sentence in each language means that it could be used for other purposes – for instance, generating random pairs of legal and equivalent sentences.

Given that an RTN has successfully recognized a sentence as grammatical in some language, we might be interested to see which route the RTN interpreter took through the network for this sentence. That is, we might be interested to see a *trace* of which networks have been entered and exited, which word categories have been found when, and so on. One possibility would be to augment our RTN interpreter to keep track of this information and display it at the end of recognition. Alternatively, we could construct a transducer that produces this information as its output.

EXAMPLE: Using a PT to record a path through an RTN

```
Name S:
  Initial 100
  Final 200
  From 100 to   0 by #_(S
  From   0 to   1 by NP
  From   1 to   2 by VP
  From   2 to 200 by #_).
```



```
Name NP:
  Initial 100
  Final 200
  From 100 to   0 by #_(NP
  From   0 to   1 by DET
  From   1 to   2 by N
  From   2 to   3 by WH
  From   3 to   2 by VP
  From   2 to 200 by #_).
Name VP:
  Initial 300
  Final 100, 200
  From 300 to   0 by #_(VP
  From   0 to   1 by V
  From   1 to   2 by NP
  From   1 to   3 by that_that.
  From   3 to   2 by S
  From   1 to 100 by #_)
  From   2 to 200 by #_).
```

N abbreviates:
  man_N, house_N, table_N, ...

NP abbreviates:
  Mayumi_NP, Maria_NP, Gilbert_NP, ...

DET abbreviates:
  a_DET, the_DET, that_DET, ...

V abbreviates:
  sees_V, hits_V, sings_V, lacks_V, ...

WH abbreviates:
  who_WH, which_WH, that_WH.

This transducer, which is again derived straightforwardly from our English sentence RTN, can be used to transduce a sentence into a trace showing the networks that the RTN would traverse in accepting it. Each network has been augmented with extra arcs at the beginning and end (one for each initial and final state), which produce output on the second tape recording entry to and exit from that network. A symbol like '(S' on the output indicates entry to the S network; the symbol ')' on the output indicates exit from the network last entered. In addition, we have each instance of an abbreviation or particular word encountered leave behind a trace of its occurrence. Thus, for instance, the input sentence 'Mayumi sees

that the man sings' would produce essentially the following output on the second tape:

(S (NP NP) (VP V that (S (NP DET N) (VP V))))

At present, our networks are so simple that it is not possible for a sentence to be recognized in more than one way. Hence, this transducer will always generate exactly one trace for each legal sentence. We can change this by allowing prepositional phrases in our sentences. At its simplest, a prepositional phrase consists of a preposition followed by a noun phrase – for instance, 'with the funny hat' and 'behind Washington'. A noun phrase can have any number of prepositional phrases at its end:

The man with the funny hat behind Washington

In this case, the prepositional phrases are telling us more about whoever it is being described by the noun phrase. Prepositional phrases can also occur at the end of verb phrases to indicate where, when or how the action takes place, as in:

Maria sang with the choir in the large room.

Sometimes it is not clear whether a given prepositional phrase belongs to a noun phrase or to a verb phrase, as in:

Maria saw the woman with the telescope.

This ambiguity corresponds to different ways in which this could be recognized as a sentence.

Exercise 3.6 Add rules to the transducer just given to allow both a noun phrase and a verb phrase to include any number of prepositional phrases at its end. What outputs will the transducer produce for the sentences 'Maria sees the woman with the telescope' and 'Maria sees the woman in the park with the telescope'? [*easy*]

Exercise 3.7 Modify the example English–French network and lexicon so as to be able to translate the following sentences into French:

John gives a horse to the man.
The man knows John lacks a house.

Ensure that 'to the' will translate into 'au' rather than 'a le'. *Hint*: You may need to use a pair like word_# or #_word in your transduction lexicon. [*intermediate*]

3.7 Implementing pushdown transducers in Prolog

We need only make minimal changes to the recognition program to make a program that will transduce from one string of words to another. The main addition is another couple of arguments, representing the second tape by a difference list in the same way as the first:

```
% transduce(Net, Node, String1, Left1, String2, Left2)
%   Net: the network in which the traversal takes place
%   Node: the node to start from
%   String1: the first tape
%   Left1: the portion of the first tape left over
%   String2: the second tape
%   Left2: the portion of the second tape left over

transduce(Net, Node, X1, X1, X2, X2) :-
  final(Net, Node).
transduce(Net, Node_1, X1, Z1, X2, Z2) :-
  arc(Node_1, Node_2, Label, Net),
  traverse2(Label, X1, Y1, X2, Y2),
  transduce(Net, Node_2, Y1, Z1, Y2, Z2).

traverse2([Word_1, Word_2], [Word_1 | X1], X1, [Word_2 | X2], X2) :-
  not(special(Word_1)),
  not(special(Word_2)).
traverse2(Category, String_1, Left_1, String_2, Left_2) :-
  word(Category, Word_pair),
  traverse2(Word_pair, String_1, Left_1, String_2, Left_2).
traverse2(Subnet, String_1, Left_1, String_2, Left_2) :-
  initial(Subnet, Node),
  transduce(Subnet, Node, String_1, Left_1, String_2, Left_2).
traverse2(['#', W2], X1, X1, [W2 | X2], X2).
traverse2([W1, '#'], [W1 | X1], X1, X2, X2).
traverse2('#', X1, X1, X2, X2).
```

This procedure uses a **traverse2** predicate very similar to the one used for FST traversal. PTs are easily encoded in Prolog with the same techniques we have already for RTNs and FSTs. For instance, here is the French translation example:

```
% S network
%

initial(s, 0).

final(s, 2).

arc(0, 1, np, s).
arc(1, 2, vp, s).
```

```
% NP network
%

initial(np, 0).

final(np, 2).

arc(0, 1, det_femn, np).
arc(1, 2, n_femn, np).
arc(0, 4, det_masc, np).
arc(4, 2, n_masc, np).
arc(2, 3, wh, np).
arc(3, 2, vp, np).

% VP network
%

initial(vp, 0).

final(vp, 1).
final(vp, 2).

arc(0, 1, v, vp).
arc(1, 2, np, vp).
arc(1, 3, [that, que], vp).
arc(3, 2, s, vp).
```

As before, we have a lexicon associating categories with pairs of words, one in each language:

```
word(n_masc, [man, homme]).
word(n_masc, [horse, cheval]).
word(n_femn, [house, maison]).
word(n_femn, [table, table]).
word(np, [john, jean]).
word(np, [mary, marie]).
word(np, [jean, jeanne]).
word(det_masc, [a, un]).
word(det_masc, [the, le]).
word(det_masc, [this, ce]).
word(det_femn, [a, une]).
word(det_femn, [the, la]).
word(det_femn, [this, cette]).
word(v, [sees, voit]).
word(v, [hits, frappe]).
word(v, [sings, chante]).
word(v, [lacks, manque]).
word(wh, [who, qui]).
word(wh, [which, qui]).
word(wh, [that, qui]).
```


Exercise 3.8 Using the example PT and transducer code, which appear as **rtrans.pl** in the appendix, experiment with the following queries:

```
?-test([john, sings], French).
?-test(English, [marie, voit, un, cheval]).
?-test([jean, sees, a, house, that, lacks, a, table], French).
?-test([the, man, who, sings, sees, that, this, house, lacks, a, table],
    French).
```

Incorporate any additions you have made to previous RTNs and PTs for English sentences and check that the Prolog version runs correctly. [*easy*]

Exercise 3.9 Write a PT and abbreviation lexicon that will permit simple FSTNs in NATR notation (omitting abbreviation statements), but reduced to lists of atoms – for example, ['Name', 'TO-LAUGH-4', ':', 'Initial', 1, 'Final', 2, 'From', 1, to, 1, by, ha, 'From', 1, to, 2, by, '!'] – to be translated into Prolog notation in a similar format – that is, **[initial**, '(', **1**, ')', **final**, '(', **2**, ')', **arc**, '(', **1**, **1**, **ha**, ')', **arc**, '(', **1**, **2**, '!', ')']. What assumptions do you need to make about the abbreviation lexicon to get this to work? Can you relax these assumptions if you modify the transduction program itself? Why would the transduction be significantly harder if NATR RTNs were to be translated in a similar fashion? [*hard*]

Exercise 3.10 Complete an FSTN NATR compiler based on the program set in the previous exercise: it should read a file containing ordinary NATR notation into a list of atoms, transduce it as above, then assemble the Prolog terms and assert them into the Prolog database. [*hard, for programmers*]

3.8 Advantages and limitations of RTNs

We have demonstrated that RTNs have certain clear notational advantages over FSTNs. They allow commonly occurring subpatterns to be expressed as named subnetworks, and large networks to be built up in a modular way. In addition, they allow us to deal naturally with some of the recursive structures in natural languages. The result is a conceptual and notational system that is clean, clear and efficient. As we have seen, however, there is a price to be paid in the complexity of the network traversal algorithms, and we might still prefer FSTNs for a given application for this reason, even though the resulting networks might be larger.

As regards mathematical adequacy, we have illustrated some recursive language structures that can be recognized by RTNs but cannot in

principle be captured by FSTNs. The classical example showing the difference in power between the two notations is the language $a^n b^n$. Whereas it is impossible for an FSTN to recognize precisely the legal strings in $a^n b^n$, it is rather easy to construct an RTN that does. Here is another simple language for which the same is true. Assume that we have just two symbols, '(' and ')', and we want to recognize whether a string of such symbols has correctly matching brackets. Thus, the following strings will be legal:

```
()(())((()))
(()())
(())((())())
```

but the following will not:

```
)(()()
((())
(()))
```

Although it is possible to construct an RTN to recognize the set of legal strings in this language (it needs to use recursion in a non-trivial way), it is not possible to do the same with an FSTN. Syntactic constructions with this formal character are rather common in natural languages, although English fails to provide a really clearcut example. However, if you imagine a language just like English except that the 'then' in the 'if ... then ...' construction is obligatory, and this is the only usage of the word 'then', then you can get a sense of the phenomenon as it might occur naturally.

Are there sets of inputs that even RTNs cannot recognize? The answer is yes: while it is possible to write an RTN that will recognize precisely the strings of $a^n b^n$, it is not possible to write one for the language $a^n b^n c^n$ – that is, the language of strings consisting of n number of *a*s, followed by n number of *b*s, followed by n number of *c*s, for any n. Similarly, the language $a^m b^n c^m d^n$ is not amenable to recognition by an RTN.

Exercise 3.11 Write RTNs that recognize exactly the legal strings of (a) $a^n b^n$ and (b) the language of matching brackets described in the text. [*easy*]

3.9 Augmented transition networks

The foregoing examples show that certain kinds of outputs, such as sentences in other languages, can be computed from natural language sentences using transducers (PTs) built from RTNs. All the examples do, however, have a kind of family resemblance, in that the order in which items appear in the output echoes the order in which corresponding items appear in the input. Now, it is possible to write PTs that produce outputs in a *different* order from the corresponding inputs, but for such a situation we have to essentially include a different set of arcs for each possible input–output pair. When the input and output languages have large vocabularies, this leads to large networks, which cannot be expressed concisely even using our abbreviation mechanism. One solution to this problem would be to have a way of specifying how a *sequence* of outputs relates to a *sequence* of inputs, rather than having to specify input–output relations at the level of single arcs. Grammars, as presented in Chapters 4 and 7, provide a basis for describing sequences of phrases, and in Chapter 8 we will see how we can compute outputs from such grammars. Alternatively, we can keep to the network metaphor and enhance our networks with extra facilities to overcome the PT limitations. This is the approach taken with *augmented transition networks (ATNs)*.

Consider, for instance, what is required to enhance our translator to deal with more interesting noun phrases. In French, adjectives standardly follow the noun they modify, whereas in English they precede it. For instance, we would want 'a short name' to be translated to 'un nom court' (literally, 'a name short'). In fact, there are good reasons why we left out adjectives in our previous translators. If we try to extend the PT to include adjectives on the English side, we find that we have to write networks like the following:

```
Name NP:
  Initial 0
  Final 2
  From 0  to 1  by DET-MASC
  From 1  to 1a by short_#
  From 1  to 1c by green_#
  ...
  From 1a to 2a by N-MASC
  From 1b to 2b by N-MASC
  From 1c to 2c by N-MASC
  ...
  From 2a to 2  by #_court
  From 2c to 2  by #_vert
  ....
```

where there are three arcs for each possible English adjective (and yet this

network only deals with single adjectives!). The problem is that the French adjectives, which are known before the English noun is encountered, have somehow to be remembered and then produced on the second tape after the noun is translated. ATNs allow values to be remembered in this way during a network traversal by providing *registers* (variables) for storing information. Registers are rather like (local) variables in a procedural programming language. Thus, in our notation for ATNs, we will include a line declaring the registers used at the beginning of each network. Each arc of the network may then be annotated with instructions for how to shuffle information between these registers when it is traversed. Instructions can also be associated with the initial and final nodes of a network, specifying things to do when the network is entered or exited. In the latter case, the instructions will concern the single value that the network is allowed to return to the network that calls it, summarizing the result of its computations.

To make things more concrete, here is how we might tackle our noun phrase translation problem in an ATN. We will use the register FNP to keep track of the value (French phrase) to be returned by the NP network. In the French translation of a proper noun, this is simply whatever single French name corresponds to that English name, if there is one. In the French translation of a simple noun phrase, we need to have a French determiner, a collection of French adjectives and a French noun. We can introduce registers called FDET, FADJS and FNOUN to keep track of this information used in the translation of a noun phrase, and annotate the NP network informally as follows (for now, read '*' as meaning the current word):

```
Name NP:
  Registers FADJS, FNOUN, FDET, FNP
  Initial 0            set FADJS to the empty string
  Final 2              return FNP
  From 0 to 1 by DET   set FDET to French(*)
  From 1 to 1 by ADJ   set FADJS to FADJS + French(*)
  From 1 to 2 by N     set FNOUN to French(*)
                       set FNP to FDET + FNOUN + FADJS.
```

where + is a form of string concatenation that inserts spaces between words and French is a function that takes English words to their French translations, which we naively assume to be unique. The register FADJS is used here to hold a string that will translate a whole sequence of adjectives. As more English adjectives are discovered, their French translations are added to the end of the current value of this register.

Registers allow us to separate the times when parts of the output are computed from the position they occupy in the output. The idea is to recognize one structure but to be able to build as output a structure that may be only indirectly related to it.

We have assumed here that the French translation of any English word – that is, French(*) – can be easily computed. In general, ATN systems allow us to call arbitrary procedures defined in a standard programming language (often LISP) from the arcs of a network, and in an actual system this would probably be how this would be implemented. Note that the network could actually specify the category of the word to be translated, which would help when a given English word was ambiguous; for example, 'slight' could be a noun or an adjective, and would translate into 'manque d'égards' or 'léger' accordingly. Nevertheless, it is still very simplistic to assume that a word can be translated without any regard to the context, and no sensible machine translation system would be designed this way.

Usually in ATNs there is one special register, called '*', which automatically holds the value we are currently considering. When '*' is mentioned on an arc that is looking for a single word (either a specific word or a word covered by an abbreviation), it will always refer to the current word that the arc is looking at (as in the preceding network). On the other hand, when '*' is mentioned on an arc that introduces a PUSH to a subnetwork, it will always refer to the result returned by that subnetwork. This enables higher networks to deal with the result of lower networks. For instance, in our translation example, the S network needs to put together the French translations of the noun phrase and the verb phrase to make the translation of the whole sentence. Here is one way it could be done, using registers FSUBJECT and FPREDICATE to hold these values:

```
Name S:
  Registers FSUBJECT, FPREDICATE
  Initial 0
  Final 2              return FSUBJECT + FPREDICATE
  From 0 to 1 by NP    set FSUBJECT to *
  From 1 to 2 by VP    set FPREDICATE to *
```

Our translation rules for English noun phrases are inadequate in a number of ways, but one of the limitations provides grounds for introducing another feature of ATNs – extra tests on the arcs. The precise form of a determiner or adjective in French depends on the gender of the noun it qualifies, this being either masculine or feminine. Thus, we have:

a green tree → un arbre vert		(a tree green)
a green table → une table verte		(a table green)

Unfortunately, in our ATN we will not find out the gender of the noun phrase until the English noun is encountered, and by then we will have already generated the translation of the adjectives. One solution would be to actually keep the *English* determiner and adjectives in registers, and then translate them at the end of the phrase. A simpler solution, however, makes use of the trick used for French determiners in the ENG_FRE-2

FST, which relies on the fact that network traversal is organized as a search process, so that all possible traversals of the network will be found. We can outline two routes through the NP network, one corresponding to a masculine NP and the other corresponding to a feminine one. Each of these routes will produce adjectives with endings appropriate to the gender chosen and will then require there to be a noun of the appropriate gender at the end of the phrase. In any actual traversal, only one of them will lead to a successful solution, because the noun in the input string will have only one gender. As long as our implementation correctly searches through all possibilities, however, it will not matter whether it finds the correct path first or whether it starts off trying the possibility that fails. Because we have an ATN, rather than an FSTN, we can introduce a further twist and avoid duplicating all the NP arcs for the two different routes. Instead, we can use a register FGENDER to record the gender that we have chosen. So, the two different routes start with different arcs, setting the register to the two different values, but share exactly the same arcs after that. The arcs that the two routes have in common will then translate the determiner and adjectives on the basis of whatever value FGENDER has. To force the analysis to succeed only when the choice of FGENDER is the same as the gender of the French noun, we add a special test to the arc from 1 to 2:

```
Name NP:
  Registers FADJS, FDET, FNP, FNOUN, FGENDER
  Initial 0               set FADJS to the empty string
  Final 2                 return FNP
  From 0 to 1 by DET      set FGENDER to "masculine"
                          set FDET to French(*, "masculine")
  From 0 to 1 by DET      set FGENDER to "feminine"
                          set FDET to French(*, "feminine")
  From 1 to 1 by ADJ      set FADJS to FADJS + French(*, FGENDER)
  From 1 to 2 by N        set FNOUN to French(*)
                          the gender of FNOUN must be
                            the same as FGENDER
                          set FNP to FDET + FNOUN + FADJS.
```

where we now assume that the function French is provided with a gender as well as an English word, where this is required. Notice that, although we needed two arcs corresponding to the original choice of the gender, we did not need to duplicate the arcs for the adjectives and noun in the same way. The use of the register FGENDER enabled us to use each arc for both of the two alternatives. For this sort of approach to work, we obviously need a search mechanism that allows a register to have different values in different possible paths through the network. This has direct implications for the way one implements ATNs.

In our simple English–French translation example, we have seen how ATN registers can be used locally to reorder material to be output. The same idea can be used globally to keep track of phrases that appear

extraposed from their canonical position. For instance, each of the following questions is identical in form to the word 'who' followed by an ordinary statement, except that somewhere in the statement there is a gap (indicated by ?) where we would normally expect a noun phrase. This gap corresponds to the position of the item whose identity is being questioned:

Which employer [? will see Maria tomorrow]
Who [will Mayumi see ?]
Who [does Mayumi have a contract with ?]

In an ATN, it is relatively straightforward to remember (left) extraposed material (here, the questioned noun phrases) by assigning a value to a global register HOLD and to retrieve the relevant information again when an apparent gap is found.

3.10 Some reflections on ATNs

The main contribution of ATNs is to introduce the notion of registers, assignment and tests into network notations. Registers can be used to keep track of pieces of text, pieces of output structure or features (like gender) of lexical items. Using registers, general tests can be made on acceptability (for example, gender agreement) and output can be built in a flexible order. The use of registers enables what would be different paths through an RTN to be merged into one, and search to be avoided by storing information before its exact significance is known. In addition, the use of global registers provides a technique for handling phenomena like the non-local dependencies found in some English relative clauses and questions.

A few words on the history of ATNs. We have noted that there are consistent structural relations between, for instance, a statement and the corresponding yes–no question, and a yes–no question and the corresponding WH question. In linguistics in the 1960s, the development of Chomsky's transformational grammar was largely motivated by observations like these. Chomsky and his co-workers described a set of transformations that could be used to derive more complex sentences (for example, WH questions) from simpler ones (for example, very simple declarative sentences). Unfortunately, it proved computationally infeasible to undo or reverse these transformations in a principled way, so as to build a language analyzer that would take arbitrary sentences and understand them in terms of their putative source representations. Woods and others developed ATNs essentially as a programming language for writing analyzers where the undoing of transformations could be carried out during processing. Unfortunately, it is not in general easy to translate any given transformation into ATN code. Nevertheless, ATNs, typically running in a left-to-

right mode and using depth-first search, were probably the most prominent framework for natural language syntactic analysis in the 1970s.

Readers will hopefully have noticed how the flavour of ATNs is different from all the other formalisms we have discussed so far. Writing an ATN is much more like writing a computer program than writing either an FSTN or an RTN. When we write programs in conventional programming languages, a program will only do the task it is designed for. Similarly, our ATN for translating from English into French cannot be used (straightforwardly) for generating random English sentences or translating from French to English, say. With FSTNs and RTNs, we were writing specifications that could in principle be used for a number of tasks. To summarize, although they are based on the same network-traversing metaphor, ATNs are a *procedural* formalism, whereas FSTNs and RTNs are essentially *declarative*.

Although ATNs still get used, there are a number of trends in NLP that are now leading to their decline. The original theory of transformational grammar whose operations they sought to embody has changed out of all recognition. Furthermore, many linguists have begun to recognize the advantages of simpler formal frameworks. The importance of declarative formalisms is becoming increasingly realized in computer science, an issue we will examine further in Chapter 4. Finally, as we will see in Chapter 6, the rise of charts as an efficient basis for parsing makes it clear that the depth-first search strategy adopted by most ATN implementations leaves something to be desired.

There would be something thoroughly perverse about seeking to program an ATN in Prolog: the native inhabitant would look at you with a puzzled frown and say 'If I wanted to go *there*, I would not start from *here*'. ATNs represent an unavoidably procedural approach to the representation of syntactic facts about natural language, whereas the whole ethos of logic programming favours declarative representations.

Our consideration of network-based language processors has led us to special-purpose programming languages for language analysis. Indeed, it can be shown that ATNs have the basic power to be used as a language for arbitrary programming tasks. Is there anything, then, that we can say theoretically about natural language analysis, apart from the fact that a general programming language is required for the job? Is there a future for declarative notations? In the next chapters we hope to show that both of these questions can be answered affirmatively.

SUMMARY

- RTNs use named subnetworks to generalize FSTNs.
- RTNs can represent recursive constructions in natural language.
- RTN traversal depends on a pushdown stack.
- PTs are to RTNs what FSTs are to FSTNs.
- PTs can be used to produce RTN parse trees.
- RTNs and PTs are descriptively more powerful than FSTNs and FSTs.
- ATNs augment RTNs with registers and register operations.
- ATNs are a procedural formalism.

Further reading

ELIZA is documented in Weizenbaum (1966). According to Woods (1970/1986), RTNs first appear in Conway (1963). For an elegant introduction to recursion in various forms, including RTNs, see Hofstadter (1979). Pushdown transducers appear to originate in Evey (1963). There is some relevant formal development in Choffrut and Culik (1983), and Wood (1987, pp. 352–5) provides a brief technical tutorial. Harel (1987) develops an elegant and sophisticated graphical formalism that can be used, inter alia, to define RTNs and PTs, as well as FSTNs and FSTs.

The immediate precursors of Woods's own classic (1970/1986) paper on ATNs were Thorne *et al.* (1968), and Bobrow and Fraser (1969). Later work by Woods on ATNs includes his (1973), (1980) and (1987) papers, while Kaplan (1972) argues for their psycholinguistic plausibility.

Tutorial material on ATNs can be found in many sources, including Bates (1978), Charniak *et al.* (1980, pp. 257–74), Charniak and McDermott (1985, pp. 197–222), Johnson (1983), Tennant (1981, pp. 49–76) and Winograd (1983, pp. 195–271). Pereira and Warren (1980/1986) provide an extended critique of ATNs, arguing for the use of declarative grammar formalisms instead.