10 PARSING WITH CONTEXT-FREE GRAMMARS

There are and can exist but two ways of investigating and discovering truth. The one hurries on rapidly from the senses and particulars to the most general axioms, and from them... derives and discovers the intermediate axioms. The other constructs its axioms from the senses and particulars, by ascending continually and gradually, till it finally arrives at the most general axioms.

Francis Bacon, *Novum Organum* Book I.19 (1620)

By the 17th century, the western philosophical tradition had begun to distinguish two important insights about human use and acquisition of knowledge. The **empiricist** tradition, championed especially in Britain, by Bacon and Locke, focused on the way that knowledge is induced and reasoning proceeds based on data and experience from the external world. The **rationalist** tradition, championed especially on the Continent by Descartes but following a tradition dating back to Plato's Meno, focused on the way that learning and reasoning is guided by prior knowledge and innate ideas.

EMPIRICIST

RATIONALIST

This dialectic continues today, and has played a important role in characterizing algorithms for **parsing**. We defined parsing in Chapter 3 as a combination of recognizing an input string and assigning some structure to it. Syntactic parsing, then, is the task of recognizing a sentence and assigning a syntactic structure to it. This chapter focuses on the kind of structures assigned by the context-free grammars of Chapter 9. Since context-free grammars are a declarative formalism, they don't specify how the parse tree for a given sentence should be computed. This chapter will, therefore, present some of the many possible algorithms for automatically assigning a context-free (phrase structure) tree to an input sentence.

Parse trees are directly useful in applications such as **grammar checking** in word-processing systems; a sentence which cannot be parsed may have grammatical errors (or at least be hard to read). In addition, parsing is an important intermediate stage of representation for **semantic analysis** (as we will see in Chapter 15), and thus plays an important role in applications like **machine translation**, **question answering**, and **information extraction**. For example, in order to answer the question

What books were written by British women authors before 1800?

we'll want to know that the subject of the sentence was *what books* and that the *by-adjunct* was *British women authors* to help us figure out that the user wants a list of books (and not just a list of authors). Syntactic parsers are also used in lexicography applications for building on-line versions of dictionaries. Finally, stochastic versions of parsing algorithms have recently begun to be incorporated into **speech recognizers**, both for **language models** (Ney, 1991) and for non-finite-state acoustic and phonotactic modeling (Lari and Young, 1991).

The main parsing algorithm presented in this chapter is the **Earley** algorithm (Earley, 1970), one of the context-free parsing algorithms based on **dynamic programming**. We have already seen a number of dynamic programming algorithms – Minimum-Edit-Distance, Viterbi, Forward. The Earley algorithm is one of three commonly-used dynamic programming parsers; the others are the Cocke-Younger-Kasami (CYK) algorithm which we will present in Chapter 12, and the Graham-Harrison-Ruzzo (GHR) (Graham *et al.*, 1980) algorithm. Before presenting the Earley algorithm, we begin by motivating various basic parsing ideas which make up the algorithm. First, we revisit the 'search metaphor' for parsing and recognition, which we introduced for finite-state automata in Chapter 2, and talk about the **top-down** and **bottom-up** search strategies. We then introduce a 'baseline' top-down backtracking parsing algorithm, to introduce the idea of simple but efficient parsing. While this parser is perspicuous and relatively efficient, it is unable to deal efficiently with the important problem of **ambiguity**: a sentence or words which can have more than one parse. The final section of the chapter then shows how the Earley algorithm can use insights from the top-down parser with bottom-up filtering to efficiently handle ambiguous inputs.

10.1 PARSING AS SEARCH

Chapters 2 and 3 showed that finding the right path through a finite-state automaton, or finding the right transduction for an input, can be viewed as a search problem. For FSAs, for example, the parser is searching through the space of all possible paths through the automaton. In syntactic parsing, the parser can be viewed as searching through the space of all possible parse trees to find the correct parse tree for the sentence. Just as the search space of possible paths was defined by the structure of the FSA, so the search space of possible parse trees is defined by the grammar. For example, consider the following ATIS sentence:

(10.1) Book that flight.

Using the miniature grammar and lexicon in Figure 10.2, which consists of some of the CFG rules for English introduced in Chapter 9, the correct parse tree that would be would assigned to this example is shown in Figure 10.1.

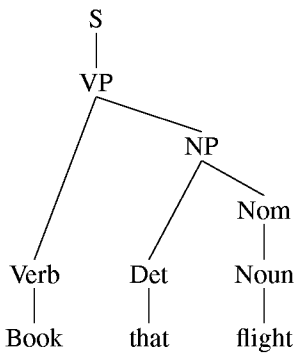

Figure 10.1 The correct parse tree for the sentence *Book that flight* according to the grammar in Figure 10.2.

How can we use the grammar in Figure 10.2 to assign the parse tree in Figure 10.1 to Example (10.1)? (In this case there is only one parse tree, but it is possible for there to be more than one.) The goal of a parsing search is to find all trees whose root is the start symbol *S*, which cover exactly the words in the input. Regardless of the search algorithm we choose, there are clearly two kinds of constraints that should help guide the search. One kind of constraint comes from the data, i.e. the input sentence itself. Whatever else

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

Figure 10.2 A miniature English grammar and lexicon.

is true of the final parse tree, we know that there must be three leaves, and they must be the words *book*, *that*, and *flight*. The second kind of constraint comes from the grammar. We know that whatever else is true of the final parse tree, it must have one root, which must be the start symbol *S*.

These two constraints, recalling the empiricist/rationalist debate described at the beginning of this chapter, give rise to the two search strategies underlying most parsers: **top-down** or **goal-directed search** and **bottom-up** or **data-directed search**.

Top-Down Parsing

TOP-DOWN

A **top-down** parser searches for a parse tree by trying to build from the root node *S* down to the leaves. Let's consider the search space that a top-down parser explores, assuming for the moment that it builds all possible trees in parallel. The algorithm starts by assuming the input can be derived by the designated start symbol *S*. The next step is to find the tops of all trees which can start with *S*, by looking for all the grammar rules with *S* on the left-hand side. In the grammar in Figure 10.2, there are three rules that expand *S*, so

PLY

the second **ply**, or level, of the search space in Figure 10.3 has three partial trees.

We next expand the constituents in these three new trees, just as we originally expanded *S*. The first tree tells us to expect an *NP* followed by a *VP*, the second expects an *Aux* followed by an *NP* and a *VP*, and the third a *VP* by itself. To fit the search space on the page, we have shown in the third ply of Figure 10.3 only the trees resulting from the expansion of the left-most leaves of each tree. At each ply of the search space we use the right-hand-sides of the rules to provide new sets of expectations for the parser, which

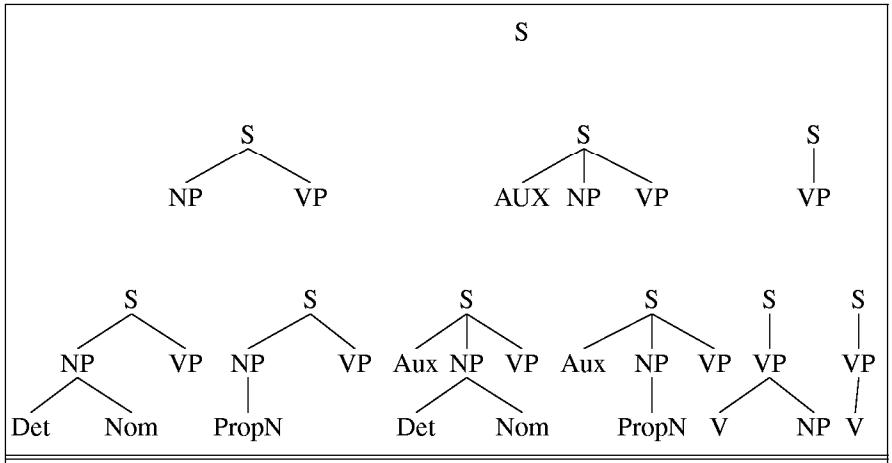

Figure 10.3 An expanding top-down search space. Each ply is created by taking each tree from the previous ply, replacing the leftmost non-terminal with each of its possible expansions, and collecting each of these trees into a new ply.

are then used to recursively generate the rest of the trees. Trees are grown downward until they eventually reach the part-of-speech categories at the bottom of the tree. At this point, trees whose leaves fail to match all the words in the input can be rejected, leaving behind those trees that represent successful parses.

In Figure 10.3, only the 5th parse tree (the one which has expanded the rule $VP \rightarrow Verb\ NP$) will eventually match the input sentence *Book that flight*. The reader should check this for themselves in Figure 10.1.

Bottom-Up Parsing

Bottom-up parsing is the earliest known parsing algorithm (it was first sug- BOTTOM-UP
gested by Yngve (1955)), and is used in the shift-reduce parsers common for computer languages (Aho and Ullman, 1972). In bottom-up parsing, the parser starts with the words of the input, and tries to build trees from the words up, again by applying rules from the grammar one at a time. The parse is successful if the parser succeeds in building a tree rooted in the start symbol *S* that covers all of the input. Figure 10.4 show the bottom-up search space, beginning with the sentence *Book that flight*. The parser begins by looking up each word (*book*, *that*, and *flight*) in the lexicon and building three partial trees with the part of speech for each word. But the word *book*

is ambiguous; it can be a noun or a verb. Thus the parser must consider two possible sets of trees. The first two plies in Figure 10.4 show this initial bifurcation of the search space.

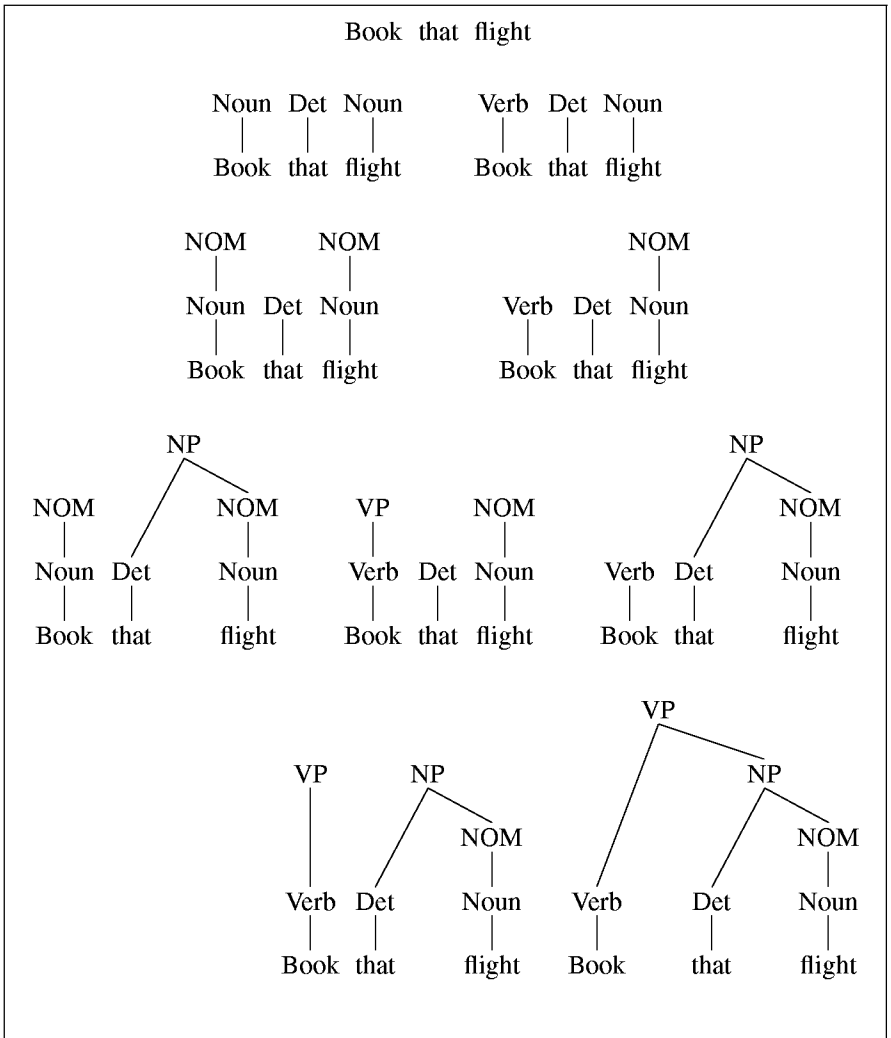

Figure 10.4 An expanding bottom-up search space for the sentence *Book that flight*. This figure does not show the final tier of the search with the correct parse tree (see Figure 10.1). Make sure you understand how that final parse tree follows from the search space in this figure.

Each of the trees in the second ply are then expanded. In the parse

on the left (the one in which *book* is incorrectly considered a noun), the *Nominal* → *Noun* rule is applied to both of the *Noun*s (*book* and *flight*). This same rule is also applied to the sole *Noun* (*flight*) on the right, producing the trees on the third ply.

In general, the parser extends one ply to the next by looking for places in the parse-in-progress where the right-hand-side of some rule might fit. This contrasts with the earlier top-down parser, which expanded trees by applying rules when their left-hand side matched an unexpanded nonterminal. Thus in the fourth ply, in the first and third parse, the sequence *Det Nominal* is recognized as the right-hand side of the *NP* → *Det Nominal* rule.

In the fifth ply, the interpretation of *book* as a noun has been pruned from the search space. This is because this parse cannot be continued: there is no rule in the grammar with the right-hand side *Nominal NP*.

The final ply of the search space (not shown in Figure 10.4) is the correct parse tree (see Figure 10.1). Make sure you understand which of the two parses on the penultimate ply gave rise to this parse.

Comparing Top-down and Bottom-up Parsing

Each of these two architectures has its own advantages and disadvantages. The top-down strategy never wastes time exploring trees that cannot result in an *S*, since it begins by generating just those trees. This means it also never explores subtrees that cannot find a place in some *S*-rooted tree. In the bottom-up strategy, by contrast, trees that have no hope of leading to an *S*, or fitting in with any of their neighbors, are generated with wild abandon. For example the left branch of the search space in Figure 10.4 is completely wasted effort; it is based on interpreting *book* as a *Noun* at the beginning of the sentence despite the fact no such tree can lead to an *S* given this grammar.

The top-down approach has its own inefficiencies. While it does not waste time with trees that do not lead to an *S*, it does spend considerable effort on *S* trees that are not consistent with the input. Note that the first four of the six trees in the third ply in Figure 10.3 all have left branches that cannot match the word *book*. None of these trees could possibly be used in parsing this sentence. This weakness in top-down parsers arises from the fact that they can generate trees before ever examining the input. Bottom-up parsers, on the other hand, never suggest trees that are not at least locally grounded in the actual input.

Neither of these approaches adequately exploits the constraints presented by the grammar and the input words. In the next section, we present

a baseline parsing algorithm that incorporates features of both the top-down and bottom-up approaches. This parser is not as efficient as the Earley or CYK parsers we will introduce later, but it is useful for showing the basic operations of parsing.

10.2 A Basic Top-down Parser

There are any number of ways of combining the best features of top-down and bottom-up parsing into a single algorithm. One fairly straightforward approach is to adopt one technique as the primary control strategy used to generate trees and then use constraints from the other technique to filter out inappropriate parses on the fly. The parser we develop in this section uses a top-down control strategy augmented with a bottom-up filtering mechanism. Our first step will be to develop a concrete implementation of the top-down strategy described in the last section. The ability to filter bad parses based on bottom-up constraints from the input will then be grafted onto this top-down parser.

PARALLEL

In our discussions of both top-down and bottom-up parsing, we assumed that we would explore all possible parse trees in **parallel**. Thus each ply of the search in Figure 10.3 and Figure 10.4 showed all possible expansions of the parse trees on the previous plies. Although it is certainly possible to implement this method directly, it typically entails the use of an unrealistic amount of memory to store the space of trees as they are being constructed. This is especially true since realistic grammars have much more ambiguity than the miniature grammar in Figure 10.2.

DEPTH-FIRST STRATEGY

A more reasonable approach is to use a **depth-first strategy** such as the one used to implement the various finite state machines in Chapter 2 and Chapter 3. The depth-first approach expands the search space incrementally by systematically exploring one state at a time. The state chosen for expansion is the most recently generated one. When this strategy arrives at a tree that is inconsistent with the input, the search continues by returning to the most recently generated, as yet unexplored, tree. The net effect of this strategy is a parser that single-mindedly pursues trees until they either succeed or fail before returning to work on trees generated earlier in the process. Figure 10.5 illustrates such a top-down, depth-first derivation using Grammar 10.2.

Note that this derivation is not fully determined by the specification of a top-down, depth-first strategy. There are two kinds of choices that have been left unspecified that can lead to different derivations: the choice of which

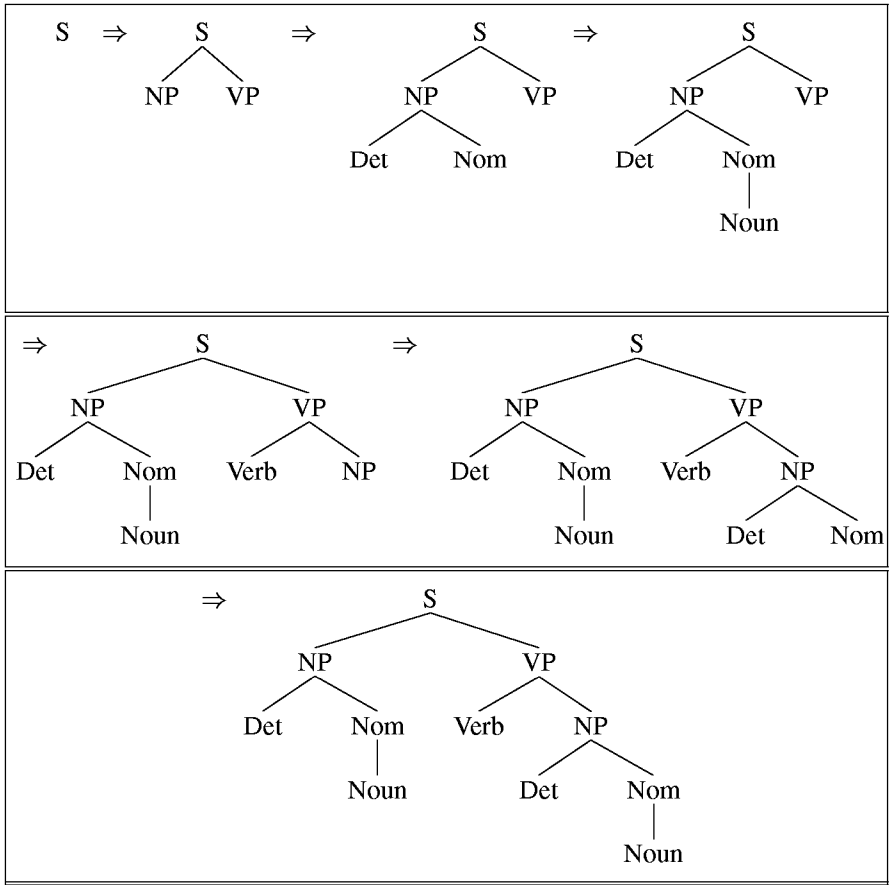

Figure 10.5 A top-down depth-first derivation with the grammar in Figure 10.2.

leaf node of a tree to expand and the order in which applicable grammar rules are applied. In this derivation, the left-most unexpanded leaf node of the current tree is being expanded first, and the applicable rules of the grammar are being applied according to their textual order in the grammar. The decision to expand the left-most unexpanded node in the tree is important since it determines the order in which the input words will be consulted as the tree is constructed. Specifically, it results in a relatively natural forward incorporation of the input words into a tree. The second choice of applying rules in their textual order has consequences that will be discussed later.

Figure 10.6 presents a parsing algorithm that instantiates this top-down, depth-first, left-to-right strategy. This algorithm maintains an **agenda** of AGENDA

```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

 agenda ← (Initial S tree, Beginning of input)
 current-search-state ← POP(agenda)
 loop
  if SUCCESSFUL-PARSE?(current-search-state) then
    return TREE(current-search-state)
  else
    if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
      if CAT(node-to-expand)
           ⊂
           POS(CURRENT-INPUT(current-search-state)) then
        PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
      else
        return reject
    else
      PUSH(APPLY-RULES(current-search-state, grammar), agenda)
  if agenda is empty then
     return reject
  else
     current-search-state ← NEXT(agenda)
 end
```

Figure 10.6 A top-down, depth-first left-to-right parser.

search-states. Each search-state consists of partial trees together with a pointer to the next input word in the sentence.

The main loop of the parser takes a state from the front of the agenda and produces a new set of states by applying all the applicable grammar rules to the left-most unexpanded node of the tree associated with that state. This set of new states is then added to the front of the agenda in accordance with the textual order of the grammar rules that were used to generate them. This process continues until either a successful parse tree is found or the agenda is exhausted indicating that the input can not be parsed.

Figure 10.7 shows the sequence of states examined by this algorithm in the course of parsing the following sentence.

(10.2) Does this flight include a meal?

In this figure, the node currently being expanded is shown in a box, while the current input word is bracketed. Words to the left of the bracketed word

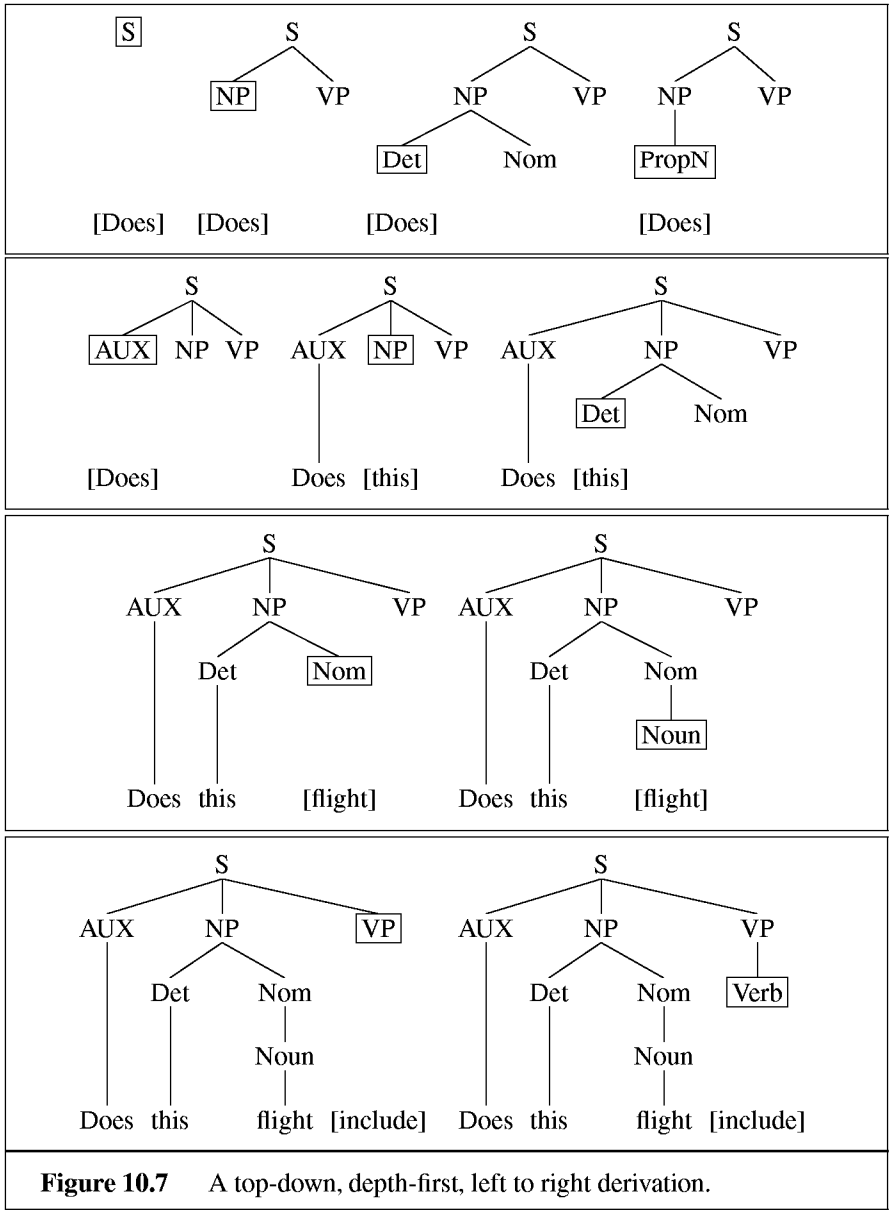

Figure 10.7 A top-down, depth-first, left to right derivation.

have already been incorporated into the tree.

The parser begins with a fruitless exploration of the $S \rightarrow NP\ VP$ rule, which ultimately fails because the word *Does* cannot be derived from any of the parts-of-speech that can begin an *NP*. The parser thus eliminates the

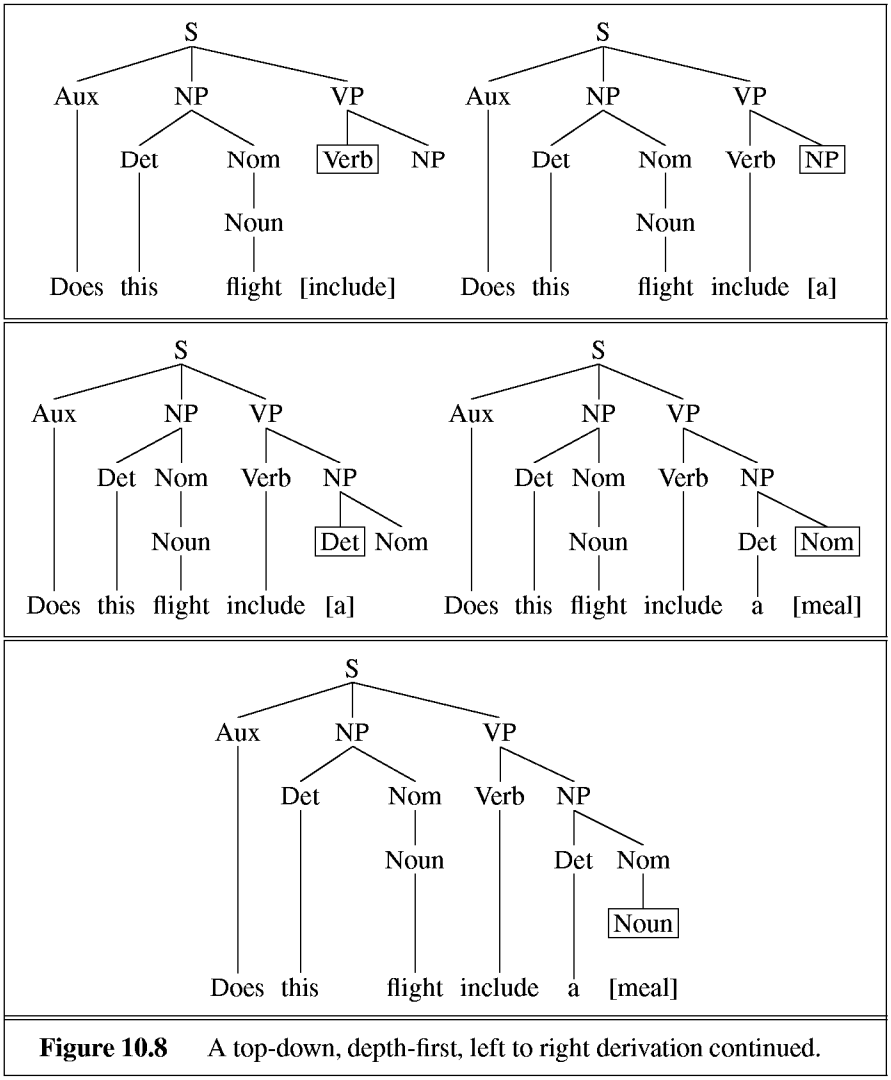

Figure 10.8 A top-down, depth-first, left to right derivation continued.

$S \rightarrow NP\ VP$ rule. The next search-state on the agenda corresponds to the $S \rightarrow Aux\ NP\ VP$ rule. Once this state is found, the search continues in a straightforward depth-first, left to right fashion through the rest of the derivation.

Adding Bottom-up Filtering

Figure 10.7 shows an important qualitative aspect of the top-down parser. Beginning at the root of the parse tree, the parser expands non-terminal symbols along the left edge of the tree, down to the word at the bottom left edge of the tree. As soon as a word is incorporated into a tree, the input pointer moves on, and the parser will expand the new next left-most open non-terminal symbol down to the new left corner word.

Thus in any successful parse the current input word must serve as the first word in the derivation of the unexpanded node that the parser is currently processing. This leads to an important consequence which will be useful in adding bottom-up filtering. The parser should not consider any grammar rule if the current input cannot serve as the *first word along the left edge of some derivation* from this rule. We call the first word along the left edge of a derivation the **left-corner** of the tree. LEFT-CORNER

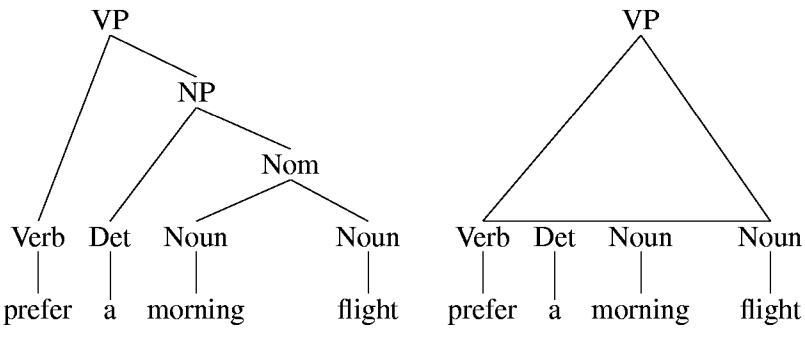

Figure 10.9 An illustration of the left-corner notion. The node *Verb* and the node *prefer* are both left-corners of *VP*.

Consider the parse tree for a *VP* shown in Figure 10.9. If we visualize the parse tree for this *VP* as a triangle with the words along the bottom, the word *prefer* lies at the lower left-corner of the tree. Formally, we can say that for non-terminals A and B, B is a left-corner of A if the following relation holds:

$$A \stackrel{*}{\Rightarrow} B\alpha$$

In other words, B can be a left-corner of A if there is a derivation of A that begins with a B.

We return to our example sentence *Does this flight include a meal?* The grammar in Figure 10.2 provides us with three rules that can be used to

expand the category *S*:

$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$
$S \rightarrow VP$

Using the left-corner notion, it is easy to see that only the $S \rightarrow Aux\ NP\ VP$ rule is a viable candidate since the word *Does* can not serve as the left-corner of either the *NP* or the *VP* required by the other two *S* rules. Knowing this, the parser should concentrate on the *Aux NP VP* rule, without first constructing and backtracking out of the others, as it did with the non-filtering example shown in Figure 10.7.

The information needed to efficiently implement such a filter can be compiled in the form of a table that lists all the valid left-corner categories for each non-terminal in the grammar. When a rule is considered, the table entry for the category that starts the right hand side of the rule is consulted. If it fails to contain any of the parts-of-speech associated with the current input then the rule is eliminated from consideration. The following table shows the left-corner table for Grammar 10.2.

Category	Left Corners
S	Det, Proper-Noun, Aux, Verb
NP	Det, Proper-Noun
Nominal	Noun
VP	Verb

Using this left-corner table as a filter in the parsing algorithm of Figure 10.6 is left as Exercise 10.1 for the reader.

10.3 Problems with the Basic Top-down Parser

Even augmented with bottom-up filtering, the top-down parser in Figure 10.7 has three problems that make it an insufficient solution to the general-purpose parsing problem. These three problems are **left-recursion**, **ambiguity**, and **inefficient reparsing of subtrees**. After exploring the nature of these three problems, we will introduce the Earley algorithm which is able to avoid all of them.

Left-Recursion

Depth-first search has a well-known flaw when exploring an infinite search space: it may dive down an infinitely-deeper path and never return to visit the unexpanded states. This problem manifests itself in top-down, depth-first, left-to-right parsers when **left-recursive grammars** are used. Formally, a grammar is left-recursive if it contains at least one non-terminal A, such that $A \stackrel{*}{\Rightarrow} \alpha A \beta$, for some α and β and $\alpha \stackrel{*}{\Rightarrow} \varepsilon$. In other words, a grammar is left-recursive if it contains a non-terminal category that has a derivation that includes itself anywhere along its leftmost branch. The grammar of Chapter 9 had just such a left-recursive example, in the rules for possessive *NP*s like *Atlanta's airport*:

LEFT-RECURSIVE GRAMMARS

$$NP \rightarrow Det\ Nominal$$
$$Det \rightarrow NP\ 's$$

These rules introduce left-recursion into the grammar since there is a derivation for the first element of the *NP*, the *Det*, that has an *NP* as its first constituent.

A more obvious and common case of left-recursion in natural language grammars involves immediately **left-recursive rules**. These are rules of the form $A \rightarrow A\ \beta$, where the first constituent of the right hand side is identical to the left hand side. The following are some of the immediately left-recursive rules that make frequent appearances in grammars of English.

LEFT-RECURSIVE RULES

$$NP \rightarrow NP\ PP$$
$$VP \rightarrow VP\ PP$$
$$S \rightarrow S\ and\ S$$

A left-recursive non-terminal can lead a top-down, depth-first left-to-right parser to recursively expand the same non-terminal over again in exactly the same way, leading to an infinite expansion of trees.

Figure 10.10 shows the kind of expansion that accompanies the addition of the $NP \rightarrow NP\ PP$ rule as the first *NP* rule in our small grammar.

There are two reasonable methods for dealing with left-recursion in a backtracking top-down parser: rewriting the grammar, and explicitly managing the depth of the search during parsing. Recall from Chapter 9, that it is often possible to rewrite the rules of a grammar into a **weakly equivalent** new grammar that still accepts exactly the same language as the original grammar. It is possible to eliminate left-recursion from certain common classes of grammars by rewriting a left-recursive grammar into a weakly

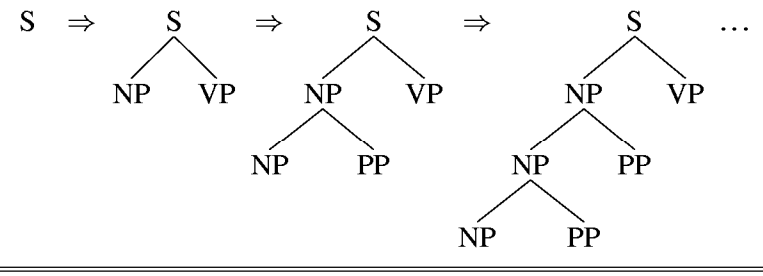

Figure 10.10 The beginning of an infinite search caused by a left recursive rule.

equivalent non-left-recursive one. The intuition is to rewrite each rule of the form $A \rightarrow A\,\beta$ according to the following schema, using a new symbol A':

$$A \rightarrow A\,\beta \mid \alpha \qquad \Rightarrow \qquad \begin{array}{l} A \rightarrow \alpha\,A' \\ A' \rightarrow \beta\,A' \mid \varepsilon \end{array}$$

This transformation changes the left-recursion to a right-recursion, and changes the trees that result from these rules from left-branching structures to a right-branching ones. Unfortunately, rewriting grammars in this way has a major disadvantage: a rewritten phrase-structure rule may no longer be the most grammatically natural way to represent a particular syntactic structure. Furthermore, as we will see in Chapter 15, this rewriting may make semantic interpretation quite difficult.

Ambiguity

> *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.*
>
> Groucho Marx, *Animal Crackers*, 1930

AMBIGUITY

The second problem with the top-down parser of Figure 10.6 is that it is not efficient at handling **ambiguity**. Chapter 8 introduced the idea of **lexical category ambiguity** (words which may have more than one part of speech) and **disambiguation** (choosing the correct part of speech for a word).

In this section we introduce a new kind of ambiguity, which arises in the syntactic structures used in parsing, called **structural ambiguity**. Structural ambiguity occurs when the grammar assigns more than one possible parse to a sentence. Groucho Marx's well-known line as Captain Spaulding for the wavfile) is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or the verb-phrase headed by *shot*.

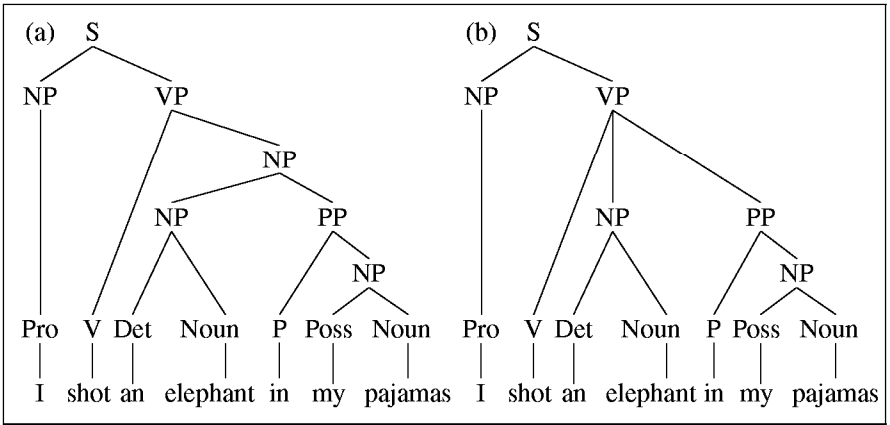

Figure 10.11 Two parse trees for an ambiguous sentence. Parse (a) corresponds to the humorous reading in which the elephant is in the pajamas, parse (b) to the reading in which Captain Spaulding did the shooting in his pajamas.

Structural ambiguity, appropriately enough, comes in many forms. Three particularly common kinds of ambiguity are **attachment ambiguity**, **coordination ambiguity**, and **noun-phrase bracketing ambiguity**.

A sentence has an attachment ambiguity if a particular constituent can be attached to the parse tree at more than one place. The Groucho Marx sentence above is an example of PP-attachment ambiguity. Various kinds of adverbial phrases are also subject to this kind of ambiguity. For example in the following example the gerundive-VP *flying to New York* can be part of a gerundive sentence whose subject is *the Grand Canyon* or it can be an adjunct modifying the VP headed by *saw*:

(10.3) I saw the Grand Canyon flying to New York.

In a similar kind of ambiguity, the sentence "Can you book TWA flights" is ambiguous between a reading meaning 'Can you book flights on behalf of TWA', and the other meaning 'Can you book flights run by TWA'). Here either one NP is attached to another to form a complex NP (*TWA flights*), or both NPs are distinct daughters of the verb phrase. Figure 10.12 shows both parses.

Another common kind of ambiguity is **coordination ambiguity**, in which there are different sets of phrases that can be conjoined by a conjunction like *and*. For example, the phrase *old men and women* can be bracketed *[old [men and women]]*, referring to *old men* and *old women*, or as *[old men] and [women]*, in which case it is only the men who are old.

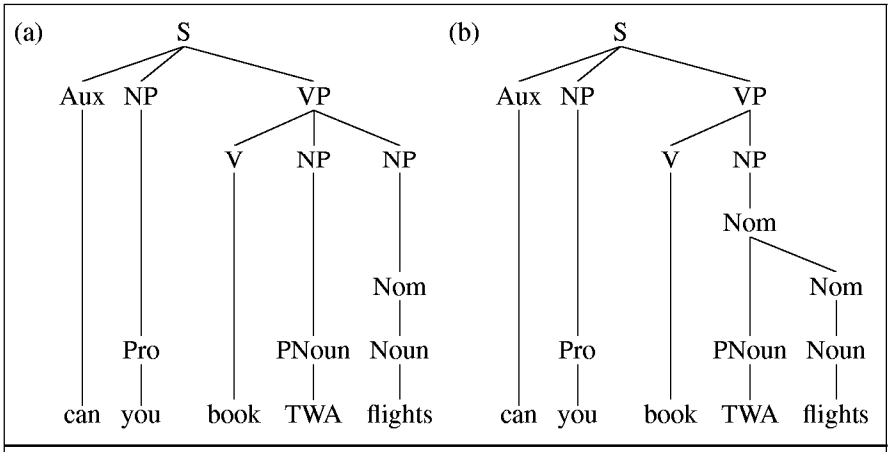

Figure 10.12 Two parse trees for an ambiguous sentence. Parse (a) corresponds to the meaning 'Can you book flights on behalf of TWA?', parse (b) to 'Can you book flights which are run by TWA'.

These ambiguities all combine in complex ways. A program that summarized the news, for example, would need to be able to parse sentences like the following from the Brown corpus :

(10.4) President Kennedy today pushed aside other White House business to devote all his time and attention to working on the Berlin crisis address he will deliver tomorrow night to the American people over nationwide television and radio.

This sentence has a number of ambiguities, although since they are semantically unreasonable, it requires a careful reading to see them. The last noun phrase could be parsed *[nationwide [television and radio]]* or *[[nationwide television] and radio]*. The direct object of *pushed aside* should be *other White House business* but could also be the bizarre phrase *[other White House business to devote all his time and attention to working]* (i.e. a structure like *Kennedy denied [his intention to propose a new budget to address the deficit]*). Then the phrase *on the Berlin crisis address he will deliver tomorrow night to the American people* could be an adjunct modifying the verb *pushed*. The *PP over nationwide television and radio* could be attached to any of the higher *VP*s or *NP*s (for example it could modify *people* or *night*).

The fact that there are many unreasonable parses for a sentence is an extremely irksome problem that affects all parsers. In practice, parsing a sentence thus requires **disambiguation**: choosing the correct parse from a

DISAMBIGUATION

multitude of possible parsers. Disambiguation algorithms generally require both statistical and semantic knowledge, so they will be introduced later, in Chapter 12 and Chapter 17.

Parsers which do not incorporate disambiguators must simply return all the possible parse trees for a given input. Since the top-down parser of Figure 10.7 only returns the first parse it finds, it would thus need to be modified to return all the possible parses. The algorithm would be changed to collect each parse as it is found and continue looking for more parses. When the search space has been exhausted, the list of all the trees found is returned. Subsequent processing or a human analyst can then decide which of the returned parses is correct.

Unfortunately, we almost certainly do not want all possible parses from the robust, highly ambiguous, wide-coverage grammars used in practical applications. The reason for this lies in the potentially exponential number of parses that are possible for certain inputs. Consider the ATIS example (10.5):

(10.5) Show me the meal on Flight UA 386 from San Francisco to Denver.

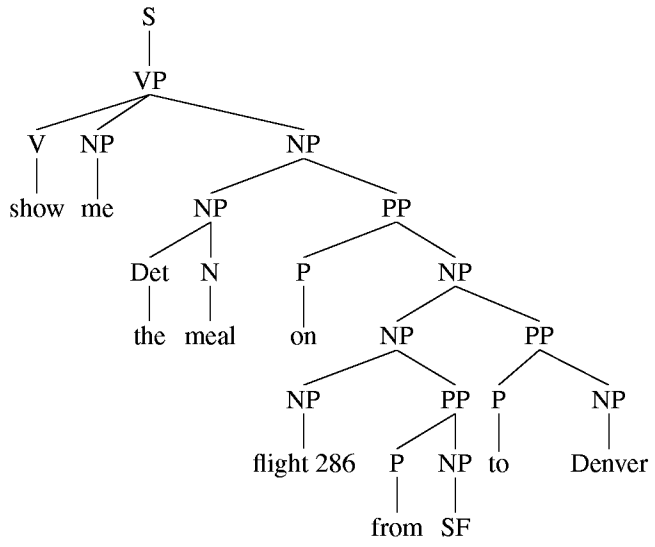

Figure 10.13 A correct parse for (10.5).

When our extremely small grammar is augmented with the recursive $VP \rightarrow VP\ PP$ and $NP \rightarrow NP\ PP$ rules introduced above, the three prepositional phrases at the end of this sentence conspire to yield a total of 14 parse trees

for this sentence. For example *from San Francisco* could be part of the *VP* headed by *show* (which would have the bizarre interpretation that the showing was happening from San Francisco).

Church and Patil (1982) showed that the number of parses for sentences of this type grows at the same rate as the number of parenthesizations of arithmetic expressions. Such parenthesization problems, in turn, are known to grow exponentially in accordance with what are called the Catalan numbers:

$$C(n) = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$$

The following table shows the number of parses for a simple noun-phrase as a function of the number of trailing prepositional phrases. As can be seen, this kind of ambiguity can very quickly make it imprudent to keep every possible parse around.

Number of PPs	Number of NP Parses
2	2
3	5
4	14
5	132
6	469
7	1430
8	4867

There are two basic ways out of this dilemma: using dynamic programming to exploit regularities in the search space so that common subparts are derived only once, thus reducing some of the costs associated with ambiguity, and augmenting the parser's search strategy with heuristics that guide it toward likely parses first. The dynamic programming approach will be explored in the next section, while the heuristic search strategies will be covered in Chapter 12.

LOCAL AMBIGUITY

Even if a sentence isn't ambiguous, it can be inefficient to parse due to **local ambiguity**. Local ambiguity occurs when some part of a sentence is ambiguous, i.e. has more than parse, even if the whole sentence is not ambiguous. For example the sentence *Book that flight* is unambiguous, but when the parser sees the first word *Book*, it cannot know if it is a verb or a noun until later. Thus it must use backtracking or parallelism to consider both possible parses.

Repeated Parsing of Subtrees

The ambiguity problem is related to another inefficiency of the top-down parser of Section 10.2. The parser often builds valid trees for portions of the input, then discards them during backtracking, only to find that it has to rebuild them again. Consider the process involved in finding a parse for the *NP* in (10.6):

(10.6) a flight from Indianapolis to Houston on TWA

The preferred parse, which is also the one found first by the parser presented in Section 10.2, is shown as the bottom tree in Figure 10.14. While there are 5 distinct parses of this phrase, we will focus here on the ridiculous amount repeated work involved in retrieving this single parse.

Because of the way the rules are consulted in our top-down, depth-first, left-to-right approach, the parser is led first to small parse trees that fail because they do not cover all of the input. These successive failures trigger backtracking events which lead to parses that incrementally cover more and more of the input. The sequence of trees attempted by our top-down parser is shown in Figure 10.14.

This figure clearly illustrates the kind of silly reduplication of work that arises in backtracking approaches. Except for its topmost component, every part of the final tree is derived more than once. The following table shows the number of times that each of the major constituents in the final tree is derived. The work done on this example would, of course, be magnified by any backtracking caused by the verb phrase or sentential level. Note, that although this example is specific to top-down parsing, similar examples of wasted effort exist for bottom-up parsing as well.

a flight	4
from Indianapolis	3
to Houston	2
on TWA	1
a flight from Indianapolis	3
a flight from Indianapolis to Houston	2
a flight from Indianapolis to Houston on TWA	1

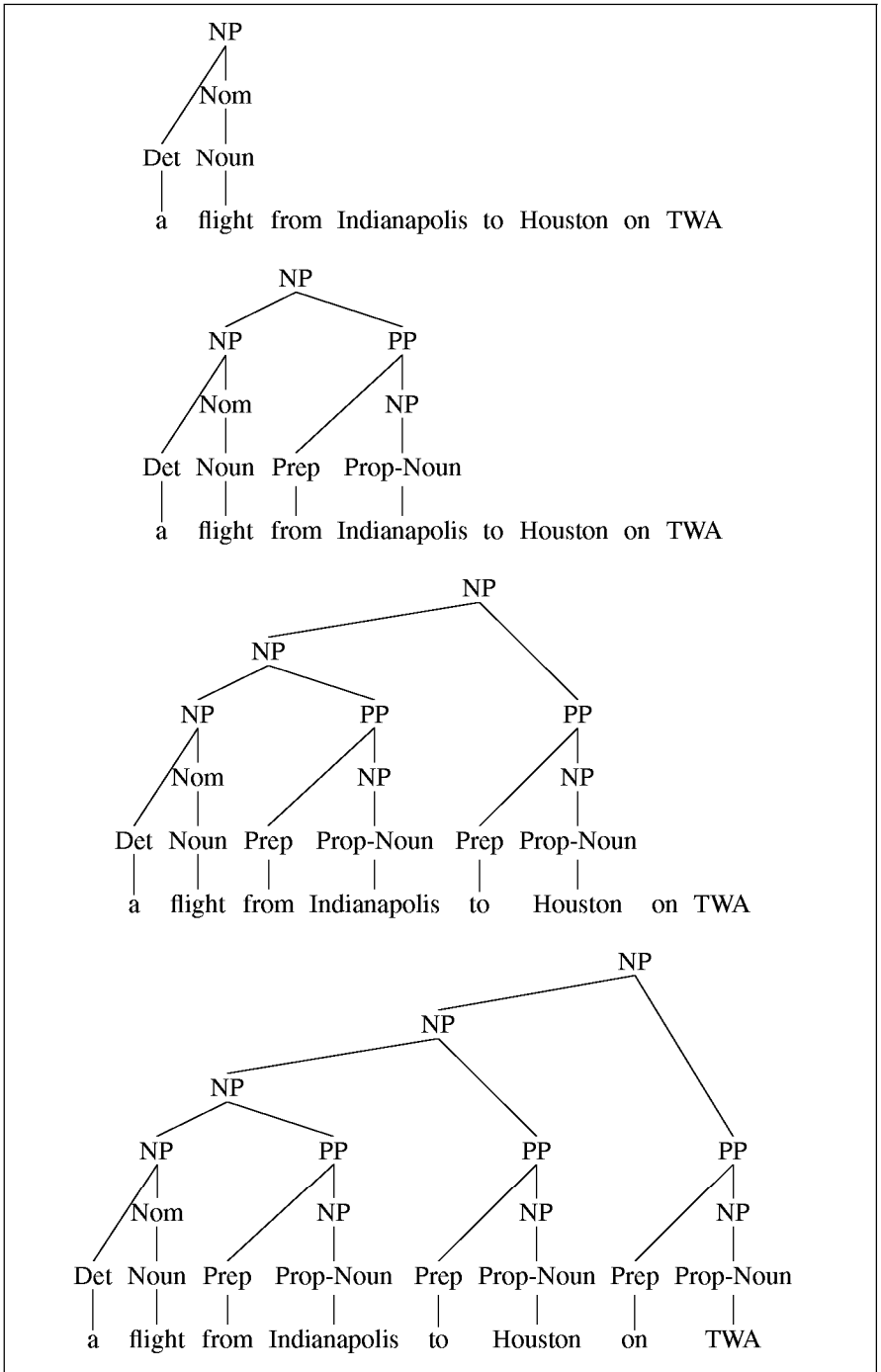

Figure 10.14 Reduplicated effort caused by backtracking in top-down parsing.