Aho and Ullmann (1972) detail many of the properties of CFLs and CF-PSGs, Joshi (1987) gives a useful summary introduction to CF-PSG and some of its derivatives, and surveys of recent mathematical work on natural languages are to be found in Perrault (1984), and Gazdar and Pullum (1985). Fallacious or empirically ill-founded arguments mounted against the context freeness of natural languages are critically discussed in Pullum and Gazdar (1982), and Pullum (1984a), while a recent exemplar can be found in Rich (1983, p. 314). The non-fallacious argument based on Swiss German is due to Huybregts (1985) and Shieber (1985b), independently. The claim that natural languages are not in general finite state is thoroughly evaluated by Daly (1974). Useful discussion of the implications of the various claims and results for work in NLP can be found in Pullum (1983, 1984b).

Indexed languages and grammars were first discussed by Aho (1968) and a less technical presentation is available in Hopcroft and Ullman (1979). Their relevance to linguistic issues is the subject of Gazdar (1988).

CHAPTER 5

PARSING, SEARCH AND AMBIGUITY

A key feature of a grammar is the assignment of syntactic structures to a phrase that is judged a legal sentence of the language described. In Chapter 8, we will see how these structures are a vital element in the compositional computation of meaning. At this point, however, we consider the problem of parsing, which involves actually computing the structures assigned to a given phrase by a given grammar. As a declarative description of a language, a grammar does not specify how syntactic analyses are to be computed, and there is a vast space of possible parsing algorithms, even for the CF-PSGs. In this chapter, we survey the main dimensions of variation within this space – namely, bottom up/top down – depth first/breadth first – and the issue of storing intermediate results. Parsing sentences of natural languages is plagued by problems of global and local ambiguity. We consider some of the characteristics of this ambiguity problem.

5.1 A simple parsing problem

We will begin this chapter by considering a very simple problem; namely, how we might set about parsing the English sentence 'MediCenter employed nurses' given only Grammar1 from Chapter 4. The problem is simple for two reasons:

(1) The example sentence contains none of the phenomena that make parsing written English less than straightforward.

(2) By choosing an example from written English, we get a head start, as the datum comes to us partially preparsed.

Let us briefly consider this second point. Written English employs a convention whereby words are set off from each other by spaces. But spoken English is not like that – words are not, in general, separated by silences, the obvious temporal analogue of white space on paper. Nor are their boundaries otherwise marked. So, for spoken English, indeed the spoken form of any natural language, the problem is significantly harder: the written analogue would be parsing 'M e d i C e n t e r e m p l o y e d n u r s e s'. Now the parser not only has to do what our parser will have to do, but it also has to determine the appropriate word boundaries in the string. If that does not seem intuitively very difficult, then consider 'T h e n e a r l y i n g o n e s e r r e d'. Actually, the problem with speech is even worse, since one effect of running spoken words together is to change or even omit the sounds that appear word-peripherally. The parsing of speech is, thankfully, a topic that falls outside the remit of this book. But the problem of a parser having to determine the position of syntactically relevant boundaries in the absence of 'white spaces' does not simply go away when we turn our attention from speech back to the written form. Consider the following correctly written Turkish sentence:

çöplüklerimizdekilerdenmiydi

which translates into English as 'Was it from those that were in our garbage cans?' Clearly, a parser for written Turkish is going to have to determine the boundaries of those syntactic and semantic elements whose counterparts in written English would be words set off with white space.

With these matters in mind, let us now turn back to parsing our three-word example. For convenience, we repeat Grammar1 in its entirety here.

```
Rule {simple sentence formation}
  S → NP VP.
Rule {transitive verb}
  VP → V NP.
Rule {intransitive verb}
  VP → V.
Word Dr Chan:
    <cat> = NP.
Word nurses:
    <cat> = NP.
Word MediCenter:
    <cat> = NP.
Word patients:
    <cat> = NP.
Word died:
    <cat> = V.
Word employed:
    <cat> = V.
```


Exercise 5.1 Write a transducer that will find word boundaries in 'scrunched' (that is, white space removed) strings of English words drawn from some small set such as the names of the digits. Your program should map **[f, i, v, e, s, e, v, e, n, t, w, o]**, say, into **[five, seven, two]**. [*easy*]

5.2 Bottom-up parsing

Given our three-word sentence:

MediCenter employed nurses.

we can see where the word boundaries are, but what else do we already know about this string? Well, thanks to the lexicon of Grammar1, we know what syntactic category the first word belongs to. So, we can label that word NP and proceed from there. What we have done is find the first place in the string that matches the RHS of a rule or lexical entry. We can then label the sequence of words or categories that matched the RHS of the rule with the LHS of the rule:

NP_____
MediCenter employed nurses.

One way of parsing involves repeating this operation, at each stage, using

the sequence of highest-level labels as the new string to operate on. So, we now continue, trying to further rewrite the string 'NP employed nurses'. Given only that, we can ask if there is any rule in the grammar whose RHS is simply NP – for example, K → NP. If there is, then we could explore the possibility of allowing this NP to be dominated by K in the structure we are trying to build up. But there is no such rule, so we are forced to consider the next word in the string, which we find to be of category V:

NP_____ V_____
MediCenter employed nurses.

Our task would now appear to be that of attempting to group these categories together in a manner permitted by the grammar. First, we check (again!) if any rule simply has NP and nothing else on the RHS, but there are no such rules in Grammar1. Now we need to know if there is a rule that allows us to group the sequence NP V together. To determine this, we look at the RHS of each rule in Grammar1 to see if it has the form NP V. None of them do. Turning our attention to the second item, we can then ask if any RHS is identical to V and, in fact there is, in the shape of the rule that allows a VP to consist simply of a V. So, we can add this information to the parse tree that we are trying to build up:

VP_____
NP_____ V_____
MediCenter employed nurses.

Now that we have found a VP following the initial NP, we can go back to the beginning of the string and ask if we have an initial sequence that can be subsumed under some category. Since we are acting out a rather stupid algorithm here, we will once again check to see if the single initial NP can be found as an RHS. Since the rule set provided by Grammar1 is a constant and not something that varies over the course of the parse, the answer will again be no. With that out of the way, we check to see if NP VP can be found as an RHS, and the answer is, of course, that it can. The only rule having NP VP as its RHS has S as its LHS, so we augment our parse tree with an S spanning the initial NP and the immediately following VP that we found. We are now looking for rules whose RHS matches parts of the string 'S nurses'. There is no rule whose RHS is 'S' or 'S nurses'. When we look 'nurses' up in our lexicon, we find it to be an NP, and so we progress to the following situation:

S_____
VP_____
NP_____ V_____ NP___
MediCenter employed nurses.

At this point, however, we cannot proceed. S itself does not occur as an RHS, nor does S NP, and nor does NP. So, there is nothing we can do with the sequence. Our goal is to find an S spanning the entire string, but this route has led us to an S spanning the first two words in the string and a dangling unattached NP at the end. Clearly, we have gone wrong somewhere.

Faced with this impasse, we will backtrack to the last point at which we were faced with a choice and explore one or more other possibilities. In doing so, we will unload the parts of the parse tree that we built subsequent to the choice point. Let's go back to the following situation:

VP_____
NP_____ V_____
MediCenter employed nurses.

We have just seen that putting the NP and VP together here as an S led nowhere. A VP by itself does not exhaust the RHS of any of the rules in our grammar, so the only thing left to do is to look up the final word in the dictionary (again!), annotate the structure accordingly, and see where we can go from there:

VP_____
NP_____ V_____ NP___
MediCenter employed nurses.

At this point, a dim parsing algorithm will once again combine the initial NP and V into a sentence. However, when this fails to work out, it will eventually try something else and check if the sequence NP VP NP occurs as an RHS (it does not), then if VP NP does (it does not) and, finally, whether the last NP can be subsumed under some other category (we have already checked this NP-as-RHS issue several times – the answer remains no). Again, we face an impasse: assuming that the parse tree contains this VP leads nowhere, as we have seen by exhaustively checking all the possibilities. So, we must backtrack still further, ridding ourselves of this VP hypothesis as we do so. After some further thrashing around, we arrive at the following configuration:

NP_____ V_____ NP___
MediCenter employed nurses.

Given our implicit order of proceeding, the next thing to check is whether V NP occurs as an RHS. It does, in the shape of the other VP rule in our

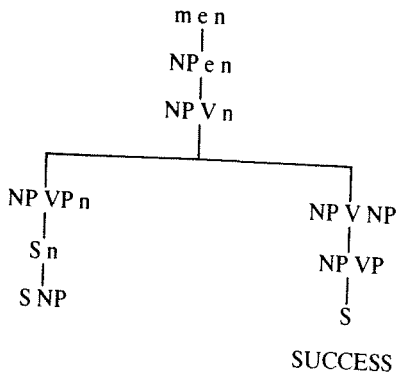

Figure 5.1 Search tree for bottom-up recognition.

grammar; that is, that which allows VP to dominate V followed by NP:

```
          _____VP_____
NP_____ V_____ NP___
MediCenter employed nurses.
```

Returning to the start of the sentence, we check (for the nth time) whether NP can exhaust an RHS, and then check if NP VP can. As previously, the answer to the latter question is yes, and so we can proceed to add S to our parse tree:

```
_____S_____
          _____VP_____
NP_____ V_____ NP___
MediCenter employed nurses.
```

This S, unlike the one we found previously, spans the entire string, and so we have a success on our hands. The parsing strategy that we have been walking through has, at last, found a parse. If all we want is the first parse tree found and no others, then we can halt. If, however, we want our parser to find all the possible parsings of this string, then we will have to let it run a while longer. It will not find any more in this case, since this string is unambiguous with respect to Grammar1, but our parser only knows that it has found an S. It cannot know at this stage that this is the only spanning S to be found. To establish that, it must continue in just the manner we have already seen, chasing right to the end of every dead end until it has exhausted them all. But it would try our patience to follow it on this fruitless journey!

What has been illustrated is a kind of *bottom-up parsing*, as the parse tree is built from the bottom upwards. As we saw, bottom-up parsing

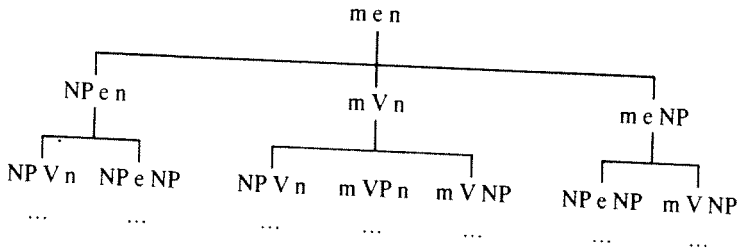

Figure 5.2 Redundancy through rewriting of lexical items.

involves searching through a space of alternatives, not all of which lead to a successful solution. In Chapter 2, we discussed the notion of a search problem that involved exploring possible states, given a way of determining which possible states might arise from any state. As we saw there, it is convenient to display the possible states in a search problem as a *search tree*, with each state appearing above the states that could come next. Figure 5.1 shows part of the search tree for our bottom-up parser looking at the three-word sentence, where *m* represents MediCenter, *e* represents employed and *n* represents nurses.

A state of a bottom-up parsing process can be summarized by the sequence of highest-level labels of phrases that have been found. In fact, this is a characterization of a state of a bottom-up *recognizer*. For brevity, we only deal with recognizers in our search trees.

In our example, there is one main place where a choice has to be made. The choice is whether to rewrite the V into a VP or whether to continue and rewrite 'nurses' into an NP. In our example trace, the parser initially took the wrong decision here, and so had to back up and change its mind. Having made this decision, it eventually reached a dead end in the search tree, where a spanning sentence had not been found but no further rewritings were possible.

Note that we have not yet specified a precise algorithm for bottom-up parsing. Indeed, it is possible to imagine algorithms that involve much worse search spaces. For instance, in our example we have assumed in advance that there is no rule in the grammar like NP → MediCenter V which, although it does not initially apply, could later incorporate 'MediCenter' into a larger phrase, if that word is not immediately converted to an NP. So, we do not even consider failing to rewrite 'MediCenter' and moving on to 'employed' instead. The assumption that lexical items are only introduced in the grammar by rules whose RHSs are wholly lexical can obviously be enforced quite easily by the grammar writer. If this is done, then we can avoid search trees like that shown in Figure 5.2, where many states, like NP V n and m V NP, occur redundantly in different parts of the search space.

Another characteristic of our informal parser is that, having decided to rewrite NP V n into NP V NP, rather than NP VP n, it did not then

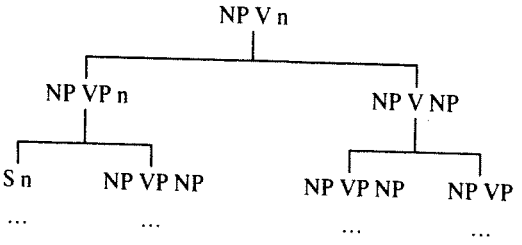

Figure 5.3 Redundancy through combining independent rewrites.

consider again rewriting the V into a VP, as this would again introduce redundancy, as shown in Figure 5.3.

The essence of bottom-up recognition can be captured quite elegantly in a simple Prolog program. As before, we can see the problem as that of successively transforming the initial string into a string consisting of a single category – for example, s. The predicate **burecog** is given a string of words and categories, implemented as a Prolog list, and succeeds if, using the rules of the grammar, it can transform it to the list [s]:

```
burecog([s]).
burecog(String) :-
  append(Front, Back, String),
  append(RHS, Rest, Back),
  rule(LHS, RHS),
  append(Front, [LHS | Rest], NewString),
  burecog(NewString).
```

This definition makes use of the **append** predicate and assumes that the grammar rules are represented in the following way:

```
rule(s, [np, vp]).
rule(np, [john]).
rule(vp, [v]).
rule(vp, [v, np]).
rule(v, [saw]).
rule(np, [mary]).
```

How does **burecog** work? First of all, **append** is used twice to find a way in which the input **String** decomposes into the **Front**, followed by the **RHS**, followed by the **Rest**. On backtracking, the **append** goals will find every possible way that this can be done. Given such a decomposition, an attempt is made to match the **RHS** part to the right-hand side of a rule. If this succeeds, a new list is constructed by putting together the items in the string that appear before the matched portion, the **LHS** of the rule and the items appearing after the matched portion. Then, an attempt is made to

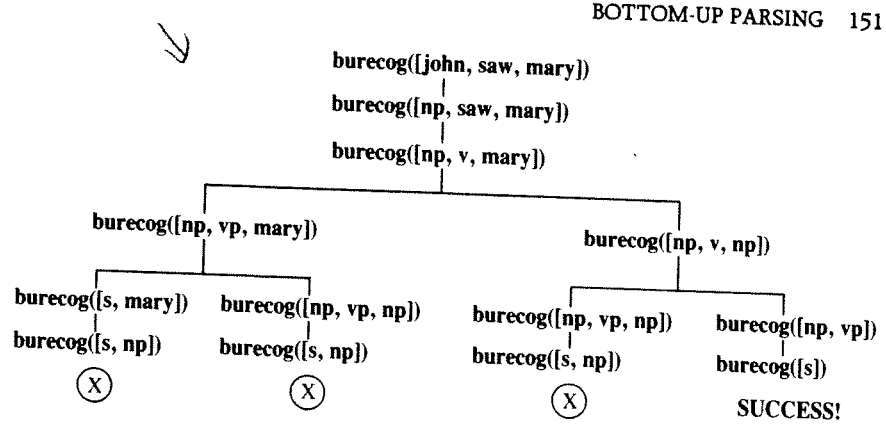

Figure 5.4 Call tree for bottom-up recognizer.

further transform the resulting **NewString**. Although this definition is a clear and concise description of bottom-up recognition, the way that **append** is used to investigate all possible decompositions of the string at each stage means that it is hopelessly inefficient. Figure 5.4 shows part of the call tree for the program parsing [**john, saw, mary**] with this grammar, and certain duplications show up in it clearly. We will consider more efficient approaches to parsing in Prolog later in this chapter.

There is a very large space of possible parsing strategies and the strategy that we have walked through is merely one among many. But it serves to illustrate the kind of way that a parser works, the kinds of questions it needs to ask, and the kinds of blind alleys and wasteful repetitions that make the simplest algorithms inefficient. The basic observation to make about our parser is that it works from left to right: it does everything it can with the first item before exploring what it can do with the first two items, and so on. Clearly, there is no logical necessity about this. We could just as well have described a parser that worked from right to left. These two possibilities by no means exhaust the candidates: we could, to choose examples at random, have scanned the string looking for a verb and then worked outwards from the verb, or we could have worked zigzag fashion across the structure we were building. Notice, however, that parsers that attempt to parse a natural language as it is being produced, either as it is spoken or as it is typed, are forced to adopt a strategy that is basically left to right.

Our parser was bottom up, driven entirely by the data presented to it and building successive layers of syntactic abstraction on the base provided by that data. Parsers that work bottom up and from left to right, when they also index the rules of the grammar on the leftmost RHS category, are common enough to have been given a usefully mnemonic name 'left-corner parsers'. One major problem with bottom-up parsing is rules that have empty RHSs – that is ε-productions. If we had replaced our

VP rules with the following rules (where VCOMP is a category of verb complements):

Rule {verb phrase}
VP → V VCOMP.

Rule {verb complement is NP}
VCOMP → NP.

Rule {verb complement is empty}
VCOMP → .

which generate the same strings, we would have had difficulties. For if there is a rule with an empty RHS, that rule applies *anywhere* in the input string. Going bottom up, our parser is thus able to 'find' a phrase of the relevant category wherever it looks. With these rules, our parser would have been able to put VCOMPs all over the place and would have had to deal with intermediate structures like:

VCOMP NP VCOMP V VCOMP NP VCOMP VCOMP VCOMP
MediCenter employed nurses

Fortunately, there is a simple algorithm to convert any CF-PSG to a grammar with no ε-productions that generates the same set of strings. The algorithm will not, in general, produce a grammar that induces the same parse trees as the original, however, and this may be a problem if we care about the syntactic structures that are built.

Exercise 5.2 Write a program that takes a grammar represented as a list of rules and carries out ε-production removal. [*hard*]

Exercise 5.3 Draw an expanded version of Figure 5.4 to show the bottom-up recognizer looking for all ways to recognize **[john, saw, mary]**. [*intermediate*]

5.3 Top-down parsing

The naive bottom-up parser of Section 5.2 never formed hypotheses about what it was looking for or used such hypotheses to decide its next move. It only ever checked rules to see if there was a legitimate way of putting together the pieces already to hand. That is why it encountered problems with rules that involved putting together 'empty space'. By contrast, a pure

(left-to-right) *top-down* (depth-first) *parser* would have proceeded roughly as follows:

I am looking for an S.
What can an S consist of?
An S can consist of an NP followed by a VP.
So I need to look for an NP first.
What can an NP consist of?
There are no grammar rules expanding NP.
There is a lexical entry listing 'nurses'
as a member of the category NP.
Is the first word in the string 'nurses'?
No.
There is a lexical entry listing 'MediCenter'
as a member of the category NP.
Is the first word in the string 'MediCenter'?
Yes.
I have found an NP consisting of the word 'MediCenter'.
I now need to look for a VP.
*** What can a VP consist of?
A VP can consist of a V.
I now need to look for a V.
What can a V consist of?
There are no grammar rules expanding V.
There is a lexical entry listing 'died'
as a member of the category V.
Is the next word in the string 'died'?
No.
There is a lexical entry listing 'employed'
as a member of the category V.
Is the next word in the string 'employed'?
Yes.
I have found a V consisting of the word 'employed'.
I have found a VP consisting of a V
consisting of the word 'employed'.
I have found an S consisting of:
an NP consisting of the word 'MediCenter'
and a VP consisting of
a V consisting of the word 'employed'.
Have I reached the end of the string?
No.
Oh dear, I must have done something wrong.
Try going back to *** and doing something different.
I still need to look for a VP.
What can a VP consist of?
A VP can also consist of a V followed by an NP.
I now need to look for a V.
What can a V consist of?
There are no grammar rules expanding V.

There is a lexical entry listing 'died'
as a member of the category V.
Is the next word in the string 'died'?
No.
There is a lexical entry listing 'employed'
as a member of the category V.
Is the next word in the string 'employed'?
Yes.
I have found a V consisting of the word 'employed'.
Now I need to look for an NP.
What can an NP consist of?
There are no grammar rules expanding NP.
There is a lexical entry listing 'nurses'
as a member of the category NP.
Is the first word in the string 'nurses'?
Yes.
I have found an NP consisting of the word 'nurses'.
I have found a VP consisting of:
a V consisting of the word 'employed'
followed by an NP consisting of the word 'nurses'.
I have found an S consisting of:
an NP consisting of the word 'MediCenter'
and a VP consisting of
a V consisting of the word 'employed'
followed by an NP consisting of the word 'nurses'.
Have I reached the end of the string?
Yes.
I have succeeded.

Figure 5.5 shows the search tree for top-down parsing of the example sentence 'MediCenter employed nurses'. In top-down recognition, a state can be characterized by a sequence of goals and a sequence of remaining words. In our tree, we separate the goals from the remaining words with a colon (:). Thus, d NP : e n describes the state of the recognizer when it is trying to find the word 'died' followed by an NP and when the remaining words are 'employed nurses'.

Top-down parsing does not have any problem with ε-productions, but (when performed left to right) encounters trouble with rules that exhibit *left recursion*. Here is another way we could have rewritten our VP rules:

Rule {simple verb phrase}
VP → V.

Rule {complex verb phrase}
VP → VP NP.

but now the rules will in fact generate more strings. Of these rules, the second is left recursive in that the first symbol on the RHS is the same as

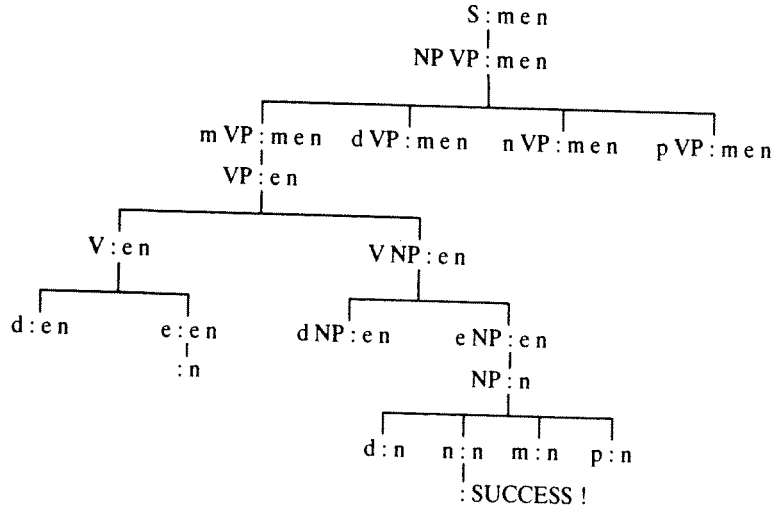

Figure 5.5 Search tree for top-down recognition.

the LHS. With left-recursive rules in a grammar, a left-to-right, top-down parser will get into an infinite loop if it makes the wrong choices, and if we are asking for all parses it will, of course, in the end have to try all possible choices:

I now need to look for a VP.
What can a VP consist of?
A VP can consist of a VP followed by an NP.
I now need to look for a VP.
What can a VP consist of?
A VP can consist of a VP followed by an NP.
I now need to look for a VP.
What can a VP consist of?
A VP can consist of a VP followed by an NP.
...

When we are using complex feature-based categories, left recursion also arises when the first category on the RHS of a rule is more general than the one on the LHS (see Chapter 7 for a fuller discussion of the issues that features give rise to in parsing). Again, it is possible to transform a left-recursive grammar into an equivalent one with no left-recursive rules, and one that generates exactly the same set of strings (although it will not assign the same structures).

Exercise 5.4 Write a program that takes a grammar represented as a list of rules and converts it into another list of rules which generate the same language but from which all left-recursion has been eliminated. [*hard*]

Exercise 5.5 Implement a version of the top-down parser whose operation was laboriously outlined and equip it with English language tracing facilities so that it prints out a commentary, as it runs that is similar to the one we used to describe it. Observe it parse 'MediCenter employed nurses' and note any discrepancies between our description and the one that it prints out. [*intermediate*]

5.4 Parsing in Prolog

In the Prolog programs that result from the standard translation, DCGs implement top-down, left-to-right recognizers or parsers. This can be readily inferred from observing the execution of a DCG parser. When the standard translation of a particular DCG rule is tried, for instance:

```
s([s, NP, VP], S1, S3) :-
  np(NP, S1, S2),
  vp(VP, S2, S3).
```

a hypothesis has been made about how a particular type of phrase (s) might be found in terms of smaller phrases (of category np and vp). Thus, the goal to recognize (or parse) a sentence decomposes into goals to recognize an NP and a VP. These themselves decompose into goals to find smaller phrases, and so the process continues until the word level is reached. This is pure top-down behaviour, as no account is taken of the input string until a hypothesis actually predicts that a word of a particular kind might appear there. The fact that the recognition proceeds left to right is a simple consequence of the Prolog execution strategy. Since Prolog attempts the goals in the body of a clause in the order they are written, in this case it will only start looking for a VP after it has found an initial NP. Similarly, for any rule that has multiple elements on the RHS, the corresponding Prolog program will look for them in the order written. The result is a pure left-to-right strategy.

If you want to do recognition or parsing in a Prolog environment, then the most straightforward way of going about it is simply to use the ready-made DCG interpreter provided with most Prolog systems and discussed in Chapter 4. But, if such an interpreter is not available, then it is still trivial to implement left-to-right, top-down, depth-first recognition or

parsing, requiring barely half a dozen lines of code in addition to the grammar. Here, for example, is a recognizer where, as usual, we exploit difference lists:

```
recognize(Category, S1, S2) :-
  rule(Category, Daughters),
  matches(Daughters, S1, S2).
```

A string can be recognized as an instance of Category if there is a rule that allows Category to dominate Daughters and if Daughters matches the string:

```
matches([ ], S, S).
matches([Word], [Word | S], S).
matches([Category | Categories], S1, S3) :-
  recognize(Category, S1, S2),
  matches(Categories, S2, S3).
```

A list of categories matches a string if:

- both are empty,
- both consist of just a word, or
- if an initial portion of the string can be recognized as the first category in the list and the remainder of the category list matches the remainder of the string.

This recognizer expects rules in the following format:

```
rule(s, [np, vp]).
rule(vp, [iv]).
rule(vp, [tv, np]).
rule(np, ['MediCenter']).
rule(np, [nurses]).
rule(iv, [died]).
rule(tv, [employed]).
```

Since DCGs (under the standard translation) give rise to top-down, left-to-right parsers, both DCG parsers and this recognizer will suffer from the problem of left recursion. That is, we can no more expect to write the DCG rule:

```
np --> np, pp.
```

than we can expect to be able to write the clause:

```
human(X) :-
  human(X),
  animate(X).
```

say. In practice, it is not usually a problem to rewrite a DCG grammar to avoid left recursion. Moreover, since the structures built by a DCG are not required to correspond exactly to the trace of the parser, in practice we can also arrange to have the same structures produced by the rewritten grammar as might have been produced by an original, left-recursive, grammar. For example:

```
np(np(Det, N)) --> det(Det), n(N).
np(np(NP, PP)) --> np(NP), pp(PP).
```

can be rewritten as:

```
np(np(Det, N)) --> det(Det), n(N).
np(PPs) --> det(Det), n(N), pps(np(Det, N), PPs).
pps(NP, np(NP, PP)) --> pp(PP).
pps(NP, PPs) --> pp(PP), pps(np(NP, PP), PPs).
```

Although the standard translation of DCGs into Prolog programs gives rise to top-down parsers, this is by no means the only possible translation available. We will now show how CF-PSG rules can be mapped into rather more complex clauses in a way that converts the CF-PSG into a bottom-up parser, although, for simplicity, we will start out by considering just recognition. Let us assume that our lexicon is encoded as follows:

```
word(np, 'MediCenter').
word(np, nurses).
word(tv, employed).
word(iv, died).
```

As before, the translation will associate a distinct predicate with each category, although the interpretation of this predicate will be different to that implicit in the standard translation. The translation will also make use of a special predicate **goal**, which also serves as a kind of top-level predicate:

```
goal(Goal, [Word0 | Words1], WordsN) :-
  word(Category, Word0),
  Term = .. [Category, Goal, Words1, WordsN],
  call(Term).
```

The first argument here is the category we are looking for, and the second and third arguments are used for the standard difference list representation of the string to be recognized. Thus, we might call **goal** like this:

```
?-goal(s, [nurses, died], []).
```


If we do, then **word(np, nurses)** will succeed and **Term** will get bound to **np(s, [died], [])** which is then called. When we call the predicate **goal**, Prolog attempts bottom-up analysis of the relevant part of the string and will succeed if there is a way of recognizing a phrase of type **Goal**. Bottom-up analysis consists of looking at the next word (seeing what category it could have) and then attempting to complete an analysis of the required **Goal** using this material. The crucial thing to note here is that the **np** predicate gets called *after* an NP has been recognized. Thus, the **np** predicate is *not* a predicate for recognizing NPs, but rather a predicate for recognizing the material that follows an NP. So, the interpretation of **np** is as follows:

```
% np(Goal, S1, S2) :-
%   The words in S1-S2 are a possible way of extending an NP to a phrase
%   of type Goal (completing a phrase of type Goal, after an initial NP).
```

Given a two-daughter rule like the following:

c0 → c1 c2

we will translate it into this:

```
c1(Goal, Words0, WordsN) :-
  goal(c2, Words0, Words1),
  c0(Goal, Words1, WordsN).
```

So, our rule for expanding S:

s → np vp

should be translated into this:

```
np(Goal, Words0, WordsN) :-
  goal(vp, Words0, Words1),
  s(Goal, Words1, WordsN).
```

In other words, if we are looking for a phrase of type **Goal** and have found an NP, we can complete the analysis by finding a VP and after that completing the analysis of a phrase of type **Goal**, working from the newly found S. Another way of saying this is 'we can extend an NP to a given **Goal** by first of all finding a VP and then extending the resulting S to make the **Goal**'. Pursuing our example, **np(s, [died], [])** will succeed if **goal(vp, [died], [])** and s(s, [], []) succeed. ...

of termination clauses, one for each syntactic category in our grammar, for our recognizer, as follows:

```
s(s, Words, Words).
np(np, Words, Words).
vp(vp, Words, Words).
iv(iv, Words, Words).
tv(tv, Words, Words).
```

These clauses have the obvious interpretation; for instance, if you are looking for an S and have found an S, then one possibility is simply to succeed, without consuming any further words. This just leaves **goal(vp, [died], [])** to consider: when we call **goal** with those arguments, **word(iv, died)** will succeed and **Term** will get bound to **iv(vp, [], [])** which is then called. This is not going to succeed in virtue of any of the termination clauses listed above, so clearly a clause that exists in virtue of the grammar translation is what is required.

Given a single-daughter rule like the following:

c0 → c1

we will translate it into this:

```
c1(Goal, Words0, WordsN) :-
  c0(Goal, Words0, WordsN).
```

So, a rule for expanding VP as just an intransitive verb,

vp → iv

should be translated into this:

```
iv(Goal, Words0, WordsN) :-
  vp(Goal, Words1, WordsN).
```

In other words, if you want to complete a phrase of type **Goal** and have found an IV, then you can use any way of completing such a phrase assuming that you have found a VP. Given this, **iv(vp, [], [])** will plainly succeed if **vp(vp, [], [])** succeeds, and the latter will succeed thanks to the presence of the relevant termination clause. So, our original goal, of recognizing 'nurses died' as an instance of S, succeeds, as we would want it to.

In some ways, the name of the predicate **goal** is slightly misleading in a recognizer of this kind, since it suggests that the program is actively seeking a particular kind of phrase, a very top-down idea. In fact, though,

if you look at how **goal** is defined, you will see that the first argument is not used at all to influence how bottom-up recognition proceeds from the first word. Instead, the **Goal** argument simply serves to ensure that any bottom-up analyses with the right category are recognized as possible solutions to the original problem.

Having considered the recognition case in some detail, we will now show how to provide a parser by an analogous translation. The top-level predicate of our parser is still **goal**, only now defined as follows:

```
goal(Goal, Parse2, [Word0 | Words1], WordsN) :-
  word(Category, Word0),
  Parse1 = .. [Category, Word0],
  Term = .. [Category, Goal, Parse1, Parse2, Words1, WordsN],
  call(Term).
```

The second argument here is the parse tree that gets returned. Thus, we might call **goal** as follows:

```
?-goal(s, Parse2, ['MediCenter', employed, nurses], []).
```

If we do, then **word(np, 'MediCenter')** will succeed, **Parse1** will get bound to **np('MediCenter')** and **Term** will get bound to **np(s, np('MediCenter'), Parse2, [employed, nurses], [])** which is then called. We can now interpret a predicate like **np** as follows:

```
% np(Goal, Parse1, Parse2, S1, S2) :-
%   The words in S1-S2 are a possible way of extending an NP with
%   parse tree Parse1 to a phrase of type Goal. The parse tree of the Goal
%   phrase is Parse2.
```

Given a single-daughter rule like the following:

c0 → c1

we will translate it into this:

```
c1(Goal, Parse1, Parse, Words0, WordsN) :-
  c0(Goal, c0(Parse1), Parse, Words0, WordsN).
```

And given a two-daughter rule like the following:

c0 → c1 c2

we will translate it into this:

```
c1(Goal, Parse1, Parse, Words0, WordsN) :-
  goal(c2, Parse2, Words0, Words1),
  c0(Goal, c0(Parse1, Parse2), Parse, Words1, WordsN).
```

Termination rules will now look like this:

```
c0(c0, Parse, Parse, String, String).
c1(c1, Parse, Parse, String, String).
c2(c2, Parse, Parse, String, String).
```

As pure bottom-up parsers, programs of this kind suffer from various kinds of inefficiencies. Consider what would happen if we had multiple lexical entries for the same word:

```
word(iv, strikes).
word(np, strikes).
```

If we are trying to recognize a sentence starting with this word, we will first look for its category and then try and extend a phrase of this category to a phrase of type S. The first possible category is IV, and so we will try to extend this analysis. A rule like:

vp → iv

tells us that we can extend an analyzed IV into an analyzed phrase of some category if we can find a way of extending a VP to a phrase of the given category. So, we will look for ways of extending a VP into an S. If there are rules with VP as the first category on the RHS, this will involve looking for further phrases that can combine with a VP. All this activity is, however, wasted if in fact an IV can never appear as the first word in a sentence. The trouble is that the bottom-up parsers only consider the goal category when they return a complete solution of some kind; that is, the goal information is not used to actively guide what alternatives are considered beforehand. We can improve the efficiency of this kind of left-to-right, depth-first, bottom-up parser by precompiling a **link** relation: a category **c1** is linked to a category **c2** if and only if **c1** can appear as an initial constituent in **c2**. The relation is reflexive, so we have:

```
link(Category, Category).
```

and, for the little grammar assumed in the foregoing discussion, we also have:

```
link(np, s).
link(iv, vp).
link(tv, vp).
```

The relationship is transitive, so if our grammar had included a rule np → det n, then we would have had both of the following:

```
link(det, np).
link(det, s).
```


Given such **link** statements, we can now modify our parser to exploit them:

```
goal(Goal, Parse2, [Word0 | Words1], WordsN) :-
  word(Category, Word0),
  link(Category, Goal),
  Parse1 = .. [Category, Word0],
  Term = .. [Category, Goal, Parse1, Parse2, Words1, WordsN],
  call(Term).
```

This now expresses the rather obvious fact that there is no point in exploring a parse based on a particular category assignment to the first word in the string if that category cannot appear as a first constituent in the category of the string assumed by the **Goal**. Likewise, we shall want to insert a **link** check at the start of each of our rule translations:

```
np(Goal, NP, Parse, Words0, WordsN) :-
  link(s, Goal),
  goal(vp, VP, Words0, Words1),
  s(Goal, s(NP, VP), Parse, Words1, WordsN).
```

There is no point in seeking to complete a sentence in this position if sentences cannot appear in this position. Analogously with the single-daughter rules:

```
iv(Goal, IV, Parse, Words0, WordsN) :-
  link(vp, Goal),
  vp(Goal, vp(IV), Parse, Words0, WordsN).
```

Exercise 5.6 Convert the top-down recognizer given in the text into a parser. Note that no additional lines of code are required. [*easy*]

Exercise 5.7 Complete the DCG for the NP–PP construction given in the text so as to achieve parses such as the following:

```
?-np(NP, [the, day, after, the, holiday, in, the, fall],[ ]).
NP = np(np(the, day), pp(after, np(np(the, holiday), pp(in, np(the, fall)))))
NP = np(np(np(the, day), pp(after, np(the, holiday))), pp(in, np(the, fall)))
```

[*easy*]

Exercise 5.8 Convert the following left-recursive rule into DCG rules that will build identical parse trees and integrate them into the grammar fragment just given:

```
n(nom(N, VP)) --> n(N), [that], vp(VP).
```

[*intermediate*]

Exercise 5.9 Work out the bottom-up recognizer translation of the vp → tv np rule needed by our grammar and then write down each step of a proof that **goal(s, ['MediCenter', employed, nurses | X], X)** will succeed. [*easy*]

Exercise 5.10 How should rules of the form c0 → c1 c2 c3 translate? [*intermediate*]

Exercise 5.11 Bottom-up parsing cannot cope with ε-productions. How is this reflected in the limits of the translation scheme described? [*intermediate*]

Exercise 5.12 Write a translation program that will take a file of CF-PSG rules written in a DCG format and compile them into:

(a) a set of **word** clauses,

(b) a set of termination clauses, and

(c) a set of bottom-up parsing rules, along the lines set out above.

[*hard*]

Exercise 5.13 Modify your DCG rule translation program so that it:

(a) sets up the the **link** relation automatically from the information in the grammar rules, and

(b) inserts the appropriate **link** clauses in the appropriate places in the rule translations.

[*hard*]

Exercise 5.14 Add the following rules to your example DCG grammar:

```
np --> np, rel.
rel --> wh, vp.
wh --> [who].
```

What happens when you try and parse with the new grammar under the standard (top-down) DCG translation? What happens when you try and parse with the output of your bottom-up translator with this grammar as input? Why is there such a difference? [*intermediate*]

Exercise 5.15 Elaborate the definition of **goal** so that:

(a) it keeps records of successful and unsuccessful calls by asserting the relevant information into the database, and

(b) it uses these records to avoid doing work that it has done already.

[*intermediate*]

Exercise 5.16 Given an example grammar with a dozen or more syntactic rules and a set of example sentences of various lengths, compare the performance of:

(a) the basic parser,

(b) the basic parser plus the **link** relation,

(c) the record-keeping parser of the last exercise with no **link** relation, and

(d) the record-keeping parser plus **link** relation.

[*intermediate project*]

5.5 Comparing strategies

As with the case of the left-to-right, right-to-left dichotomy, the limiting cases of pure top-down and pure bottom-up by no means exhaust the strategies possible on this dimension. We could, for example, have a parsing strategy that begins by operating bottom up, until it has identified the categories of all the lexical items, and then operates top down in its search for the higher-level syntactic structures. With a more sophisticated grammar than Grammar1, we might employ a mixed parsing strategy that depended on the feature system: constituents carrying one class of feature would be hypothesized top down, whereas words identified as carrying features from another class would trigger attempts to build up the relevant constituents from the bottom.

How are we to decide which of top-down or bottom-up parsing would be better for a given application, or whether a mixed strategy is called for? It helps to consider the shape of the parsing search spaces for example sentences. Looking at the foregoing top-down and bottom-up search spaces, we see that a great deal of the branching in the top-down space occurs at the level of lexical items. If we altered our top-down algorithm to look ahead at the next remaining word, before spawning a state that required a specific lexical item, this would make the two trees for our example have exactly the same amount of branching. To see real differences between top-down and bottom-up search trees, we have

consider more complex grammars and sentences. Bottom-up parsing can try to build a phrase that top-down parsing would never consider, because such a phrase could never occur in the context of the surrounding phrases. Conversely, top-down parsing may hypothesize many kinds of phrase that bottom-up parsing will not consider, because relevant lexical items are not present.

The shape of the search trees does not, however, tell us everything about parsing efficiency. In the last paragraph, it was suggested changing the way the top-down parser deals with the prediction of lexical items, but in so doing we are just moving the equality test slightly forwards in time, and not affecting the number of such tests that would have to be made. In general, an important factor is the *number* of rules tried at each point of the analysis, because even rules that cannot actually be used in the analysis can cost the parser if they have to be tried. The number of rules tried depends crucially on the extent to which we can avoid repeating previous unsuccessful tries on material that has not changed, and also on the way in which the rules are stored. We will consider the problem of remembering previous rule failures in Section 5.6 as well as in the next chapter, but let us concentrate, for now, on the rule storage issue.

For top-down parsing, it is common practice to index rules by their LHS categories, so that it is easy to go from a category to all the rules that could expand it. For bottom-up parsing, a rule can be indexed by its leftmost RHS category (for left–right parsing) or by some other category, such as a category that appears in very few rules. Treating a lexical rule top down thus involves considering all the lexical rules for the relevant category. On the other hand, treating it bottom up only involves looking at the rules with a particular word on the RHS, and we would not expect there to be many of these. In general, we would like to parse instances of a category bottom up if there are many rules for that category but the indexing categories on the RHS of the rules index very few rules. Conversely, we would like to parse instances of a category top down if there are few rules for that category and the RHSs of those rules contain indexing categories that index many rules. These competing requirements may not, of course, be reconcilable.

5.6 Breadth-first and depth-first search

All the parsers presented so far have been depth-first parsers. A *depth-first* strategy is one in which each hypothesis is pushed as far as it can be before entering upon the exploration of another hypothesis. Thus, our bottom-up parser went as far as it could with the hypothesis that the verb 'employed' was the sole content of a dominating verb phrase before being forced to abandon this idea and explore instead the hypothesis of putting the verb

together with the following noun phrase. A depth-first strategy then is a quintessentially sequential one. By contrast, a *breadth-first* strategy maintains a number of hypotheses at the same time, advancing each in turn by a single step. Ideally, as time goes by, hypotheses fail and the breadth-first device is left with a smaller space of hypotheses to consider.

Such a strategy is ideally suited to implementation on parallel machinery, but can be readily implemented by a sequential machine that distributes turns, rather like the way in which a card dealer distributes cards prior to a game like bridge. Here is how a particular bottom-up, breadth-first parser might have gone about parsing our three-word example sentence. Notice that 'a' and 'b' have been used in the following simply to make it clear which constituent is being referred to – they do not form part of the category name:

Assign categories to each lexical item:
MediCenter: NPa.
employed: V.
nurses: NPb.

Check each category to see if it exhausts an RHS:
NPa: no.
V: can be dominated by VPa.
NPb: no.

Check each adjacent pair of categories to see if they exhaust an RHS:
NPa V: no.
NPa VPa: can be dominated by Sa.
V NPb: can be dominated by VPb.
VPa NPb: no.

Check each triple of categories to see if they exhaust an RHS:
NPa V NPb: no.
NPa VPa NPb: no.

Check each new category to see if it exhausts an RHS:
VPa: no.
VPb: no.
Sa: no.

Check each new adjacent pair of categories to see if they exhaust an RHS:
Sa NPb: no.
NPa VPb: can be dominated by Sb. [SUCCESS]

Notice that the left-to-right/right-to-left distinction is of much less potential significance in the context of a breadth-first parser than it is in the case of a depth-first one. Once again, the breadth- versus depth-first distinction is a dimension rather than a simple binary choice for the parser as a whole. It is quite common, for example, for parsers to begin by assigning all the words in the string to their (multiple possible) lexical categories (thus working in a breadth-first manner), but to proceed from then on in a largely depth-first manner.

All our Prolog recognizers and parsers, of course, use depth-first search, as they rely on the depth-first backtracking facility of Prolog to do their searching. For this reason, it is obviously much easier to write depth-first search programs rather than other kinds in Prolog. The use of breadth-first search requires programs that explicitly manipulate lists of alternative states, rather than leaving the task of enumerating and remembering the states to the Prolog execution mechanism.

Exercise 5.17 Implement a version of the breadth-first, bottom-up parser discussed in the text and examine the way it behaves. Then implement a breadth-first, top-down parser and compare its behaviour with your depth-first, top-down parser of Exercise 5.5. [*hard*]

5.7 Storing intermediate results

We noted earlier that a feature of our first parser was that it failed to store any intermediate results. That is a key reason for its obsessively time-wasting activity in rechecking things that it has already checked and which cannot have changed. You can think of it as an amnesiac if you like. But there has been a subtle shift in the breadth-first parser that we have just presented semi-formally: this device checks each *new* category to see if it exhausts an RHS and checks each *new* adjacent pair of categories to see if they exhaust an RHS. This makes it slightly harder to program, since the implementation now has to make it possible to keep a record of which categories and pairs have already been considered. Indeed, the program we have shown does not have this feature. But the pay-off is a more efficient parser that does not spend its time thrashing about exploring dead ends that it has already been down half a dozen times. This issue of storage of intermediate results is independent of the three distinctions already discussed, even though any parser needs to store *some* states, simply to remember what it is currently doing; in particular, a breadth-first parser, of necessity, has to remember the multiple hypothesis states that are currently being entertained. Storage of intermediate results also turns out to be the key to efficient parsing in a way that the other distinctions are not – at least, not in general. In Chapter 6, we will look in some detail at chart parsers, which are parsers that use a data structure called a chart to encode all intermediate results obtained in the course of a parse. The cost of this increase in efficiency is not simply one of implementation complexity: there is also a cost in terms of the space requirements of the parsing program.

Such a time versus space trade-off is a familiar one in computer science. Consequently, we would expect simple-minded amnesiac phrase structure parsers, like the one we have sketched out, to be linear in space, but exponential in time, in the worst case. This means that, at worst, they will make space (that is, memory) demands that vary in direct proportion (hence linear) to the length of the input string measured, in number of words, which is a very parsimonious use of space, but make time demands proportional to a constant k raised to the power of the string length measure – thus, $t = k^3$ for a three-word string, $t = k^{11}$ for an 11-word string, and so on. Such a worst-case demand for time is typically quite unacceptable for anything but the shortest strings. By contrast, the kind of phrase structure parser that stores intermediate results, such as those discussed in Chapter 6, can be expected to make space demands that are at worst quadratic in the length of the string; that is, they are proportional to n^2 where n is the length of the string. This is worse than linear but acceptable for strings of the length that are typically used in natural language communication. The time demands, on the other hand, are at worst cubic in the length of the string; that is, they are proportional to n^3 where n is the length of the string. This latter figure is really not good enough, but is vastly better than the exponential worst case just considered, and, in practice, this theoretical worst case of n^3 will be rarely, almost certainly never, encountered with the kinds of grammars appropriate to natural languages and the kind of strings such parsers are standardly faced with. The cubic worst-case time efficiency problem for natural language parsers, to the extent that it is a problem at all, is completely dwarfed in practice by a much more serious problem, that of pervasive natural language ambiguity. It is to this that we turn in the following section. The efficiency figures we have just been considering all relate to the time (or space) taken to recognize a string; that is, to the time to decide whether it is grammatical. The time taken to enumerate all parses of a string is affected by the degree of ambiguity and in the worst case is exponential.

5.8 Ambiguity

Let us add a single entry to the lexicon of Grammar1:

```
Word nurses:
    <cat> = V.
```

This now allows Grammar1 to admit sentences such as 'Dr Chan nurses patients'. What we have done is infect the lexicon of Grammar1 with a

lexical ambiguity caused by the presence of this new entry alongside the existing entry for 'nurses':

```
Word nurses:
    <cat> = NP.
```

The introduction of this lexical ambiguity into Grammar1 does not mean that Grammar1 can now generate any ambiguous sentences – it cannot. If 'nurses' occurs as the first word or the third word of a Grammar1 sentence, then it must be of category NP, whereas if it occurs as the second word, then it must be of category V. However, this lexical ambiguity is going to cause extra work for at least one of the parsers already considered: a bottom-up parser finding 'nurses' will assign it to both the V category and the NP category, and explore where each trail leads.

Lexical ambiguity of this kind is omnipresent in languages like English. Most English words are nouns and many English nouns can be used as verbs. Moreover the ending '-s' is used both by almost all English nouns as their orthographic plural form and by almost all English verbs as their orthographic third-person singular form. The result is a large number of possibilities for ambiguity. To take another example from the many available, consider words of the form 'Xer'. Many of these, such as 'smoother' and 'cleaner', can either mean 'one that Xes' or mean 'more X'. Or consider words ending in '-ing' and compare the adjectival sense found in 'driving rain' with the verbal sense found in 'driving Chevrolets'. Open any English dictionary at a randomly selected page and look at the words listed on it: most of them will be given as belonging to more than one part of speech, or more than one syntactic category in our terminology, and most of them will be given as having more than one meaning. If a parser is part of a system intended to map English sentences into meanings, then it has to cope with both these aspects of lexical ambiguity. In contrast, an experimental or toy system engaged simply in assigning syntactic structures to strings can, if it chooses, simply ignore lexical meaning ambiguities that are not paralleled by a categorial ambiguity. Thus, for example, the semantic ambiguity of the noun 'bank' has no syntactic correlate in American English. (In British English, 'the bank are pressing me for repayment' is grammatical on the financial institution reading, but Americans find it ungrammatical on either reading.)

There is a distinction to be made in principle between lexical ambiguity and structural ambiguity although, in practice, the former often gives rise to the latter, and is thus bound up with it. An American English sentence like 'I walked to the bank' contains a purely lexical ambiguity due to the two different senses of the noun 'bank': no structural ambiguity is involved. By contrast, a sentence like 'I observed the boy with a telescope' contains a purely structural ambiguity, corresponding to whether the speaker or the boy is in possession of the telescope, and no relevant lexical ambiguities. Often, we find the two together, as in the classic 'I saw her

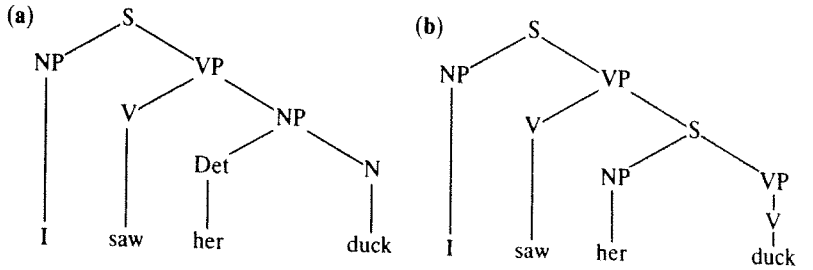

Figure 5.6 An example showing combined lexical and structural ambiguity.

duck', where the structural ambiguity between sentential and NP complements to the verb 'saw' is triggered by the lexical ambiguity of 'her' (accusative pronominal NP versus possessive pronominal determiner) and 'duck' (verb of motion versus noun naming species of aquatic fowl), as illustrated in Figure 5.6.

Notice that ambiguity is potentially multiplicative rather than simply additive: a string containing one 2-way ambiguous item and one 3-way ambiguous item could, although it need not, have as many as six different structures (or meanings) as a result, not merely five as addition would suggest. Sometimes, other syntactic considerations ensure that the actual number of readings is less than the arithmetical potential maximum. Thus, 'I saw her duck' only has two readings, not four, despite the presence of two 2-way lexically ambiguous items in the string. Often, however, especially where purely structural ambiguity is concerned, the actual number of available readings coincides with or approaches the arithmetic maximum. What is disturbing for parser and grammar designers is that this figure can get very large, even for strings of a length typically encountered in practice. We will take a look at the primary sources of such multiplicative ambiguity shortly, but before we do so there are one or two matters in need of clarification.

In talking about ambiguity in this book, we are always concerned with what might be called grammatically available ambiguity, as opposed to some more intuitive notion. Thus, intuitively, 'flying planes made her duck' is unambiguous and has only the reading in which planes that were flying caused an animate female entity to move the upper part of her body to avoid them. But our intuitions make only this reading available to us (unreflectingly, at least) because, presumably, they are informed by a knowledge of planes, flying, and so on. As far as the syntax of English is concerned, and thus as far as a syntactically based parser for English is concerned, this sentence is ambiguous and has no less than four distinct structures associated with it, only one of which corresponds to the intuitively perceived reading. The expression 'flying planes' has two structures, which can be paraphrased as 'planes that fly' and 'the act of flying planes', as does 'made her duck', and these two 2-way ambiguities multiply out.

This may seem perverse, but it is virtually inescapable: only a parser that could bring the meanings of the subexpressions together with sophisticated, wide-ranging, real-world knowledge (about ducks, planes, and so forth) could hope to escape constructing the three intuitively bizarre but grammatically available readings in a principled manner. In Chapters 8 and 10, we will consider limited ways in which semantic and pragmatic information can be used to filter out inappropriate parses, but there is no known general solution to the problem.

Returning to the main thread of this section, we will now consider the three major sources of pure structural ambiguity in English (and many other languages):

- PP attachment,
- coordination, and
- noun–noun compounding.

In English, prepositional phrases (PPs) can attach themselves, normally in a constituent-final position, to constituents of almost any syntactic category – sentences, verb phrases, noun phrases and adjectival phrases to name but four. This means that if we have a nest of constituents, all of them nested to the right, and a single final PP, then we have a degree of ambiguity that varies with the depth of the nest. This is best illustrated diagrammatically, as shown in Figure 5.7. Often, the differences in interpretation between these attachment positions can be quite subtle. Consider, for example, the various readings of 'I can leave for 10 minutes'.

If you look back to Grammar2 in Chapter 4, you will see a rule for coordination which, in a slightly simplified form, looks like this:

```
Rule {coordination}
  X → X C X.
```

where X can be any category and C can be realized as 'and' or 'or'. There is a small but crucial difference between this rule and the rule that you will find given, implicitly or explicitly, in logic textbooks for the logical counterparts of English 'and' and 'or'. The rule logicians use, in our notation, is:

```
Rule {unambiguous coordination}
  X → (X C X).
```

The parentheses, which are expressions of the language being defined although not expressions having a meaning of their own, make all the difference in that they ensure that the sentences of logic are never ambiguous: you have to say either '(P and (Q or R))' or '((P and Q) or R)' and you are not allowed to say 'P and Q or R'. English and many other languages are just the opposite: they more or less force an ambiguous form

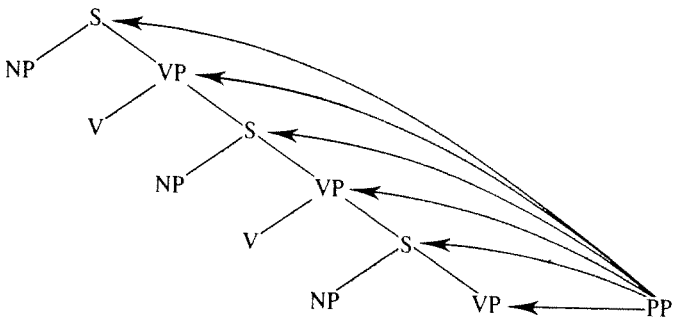

Figure 5.7 Possibilities for PP attachment.

on the user. Since coordination, like modification by PP, is one of the most frequently used constructions, the ambiguity of sentences like 'Mayumi and Hans or Beryl left early' is a real problem for natural language parsers. Programming languages sometimes *look* as if they are as ambiguous as English in respect of such constructions, but appearances are misleading: normally, the language defines an explicit precedence hierarchy among the operators and this eliminates the potential for ambiguity.

There is a sense in which the phrase 'widget hammer' is ambiguous: it could mean the same as 'widget used as hammer', 'hammer for hitting widgets', 'hammer used by widgets' and numerous other things beside. But this ambiguity is not structural. There is only one way in which two nouns can be put together in a fixed order. But now consider 'town widget hammer'. This is at least 2-way structurally ambiguous and can be parsed as ((town widget) hammer) or (town (widget hammer)), and thus gives rise to just the same issue as coordination. Like the latter, noun–noun compounding is extremely common in English, especially official, technical and bureaucratic English. You do not have to look hard to find things like 'Judiciary plea bargain settlement account audit'. Assuming that only binary noun combination is permitted, this innocent-looking bit of jargon has no less than 42 distinct structural descriptions.

Now consider a sentence from a press release with a couple of noun–noun compounds, a couple of PP modifiers at the end, as well as a few 'ands' and 'ors' for good measure. Since ambiguities multiply out, such a sentence may have literally tens of thousands of different structural descriptions, only one of which is associated with the intended meaning of the sentence used. This kind of ambiguity is not a theoretical problem (at least, not in the present context), but it is a very real practical problem for those who have to write parsers that are actually expected to form a component of a useful natural language system. In fact, there are two problems, one of which is how to determine what the intended structure is, which is not really the parser's problem at all. The other problem is how to cope with the sheer size of the set of partial structures that will be created

during the parse of a fairly ambiguous string. A partial solution to the latter problem is provided by the chart data structure, introduced in Chapter 6, but more radical moves are probably called for.

Exercise 5.18 Write a small grammar for a fragment of English with, say, a dozen or more words in its lexicon, each of which belongs to at least two syntactic categories used by the syntactic rules. Study the behaviour, when using this grammar, of the various parsers developed in response to earlier exercises. [*easy*]

Exercise 5.19 Devise a grammar and lexicon that will allow for the ambiguity of 'I saw her duck' and 'flying planes made her duck', and examine the behaviour of the various parsers already developed on these sentences. [*easy*]

Exercise 5.20 Devise a grammar that will allow PP modifiers, coordination and noun-noun compounding. Using one of the parsers already written, explore how many distinct parses you get for a range of examples admitted by this grammar. [*intermediate*]

5.9 Determinism and lookahead

Our discussion in the previous section concerned itself almost wholly with what we shall call global ambiguity; that is, cases where an expression is correctly assigned two or more structures and where those structures persist, or carry over, into the larger structures of which the expression forms a part. But of equal concern to the designer of parsers is the phenomenon of local ambiguity; that is, cases where an expression is correctly assigned two or more structures and where those structures may or may not persist, or carry over, into the larger structures of which the expression forms a part. The parsing of a wholly unambiguous sentence may nonetheless involve the exploration of a great deal of local ambiguity en route to the ultimate, single, structural description of the whole. Consider, for example, a right-to-left, bottom-up, depth-first parser working on the sentence:

> The surgeon whose most famous pupil was Dr Chan founded MediCenter.

Such a parser may well try to build an analysis in which 'Dr Chan founded MediCenter' is a constituent, even though, ultimately, no such constituent

is involved in the structure of the sentence. This sort of activity, as opposed to activity spent on global ambiguities, is time wasting, and much effort and ingenuity in computational linguistics has gone into trying to find ways to minimize or eliminate it.

One project aimed at its elimination can usefully be called the 'determinism' project, inspired originally by the work of Marcus. The aim of this project is to construct parsers that will parse (unambiguous) sentences deterministically; that is, construct parsers that (almost) never get fooled by local ambiguities, never have to backtrack, never change their mind, but which march inexorably to the single, correct, structural description. Why '(almost) never'? Well, 'never' may well be too ambitious. The human parser gags on certain unambiguous strings – those that have come to be known as garden path sentences, as in:

> The horse raced past the barn fell.
>
> The prime number few.
>
> The granite rocks during the earthquake.

Since humans have difficulty with such examples, it is not unreasonable to permit a computer parser to also. In fact, many of those engaged in the determinism project have had cognitive science goals in mind, rather than engineering ones, since they have been attempting to model what they take the human parser to be like, not attempting to build fast parsers that never fail on well-formed strings.

Lookahead involves looking a bounded distance ahead in the string before deciding how to interpret a word or sequence of words. It can be regarded as a very constrained form of backtracking. Information about the frequency of different syntactic constructions may well have played a part in the evolution of the human parser and can be used in making plausible decisions, when lookahead does not suggest a single possibility. Deterministic parsers generally achieve adequate performance using a combination of lookahead and heuristics about the relative frequencies of different constructions.

Although Marcus-type parsers will only parse CFLs, they employ a more procedural notion of syntactic rule than that considered in Chapter 4. More recent work in this tradition has, however, employed the CF-PSG format for grammars and attempted to show how existing deterministic parsing techniques from computer science, such as shift–reduce parsing, can be used directly or else extended to handle natural language input. This work has begun to exploit, and explicate, two interesting observations made by psycholinguists about the way in which the human parser appears to work – namely, by 'right association' and 'minimal attachment'. The former says that, ceteris paribus, phrases get attached as far to the right as they can be, whereas latter says that, ceteris paribus, phrases are attached in a way that minimizes the amount of structure that has to be built

SUMMARY

- A parser maps strings into their structures.
- Bottom-up parsers work up from the words in the string.
- Top-down parsers work down from syntactic category goals.
- Parsing involves search.
- Depth-first search pursues one hypothesis at a time.
- Breadth-first search pursues hypotheses in parallel.
- Simple parsers do not store intermediate results.
- Storage of intermediate results is the key to efficient parsing.
- Global and local ambiguity are key problems in parsing.
- Deterministic parsers seek to eliminate local ambiguity.

Further reading

Good introductions to the technical side of context-free parsing algorithms can be found in Aho and Ullman (1972, Sections 3.4, 4.1 and 4.2 and 1977, Chapters 4 and 5). Petrick (1987) presents an NLP perspective on parsing issues. Linguistic and cognitive science perspectives on some of these issues are developed in Johnson-Laird (1983, Chapters 12 and 13), Dowty *et al.* (1985), Pulman (1983) and Sampson (1983a). The application of context-free parsing algorithms in the NLP context is discussed by Gazdar and Pullum (1985), Martin *et al.* (1987), Tomita (1986) and Winograd (1983, Sections 3.1–3.5, 6.7), among many others. CF-PSG-based NLP parsers have even been implemented in hardware – see Sanamrad *et al.*(1987). The technique for bottom-up parsing of CF-PSGs presented in the text is due to Matsumoto *et al.* (1983), and a variety of interpreters or compilers for CF-PSG rules in DCG format are to be found in Matsumoto *et al.* (1985), Matsumoto (1987), Matsumoto and Sugimura (1987), Okunishi *et al.* (1988), and Pereira and Shieber (1987, Chapter 6). Other relevant work on CF-PSG parsing in Prolog includes Miyoshi and Furukawa (1985), and Uehara *et al.* (1984).

Discussion of ambiguity is pervasive in the NLP literature. See Alshawi (1987), Heidorn (1982), Hirst (1986), Sparck Jones (1983) and Tait (1983), for a representative sample, while the sheer size of the problem posed by multiple structural ambiguities is established in Church and Patil (1982).

Marcus's approach to deterministic parsing is most fully developed in Marcus (1980), but Charniak (1983), Ritchie (1983a), Sampson (1983b), Thompson and Ritchie (1984, pp. 263–84), Winston (1984, pp. 307–14) and Winograd (1983, pp. 402–10) provide tutorial material on the topic, while Milne (1982) makes proposals to extend the coverage of Marcus-type parsers. A proof that Marcus-type parsers will only parse CFLs is to be found in Nozohoor-Farshi (1987).

The notion of right association was introduced by Kimball (1973) and that of minimal attachment by Frazier and Fodor (1978). There is extensive subsequent literature on the claims these authors make, some of which call into question its empirical basis – see Schubert (1986) and the references therein. Work by Shieber (1983), Tomita (1984) and Pereira (1985) advocates the use of shift–reduce parsers in NLP to account for attachment preferences. Aho and Ullman (1972, Chapter 5) provide a thorough introduction to a variety of shift–reduce parsing algorithms. Abramson (1986) and Stabler (1983) outline deterministic parsing techniques in Prolog.