3. Only use them for English, or other languages in which word order determines grammatical structure. ATNs would not be useful in parsing inflected languages like Latin.

4. Don't expect them to work all the time. Use them in applications where it's helpful if they work ninety percent of the time, not those where it's critical that they work a hundred percent of the time.

Within these limits there are plenty of useful applications. The canonical example is as the front-end of a database. If you attach an ATN-driven interface to such a system, then instead of making a formal query, users can ask questions in a constrained form of English.

23.2 The Formalism

To understand what ATNs do, we should recall their full name: augmented transition networks. A transition network is a set of nodes joined together by directed arcs—essentially, a flow-chart. One node is designated the start node, and some other nodes are designated terminal nodes. Conditions are attached to each arc, which have to be met before the arc can be followed. There will be an input sentence, with a pointer to the current word. Following some arcs will cause the pointer to be advanced. To parse a sentence on a transition network is to find a path from the start node to some terminal node, along which all the conditions can be met.

ATNs add two features to this model:

1. ATNs have registers—named slots for storing away information as the parse proceeds. As well as performing tests, arcs can modify the contents of the registers.

2. ATNs are recursive. Arcs may require that, in order to follow them, the parse must successfully make it through some sub-network.

Terminal nodes use the information which has accumulated in the registers to build list structures, which they return in much the same way that functions return values. In fact, with the exception of being nondeterministic, ATNs behave a lot like a functional programming language.

The ATN defined in Figure 23.1 is nearly the simplest possible. It parses noun-verb sentences of the form "Spot runs." The network representation of this ATN is shown in Figure 23.2.

What does this ATN do when given the input `(spot runs)`? The first node has one outgoing arc, a cat, or category arc, leading to node s2. It says, effectively,


```
(defnode s
  (cat noun s2
    (setr subj *)))

(defnode s2
  (cat verb s3
    (setr v *)))

(defnode s3
  (up `(sentence
         (subject ,(getr subj))
         (verb ,(getr v)))))
```

Figure 23.1: A very small ATN.

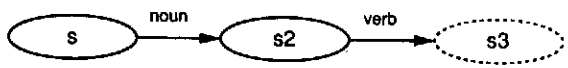

Figure 23.2: Graph of a small ATN.

you can follow me if the current word is a noun, and if you do, you must store the current word (indicated by *) in the `subj` register. So we leave this node with `spot` stored in the `subj` register.

There is always a pointer to the current word. Initially it points to the first word in the sentence. When cat arcs are followed, this pointer is moved forward one. So when we get to node `s2`, the current word is the second, `runs`. The second arc is just like the first, except that it is looking for a verb. It finds `runs`, stores it in register `v`, and proceeds to `s3`.

The final node, `s3`, has only a pop, or terminal, arc. (Nodes with pop arcs have dashed borders.) Because we arrive at the pop arc just as we run out of input, we have a successful parse. The pop arc returns the backquoted expression within it:

```
(sentence (subject spot)
          (verb runs))
```

An ATN corresponds to the grammar of the language it is designed to parse. A decent sized ATN for parsing English will have