ıpter 13
tactic Parsing

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)

We defined parsing in Chapter 3 as a combination of recognizing an input string and assigning a 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 context-free grammars of the kind described in Chapter 12. However, since they are based on a purely declarative formalism, context-free grammars don't specify *how* the parse tree for a given sentence should be computed. We therefore need to specify algorithms that employ these grammars to produce trees. This chapter presents three of the most widely used parsing algorithms for automatically assigning a complete context-free (phrase-structure) tree to an input sentence.

These kinds of parse trees are directly useful in applications such as **grammar checking** in word-processing systems: a sentence that cannot be parsed may have grammatical errors (or at least be hard to read). More typically, however, parse trees serve as an important intermediate stage of representation for **semantic analysis** (as we show in Chapter 18) and thus play an important role in applications like **question answering** and **information extraction**. For example, to answer the question

What books were written by British women authors before 1800?

we'll need 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 a list of authors).

Before presenting any parsing algorithms, we begin by describing some of the factors that motivate the standard algorithms. 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 discuss how the ambiguity problem rears its head again in syntactic processing and how it ultimately makes simplistic approaches based on backtracking infeasible.

The sections that follow then present the Cocke-Kasami-Younger (CKY) algorithm (Kasami, 1965; Younger, 1967), the Earley algorithm (Earley, 1970), and the chart parsing approach (Kay, 1982; Kaplan, 1973). These approaches all combine insights from bottom-up and top-down parsing with dynamic programming to efficiently handle complex inputs. Recall that we've already seen several applications of dynamic

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

Figure 13.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

programming algorithms in earlier chapters — Minimum-Edit-Distance, Viterbi, Forward. Finally, we discuss **partial parsing methods**, for use in situations in which a superficial syntactic analysis of an input may be sufficient.

13.1 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 finite-state automata, the search is through the space of all possible paths through a machine. In syntactic parsing, the parser can be viewed as searching through the space of possible parse trees to find the correct parse tree for a given sentence. Just as the search space of possible paths was defined by the structure of an automaton, so the search space of possible parse trees is defined by a grammar. Consider the following ATIS sentence:

(13.1) Book that flight.

Figure 13.1 introduces the $\mathscr{L}_1$ grammar, which consists of the $\mathscr{L}_0$ grammar from the last chapter with a few additional rules. Given this grammar, the correct parse tree for this example would be the one shown in Fig. 13.2.

How can we use $\mathscr{L}_1$ to assign the parse tree in Fig. 13.2 to this example? The goal of a parsing search is to find all the trees whose root is the start symbol S and that cover exactly the words in the input. Regardless of the search algorithm we choose, two kinds of constraints should help guide the search. One set of constraints comes from the data, that is, the input sentence itself. Whatever else is true of the final parse tree, we know that there must be three leaves and that they must be the words *book*, *that*, and *flight*. The second kind of constraint comes from the grammar. We know that

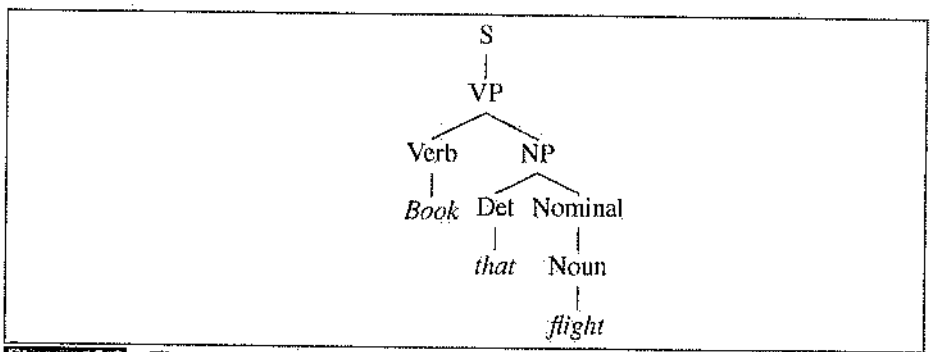


Figure 13.2 The parse tree for the sentence *Book that flight* according to grammar $\mathscr{L}_1$.

whatever else is true of the final parse tree, it must have one root, which must be the start symbol *S*.

These two constraints, invoked by Bacon at the start 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**. These constraints are more than just search strategies. They reflect two important insights in the western philosophical tradition: the **rationalist** tradition, which emphasizes the use of prior knowledge, and the **empiricist** tradition, which emphasizes the data in front of us.

Rationalist
Empiricist

13.1.1 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 that the input can be derived by the designated start symbol *S*. The next step is to find the tops of all trees that can start with *S*, by looking for all the grammar rules with *S* on the left-hand side. In the grammar in Fig. 13.1, three rules expand *S*, so the second **ply**, or level, of the search space in Fig. 13.3 has three partial trees.

Ply

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 Fig. 13.3 only a subset of the trees that result from the expansion of the leftmost 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 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 Fig. 13.3, only the fifth parse tree in the third ply (the one that has expanded the rule $VP \rightarrow Verb\ NP$) will eventually match the input sentence *Book that flight*.

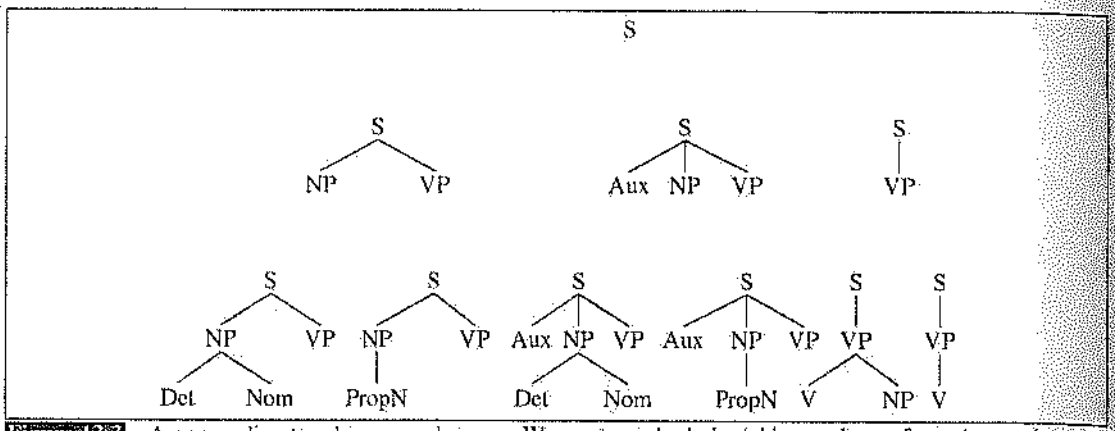


Figure 13.3 An expanding top-down search space. We create each ply by taking each tree from the previous ply, replacing the leftmost non-terminal with each of its possible expansions and collecting each of these trees into a new ply.

13.1.2 Bottom-Up Parsing

Bottom-up

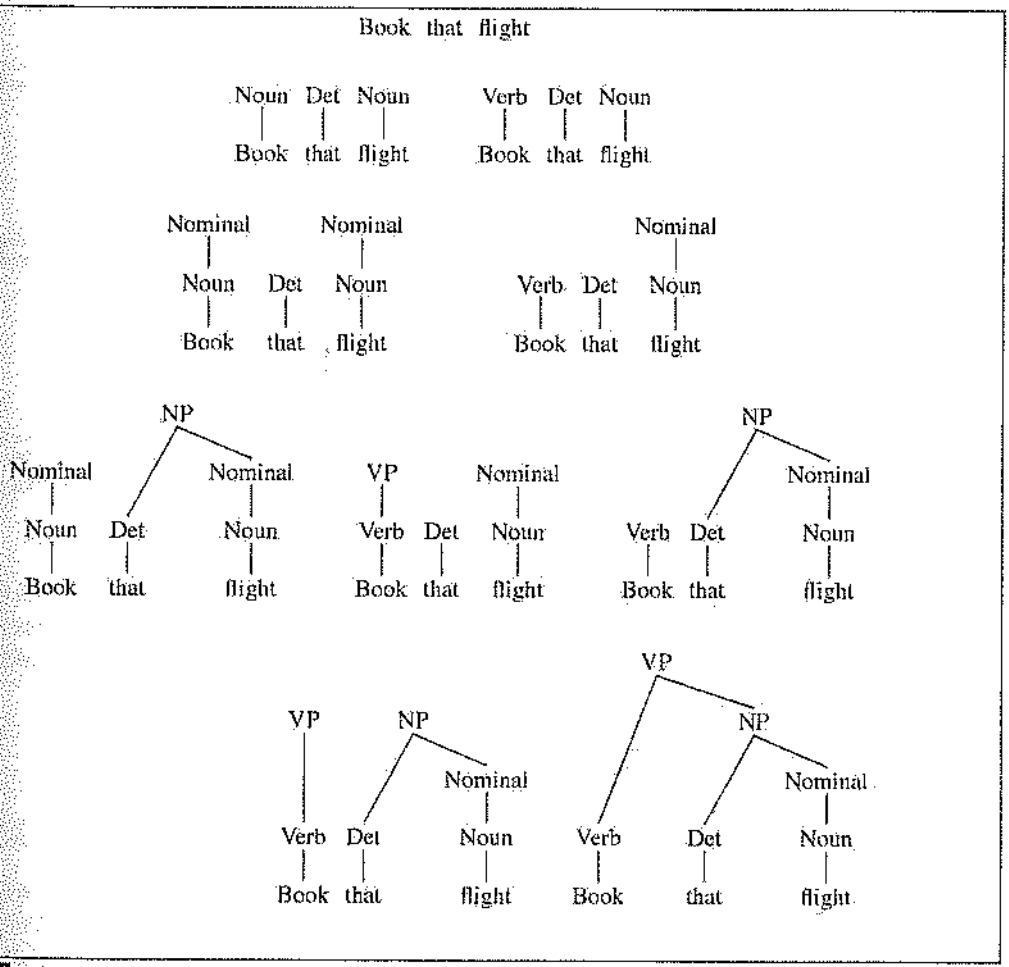
Bottom-up parsing is the earliest known parsing algorithm (it was first suggested by Yngve (1955)) and is used in the shift-reduce parsers common for computer languages (Aho and Ullman, 1972). In bottom-up parsing the parser starts with the words of the input, and tries to build trees from the words 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 13.4 shows the bottom-up search space, beginning with the sentence *Book that flight*. The parser begins by looking up each input word 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 Fig. 13.4 show this initial bifurcation of the search space.

Each of the trees in the second ply is 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 nouns (*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 non-terminal.

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 Fig. 13.4) contains the correct parse (see Fig. 13.2).



gure 13.4 An expanding bottom-up search space for the sentence *Book that flight*. This figure does not show the nal phase of the search with the correct parse tree (see Fig. 13.2). Make sure you understand how that final parse e follows from the search space in this figure.

13.1.3 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.

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 Fig. 13.3 all have left branches that cannot match the word *book*. None of these trees

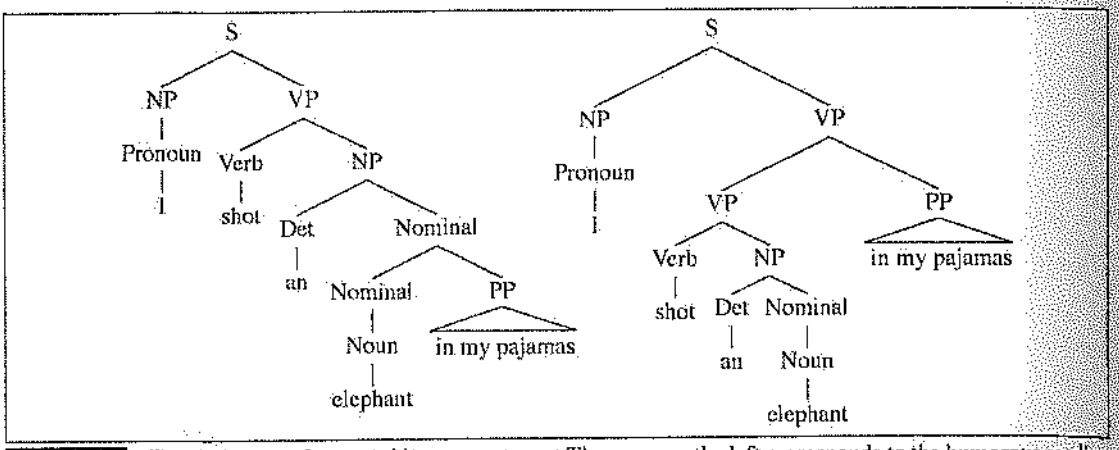


Figure 13.5 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

could possibly be used in parsing this sentence. This weakness in top-down parsers arises from the fact that they 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 input.

13.2 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 is perhaps the most serious problem faced by parsers. Chapter 5 introduced the notions of **part-of-speech ambiguity** and **part-of-speech disambiguation**. In this section we introduce a new kind of ambiguity, called **structural ambiguity**, which arises in the syntactic structures used in parsing. Structural ambiguity occurs when the grammar assigns more than one possible parse to a sentence. Groucho Marx's well-known line as Captain Spaulding is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or of the verb phrase headed by *shot*. Figure 13.5 illustrates these two analyses of Marx's line.

Structural ambiguity

Structural ambiguity, appropriately enough, comes in many forms. Two common kinds of ambiguity are **attachment ambiguity** and **coordination ambiguity**.

Attachment 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 instance, in the following example the gerundive-*VP flying to Paris* can be part of a gerundive sentence whose subject is *the Eiffel Tower* or it can be an adjunct modifying the *VP* headed by *saw*:

(13.2) We saw the Eiffel Tower flying to Paris.

Coordination ambiguity

In **coordination ambiguity** different sets of phrases can be conjoined by a conjunction like *and*. For example, the phrase *old men and women* can be bracketed as *[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.

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

(13.3) 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 affirmed [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*. A *PP* like *over nationwide television and radio* could be attached to any of the higher *VP*s or *NP*s (e.g., it could modify *people* or *night*).

The fact that there are many unreasonable parses for naturally occurring sentences is an extremely irksome problem that affects all parsers. Ultimately, most natural-language processing systems need to be able to choose the correct parse from the multitude of possible parses through a process known as **syntactic disambiguation**. Unfortunately, effective disambiguation algorithms generally require statistical, semantic, and pragmatic knowledge not readily available during syntactic processing (techniques for making use of such knowledge are introduced in Chapter 14 and Chapter 18).

Syntactic disambiguation

Lacking such knowledge, we are left with the choice of simply returning all the possible parse trees for a given input. Unfortunately, generating all the possible parses from robust, highly ambiguous, wide-coverage grammars such as the Penn Treebank grammar described in Chapter 12 is problematic. The reason for this lies in the potentially exponential number of parses that are possible for certain inputs. Consider the following ATIS example:

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

The recursive $VP \rightarrow VP\ PP$ and $Nominal \rightarrow Nominal\ PP$ rules conspire with the three prepositional phrases at the end of this sentence 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). Figure 13.6 illustrates a reasonable parse for this sentence. Church and Patil (1982) showed that the number of parses for sentences of this type grows exponentially at the same rate as the number of parenthesizations of arithmetic expressions.

Even if a sentence isn't ambiguous (i.e., it doesn't have more than one parse in the end), it can be inefficient to parse because of **local ambiguity**. Local ambiguity occurs

Local ambiguity

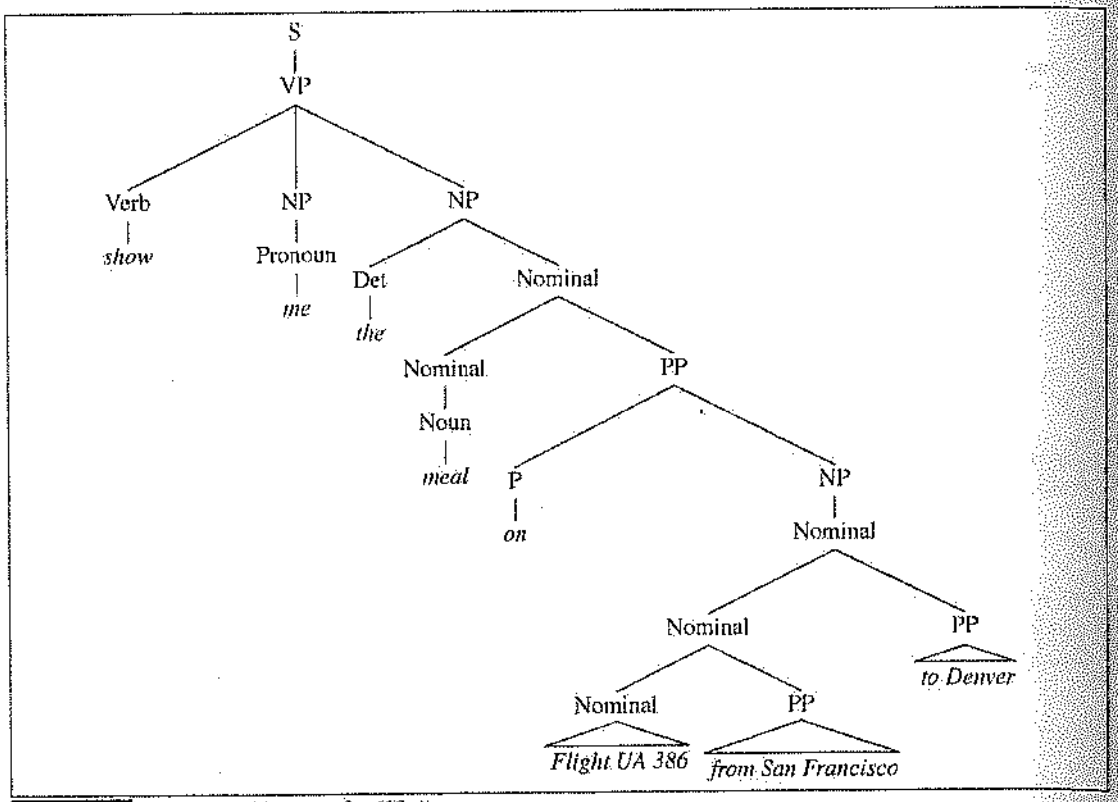


Figure 13.6 A reasonable parse for (13.4).

when some part of a sentence is ambiguous, that is, has more than one 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 the word is a verb or a noun until later. Thus, it must consider both possible parses.

13.3 Search in the Face of Ambiguity

To fully understand the problem that local and global ambiguity pose for syntactic parsing, let's return to our earlier description of top-down and bottom-up parsing. There we made the simplifying assumption that we could explore all possible parse trees in parallel. Thus, each ply of the search in Fig. 13.3 and Fig. 13.4 showed parallel expansions of the parse trees on the previous plies. Although it is certainly possible to implement this method directly, it typically requires an unrealistic amount of memory to store the space of trees as they are constructed. This is especially true since realistic grammars have much more ambiguity than the miniature grammar we've been using.

A common alternative approach to exploring complex search spaces is to use an agenda-based backtracking strategy such as those used to implement the various finite-

state machines in Chapters 2 and 3. A backtracking approach expands the search space incrementally by systematically exploring one state at a time. The state chosen for expansion can be based on simple systematic strategies, such as depth-first or breadth-first methods, or on more complex methods that make use of probabilistic and semantic considerations. When the given strategy arrives at a tree that is inconsistent with the input, the search continues by returning to an unexplored option already on the agenda. 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.

Unfortunately, the pervasive ambiguity in typical grammars leads to intolerable inefficiencies in any backtracking approach. Backtracking parsers will often build valid trees for portions of the input and then discard them during backtracking, only to find that they have to be rebuilt again. Consider the top-down backtracking process involved in finding a parse for the *NP* in (13.5):

(13.5) a flight from Indianapolis to Houston on NWA

The preferred complete parse is shown as the bottom tree in Fig. 13.7. While this phrase has numerous parses, we focus here on the amount of repeated work expended on the path to retrieving this single preferred parse.

A typical top-down, depth-first, left-to-right backtracking strategy leads to small parse trees that fail because they do not cover all of the input. These successive failures trigger backtracking events that lead to parses that incrementally cover more and more of the input. The sequence of trees attempted on the way to the correct parse by this top-down approach is shown in Fig. 13.7.

This figure clearly illustrates the kind of 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 work done on this simple example would, of course, be magnified by any ambiguity introduced at 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.

13.4 Dynamic Programming Parsing Methods

The previous section presented some of the ambiguity problems that afflict standard bottom-up or top-down parsers. Luckily, a single class of algorithms can solve these problems. **Dynamic programming** once again provides a framework for solving this problem, just as it helped us with the Minimum Edit Distance, Viterbi, and Forward algorithms. Recall that dynamic programming approaches systematically fill in tables of solutions to sub-problems. When complete, the tables contain the solution to all the sub-problems needed to solve the problem as a whole.

In the case of parsing, such tables store subtrees for each constituent in the input as it is discovered. The efficiency gain arises because these subtrees are discovered once, stored, and then used in all parses calling for that constituent. This solves the re-parsing problem (subtrees are looked up, not re-parsed) and partially solves the ambiguity problem (the dynamic programming table implicitly stores all possible parses

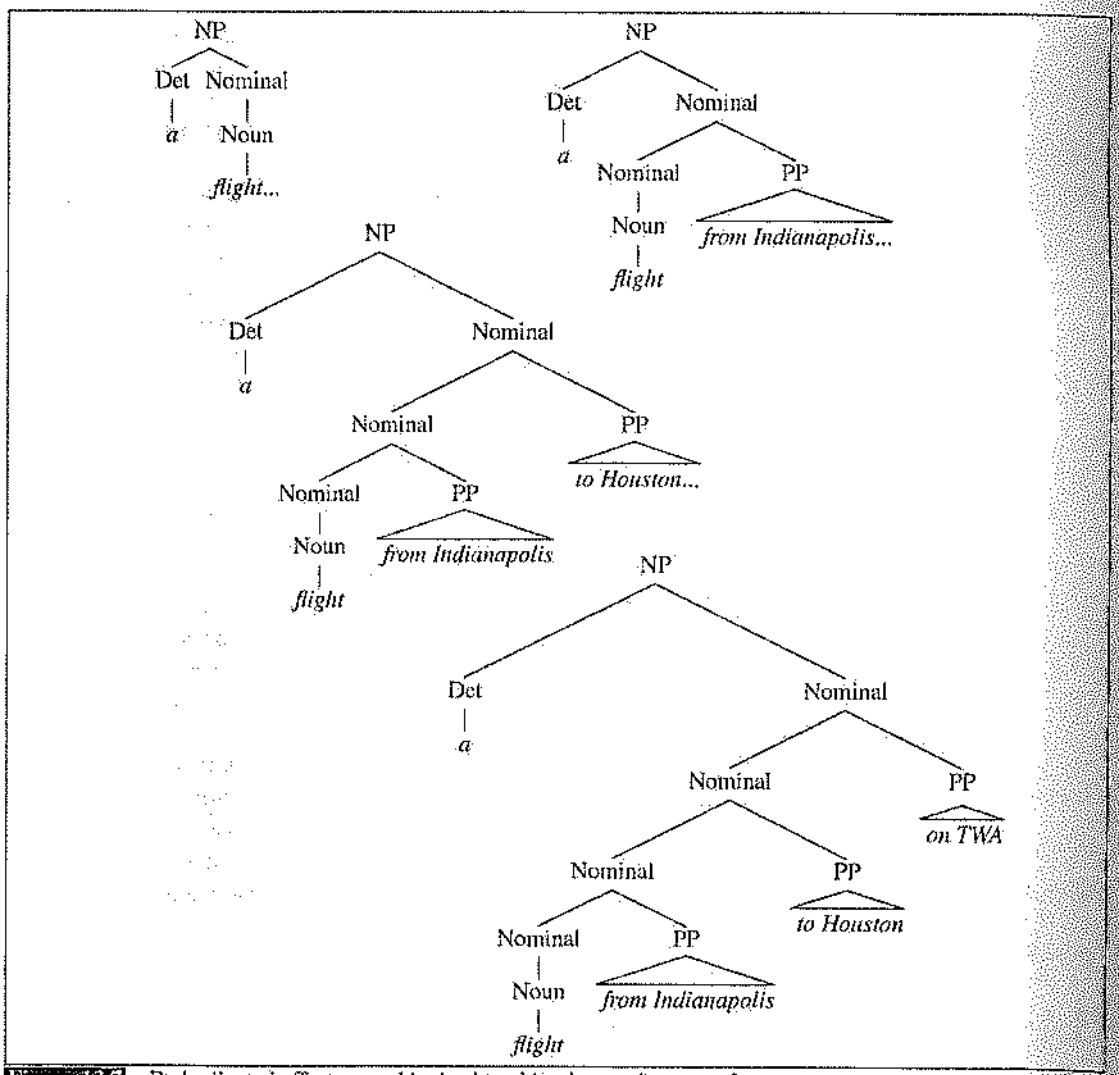


Figure 13.7 Reduplicated effort caused by backtracking in top-down parsing.

by storing all the constituents with links that enable the parses to be reconstructed). As we mentioned earlier, the three most widely used methods are the Cocke-Kasami-Younger (CKY) algorithm, the Earley algorithm, and chart parsing.

13.4.1 CKY Parsing

Let's begin our investigation of the CKY algorithm by examining one of its major requirements: the grammars used with it must be in Chomsky normal form (CNF). Recall from Chapter 12 that grammars in CNF are restricted to rules of the form $A \rightarrow B\ C$ or $A \rightarrow w$. That is, the right-hand side of each rule must expand either to two nonterminals or to a single terminal. Recall also that restricting a grammar to CNF does

not lead to any loss in expressiveness, since any context-free grammar can be converted into a corresponding CNF grammar that accepts exactly the same set of strings as the original grammar. This single restriction gives rise to an extremely simple and elegant table-based parsing method.

Conversion to Chomsky Normal Form

Let's start with the process of converting a generic CFG into one represented in CNF. Assuming we're dealing with an ϵ-free grammar, there are three situations we need to address in any generic grammar: rules that mix terminals with non-terminals on the right-hand side, rules that have a single non-terminal on the right, and rules in which the length of the right-hand side is greater than 2.

The remedy for rules that mix terminals and non-terminals is to simply introduce a new dummy non-terminal that covers only the original terminal. For example, a rule for an infinitive verb phrase such as *INF-VP* $\rightarrow$ *to VP* would be replaced by the two rules *INF-VP* $\rightarrow$ *TO VP* and *TO* $\rightarrow$ *to*.

Unit productions

Rules with a single non-terminal on the right are called **unit productions**. We can eliminate unit productions by rewriting the right-hand side of the original rules with the right-hand side of all the non-unit production rules that they ultimately lead to. More formally, if $A \stackrel{*}{\Rightarrow} B$ by a chain of one or more unit productions and $B \rightarrow \gamma$ is a non-unit production in our grammar, then we add $A \rightarrow \gamma$ for each such rule in the grammar and discard all the intervening unit productions. As we demonstrate with our toy grammar, this can lead to a substantial *flattening* of the grammar and a consequent promotion of terminals to fairly high levels in the resulting trees.

Rules with right-hand sides longer than 2 are normalized through the introduction of new non-terminals that spread the longer sequences over several new rules. Formally, if we have a rule like

$$A \rightarrow B\,C\,\gamma$$

we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production result in the following new rules:

$$X1 \rightarrow B\,C$$
$$A \rightarrow X1\,\gamma$$

In the case of longer right-hand sides, we simply iterate this process until the offending rule has been replaced by rules of length 2. The choice of replacing the leftmost pair of non-terminals is purely arbitrary; any systematic scheme that results in binary rules would suffice.

In our current grammar, the rule $S \rightarrow$ *Aux NP VP* would be replaced by the two rules $S \rightarrow$ *X1 VP* and *X1* $\rightarrow$ *Aux NP*.

The entire conversion process can be summarized as follows:

1. Copy all conforming rules to the new grammar unchanged.
2. Convert terminals within rules to dummy non-terminals.
3. Convert unit-productions.
4. Make all rules binary and add them to new grammar.

$\mathscr{L}_1$ Grammar	$\mathscr{L}_1$ in CNF
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 13.8 $\mathscr{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown here, all the original lexical entries from $\mathscr{L}_1$ carry over unchanged as well.

Figure 13.8 shows the results of applying this entire conversion procedure to the $\mathscr{L}_1$ grammar introduced earlier on page 428. Note that this figure doesn't show the original lexical rules; since these original lexical rules are already in CNF, they all carry over unchanged to the new grammar. Figure 13.8 does, however, show the various places where the process of eliminating unit productions has, in effect, created new lexical rules. For example, all the original verbs have been promoted to both *VP*s and to *S*s in the converted grammar.

CKY Recognition

With our grammar now in CNF, each non-terminal node above the part-of-speech level in a parse tree will have exactly two daughters. A simple two-dimensional matrix can be used to encode the structure of an entire tree. More specifically, for a sentence of length n, we are working with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. Each cell $[i, j]$ in this matrix contains a set of non-terminals that represent all the constituents that span positions i through j of the input. Since our indexing scheme begins with 0, it's natural to think of the indexes as pointing at the gaps between the input words (as in $_0$ *Book* $_1$ *that* $_2$ *flight* $_3$). It follows then that the cell that represents the entire input resides in position $[0, n]$ in the matrix.

Since our grammar is in CNF, the non-terminal entries in the table have exactly two daughters in the parse. Therefore, for each constituent represented by an entry $[i, j]$ in

the table, there must be a position in the input, k, where it can be split into two parts such that $i < k < j$. Given such a position k, the first constituent $[i, k]$ must lie to the left of entry $[i, j]$ somewhere along row i, and the second entry $[k, j]$ must lie beneath it, along column j.

To make this more concrete, consider the following example with its completed parse matrix, shown in Fig. 13.9.

(13.6) Book the flight through Houston.

The superdiagonal row in the matrix contains the parts of speech for each input word in the input. The subsequent diagonals above that superdiagonal contain constituents that cover all the spans of increasing length in the input.

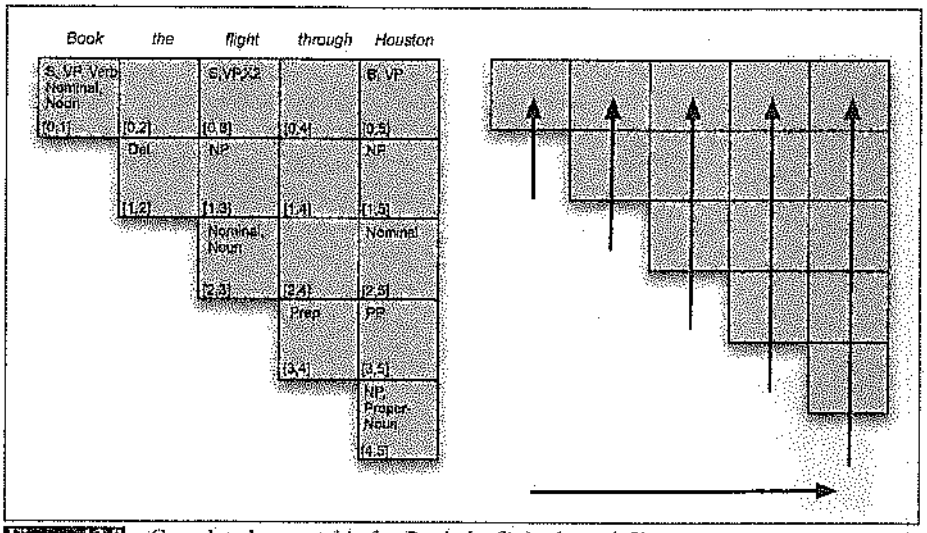


Figure 13.9 Completed parse table for *Book the flight through Houston*.

Given all this, CKY recognition is simply a matter of filling the parse table in the right way. To do this, we'll proceed in a bottom-up fashion so that at the point where we are filling any cell $[i, j]$, the cells containing the parts that could contribute to this entry (i.e., the cells to the left and the cells below) have already been filled. There are several ways to do this; as the right side of Fig. 13.9 illustrates, the algorithm given in Fig. 13.10 fills the upper-triangular matrix a column at a time working from left to right. Each column is then filled from bottom to top. This scheme guarantees that at each point in time we have all the information we need (to the left, since all the columns to the left have already been filled, and below since we're filling bottom to top). It also mirrors on-line parsing since filling the columns from left to right corresponds to processing each word one at a time.

The outermost loop of the algorithm given in Fig. 13.10 iterates over the columns, and the second loop iterates over the rows, from the bottom up. The purpose of the innermost loop is to range over all the places where a substring spanning i to j in the input might be split in two. As k ranges over the places where the string can be split, the pairs of cells we consider move, in lockstep, to the right along row i and down along

```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
    table[j − 1, j] ← {A | A → words[j] ∈ grammar}
    for i ← from j − 2 downto 0 do
      for k ← i + 1 to j − 1 do
        table[i,j] ← table[i,j] ∪
                     {A | A → BC ∈ grammar,
                          B ∈ table[i,k],
                          C ∈ table[k,j]}
```

Figure 13.10 The CKY algorithm.

column j. Figure 13.11 illustrates the general case of filling cell $[i, j]$. At each such split, the algorithm considers whether the contents of the two cells can be combined in a way that is sanctioned by a rule in the grammar. If such a rule exists, the non-terminal on its left-hand side is entered into the table.

Figure 13.12 shows how the five cells of column 5 of the table are filled after the word *Houston* is read. The arrows point out the two spans that are being used to add an entry to the table. Note that the action in cell $[0,5]$ indicates the presence of three alternative parses for this input, one where the *PP* modifies the *flight*, one where it modifies the booking, and one that captures the second argument in the original $VP \rightarrow Verb\ NP\ PP$ rule, now captured indirectly with the $VP \rightarrow X2\ PP$ rule.

In fact, since our current algorithm manipulates *sets* of non-terminals as cell entries, it won't include multiple copies of the same non-terminal in the table; the second *S* and *VP* discovered while processing $[0,5]$ would have no effect. We revisit this behavior in the next section.

CKY Parsing

The algorithm given in Fig. 13.10 is a recognizer, not a parser; for it to succeed, it simply has to find an S in cell $[0,N]$. To turn it into a parser capable of returning all possible parses for a given input, we can make two simple changes to the algorithm: the first change is to augment the entries in the table so that each non-terminal is paired with pointers to the table entries from which it was derived (more or less as shown in Fig. 13.12), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 13.12). With these changes, the completed table contains all the possible parses for a given input. Returning an arbitrary single parse consists of choosing an S from cell $[0,n]$ and then recursively retrieving its component constituents from the table.

Of course, returning all the parses for a given input may incur considerable cost. As we saw earlier, an exponential number of parses may be associated with a given input. In such cases, returning all the parses will have an unavoidable exponential cost. Looking forward to Chapter 14, we can also think about retrieving the best parse for a given input by further augmenting the table to contain the probabilities of each entry. Retrieving the most probable parse consists of running a suitably modified version of the Viterbi algorithm from Chapter 5 over the completed parse table.

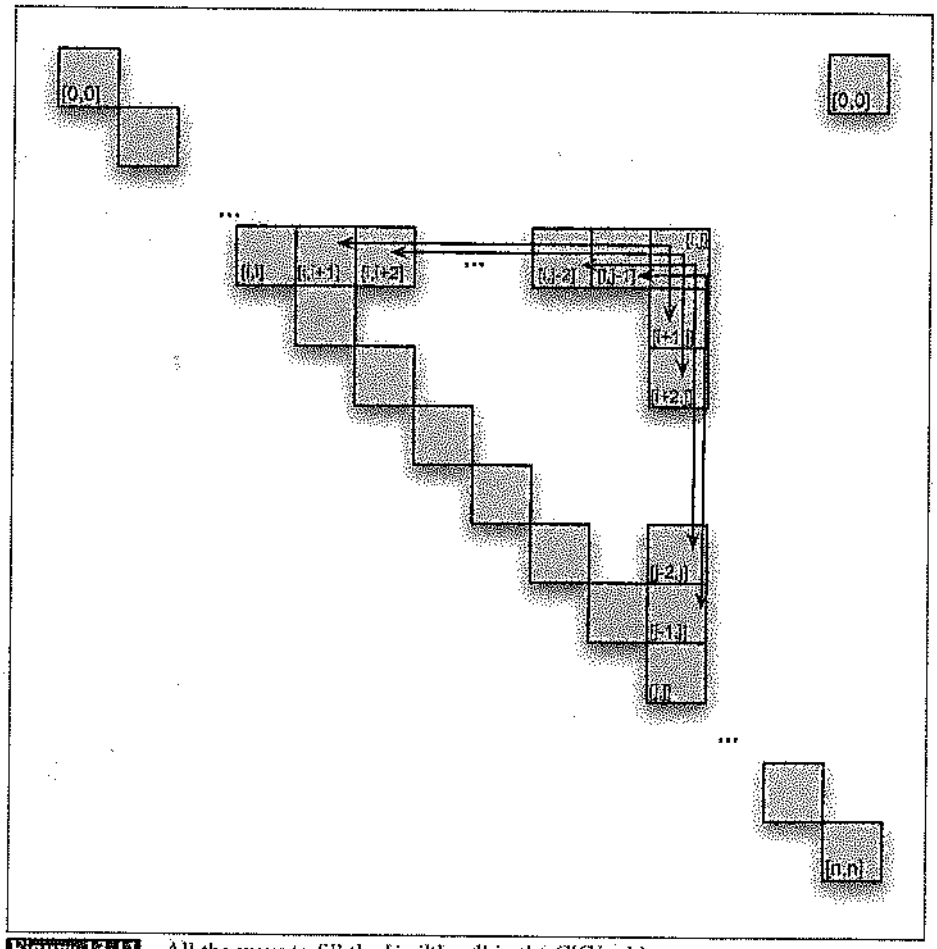


Figure 13.11 All the ways to fill the $[i, j]$th cell in the CKY table.

CKY in Practice

Finally, we should note that while the restriction to CNF does not pose a problem theoretically, it does pose some non-trivial problems in practice. Obviously, as things stand now, our parser isn't returning trees that are consistent with the grammar given to us by our friendly syntacticians. In addition to making our grammar developers unhappy, the conversion to CNF will complicate any syntax-driven approach to semantic analysis.

One approach to getting around these problems is to keep enough information around to transform our trees back to the original grammar as a post-processing step of the parse. This is trivial in the case of the transformation used for rules with length greater than 2. Simply deleting the new dummy non-terminals and promoting their daughters restores the original tree.

In the case of unit productions, it turns out to be more convenient to alter the basic CKY algorithm to handle them directly than it is to store the information needed to recover the correct trees. Exercise 13.3 asks you to make this change. Many of the

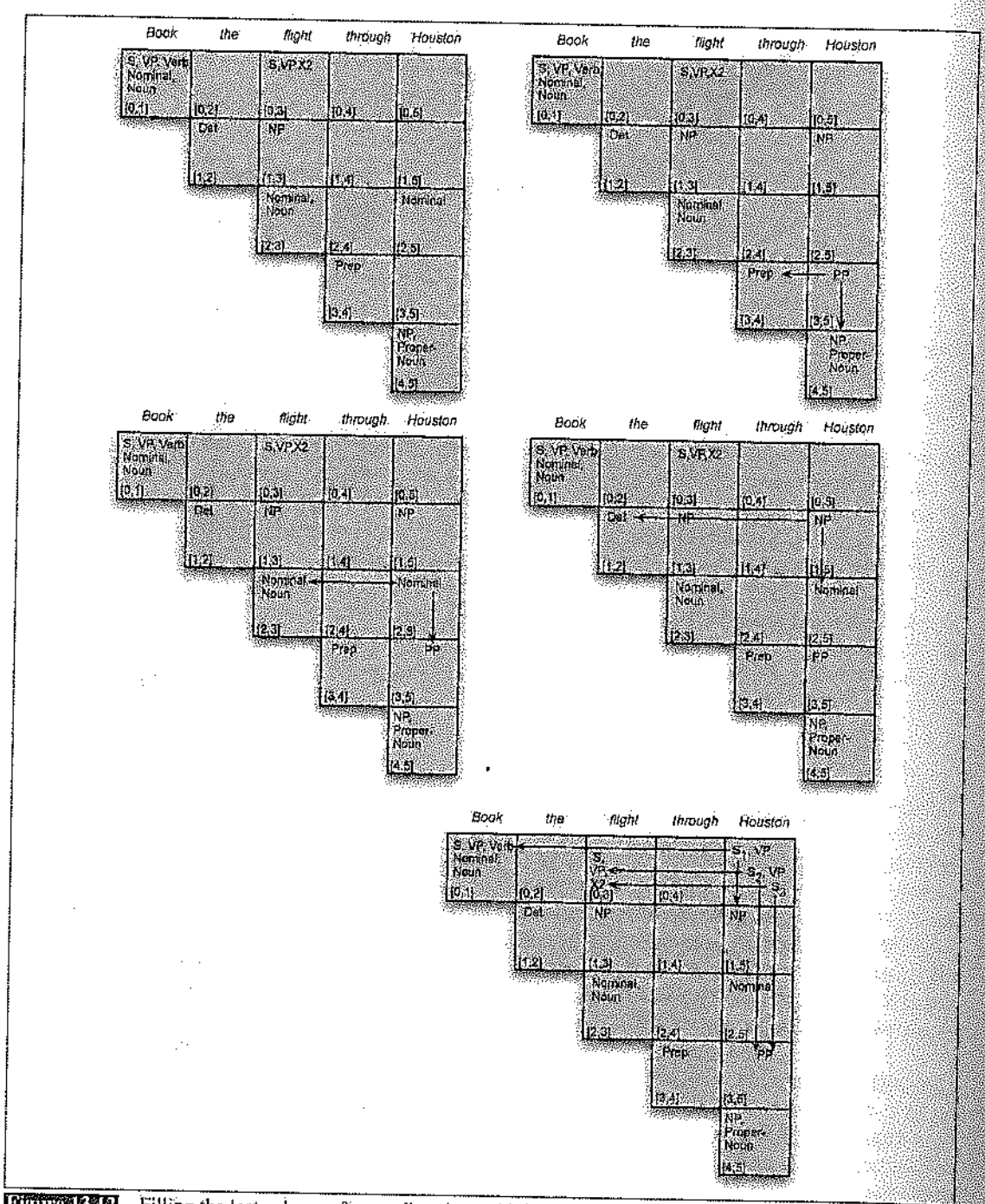


Figure 13.12 Filling the last column after reading the word *Houston*.

probabilistic parsers presented in Chapter 14 use the CKY algorithm altered in just this manner. Another solution is to adopt a more complex dynamic programming solution that simply accepts arbitrary CFGs. The next section presents such an approach.

13.4.2 The Earley Algorithm

In contrast with the bottom-up search implemented by the CKY algorithm, the Earley algorithm (Earley, 1970) uses dynamic programming to implement a **top-down search** of the kind discussed earlier in Section 13.1.1. The core of the Earley algorithm is a single left-to-right pass that fills an array—we'll call it a **chart**—that has $N+1$ entries. For each word position in the sentence, the chart contains a list of states representing the partial parse trees that have been generated so far. As with the CKY algorithm, the indexes represent the locations between the words in an input (as in ${}_0Book_1that_2flight_3$). By the end of the sentence, the chart compactly encodes all the possible parses of the input. Each possible subtree is represented only once and can thus be shared by all the parses that need it.

Chart

The individual states contained within each chart entry contain three kinds of information: a subtree corresponding to a single grammar rule, information about the progress made in completing this subtree, and the position of the subtree with respect to the input. We'll use a • within the right-hand side of a state's grammar rule to indicate the progress made in recognizing it. The resulting structure is called a **dotted rule**. A state's position with respect to the input is represented by two numbers indicating where the state begins and where its dot lies.

Dotted rule

Consider the following example states, which would be among those created by the Earley algorithm in the course of parsing (13.7):

(13.7) Book that flight.

$$\begin{aligned} S &\rightarrow \bullet\, VP, [0,0] \\ NP &\rightarrow Det \bullet Nominal, [1,2] \\ VP &\rightarrow V\, NP \bullet, [0,3] \end{aligned}$$

The first state, with its dot to the left of its constituent, represents a top-down prediction for this particular kind of *S*. The first 0 indicates that the constituent predicted by this state should begin at the start of the input; the second 0 reflects the fact that the dot lies at the beginning as well. The second state, created at a later stage in the processing of this sentence, indicates that an *NP* begins at position 1, that a *Det* has been successfully parsed, and that a *Nominal* is expected next. The third state, with its dot to the right of its two constituents, represents the successful discovery of a tree corresponding to a *VP* that spans the entire input.

The basic operation of an Earley parser is to march through the $N+1$ sets of states in the chart in a left-to-right fashion, processing the states within each set in order. At each step, one of the three operators described below is applied to each state, depending on its status. In each case, this results in the addition of new states to the end of either the current or the next set of states in the chart. The algorithm always moves forward through the chart, making additions as it goes; states are never removed and

the algorithm never backtracks to a previous chart entry once it has moved on. The presence of a state $S \rightarrow \alpha\bullet, [0,N]$ in the list of states in the last chart entry indicates a successful parse. Figure 13.13 gives the complete algorithm.

```
function EARLEY-PARSE(words, grammar) returns chart

  ENQUEUE((γ → • S, [0,0]), chart[0])
  for i ← from 0 to LENGTH(words) do
    for each state in chart[i] do
      if INCOMPLETE?(state) and
            NEXT-CAT(state) is not a part of speech then
        PREDICTOR(state)
      elseif INCOMPLETE?(state) and
            NEXT-CAT(state) is a part of speech then
        SCANNER(state)
      else
        COMPLETER(state)
    end
  end
  return(chart)

procedure PREDICTOR((A → α • B β, [i,j]))
  for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
      ENQUEUE((B → • γ, [j,j]), chart[j])
  end

procedure SCANNER((A → α • B β, [i,j]))
  if B ⊂ PARTS-OF-SPEECH(word[j]) then
    ENQUEUE((B → word[j], [j, j+1]), chart[j+1])

procedure COMPLETER((B → γ •, [j,k]))
  for each (A → α • B β, [i,j]) in chart[j] do
      ENQUEUE((A → α B • β, [i,k]), chart[k])
  end

procedure ENQUEUE(state, chart-entry)
  if state is not already in chart-entry then
      PUSH(state, chart-entry)
  end
```

Figure 13.13 The Earley algorithm.

The following three sections describe in detail the three operators used to process states in the chart. Each takes a single state as input and derives new states from it. These new states are then added to the chart as long as they are not already present. PREDICTOR and COMPLETER add states to the chart entry being processed, and SCANNER adds a state to the next chart entry.

PREDICTOR

As might be guessed from its name, the job of PREDICTOR is to create new states representing top-down expectations generated during the parsing process. PREDICTOR is applied to any state that has a non-terminal immediately to the right of its dot when the non-terminal is not a part-of-speech category. This application results in the creation of one new state for each alternative expansion of that non-terminal provided by the grammar. These new states are placed into the same chart entry as the generating state. They begin and end at the point in the input where the generating state ends.

For example, applying PREDICTOR to the state $S \rightarrow \bullet\ VP,\ [0,0]$ results in the addition of the following five states to the first chart entry.

$$\begin{aligned}
&VP \rightarrow \bullet\ Verb,\ [0,0]\\
&VP \rightarrow \bullet\ Verb\ NP,\ [0,0]\\
&VP \rightarrow \bullet\ Verb\ NP\ PP,\ [0,0]\\
&VP \rightarrow \bullet\ Verb\ PP,\ [0,0]\\
&VP \rightarrow \bullet\ VP\ PP,\ [0,0]
\end{aligned}$$

SCANNER

When a state has a part-of-speech category to the right of the dot, SCANNER is called to examine the input and incorporate into the chart a state corresponding to the prediction of a word with a particular part-of-speech. SCANNER accomplishes this by creating a new state from the input state with the dot advanced over the predicted input category. Note that unlike CKY, Earley uses top-down input to help deal with part-of-speech ambiguities; only those parts-of-speech of a word that are predicted by some existing state will find their way into the chart.

Returning to our example, when the state $VP \rightarrow \bullet\ Verb\ NP,\ [0,0]$ is processed, SCANNER consults the current word in the input since the category following the dot is a part-of-speech. It then notes that *book* can be a verb, matching the expectation in the current state. This results in the creation of the new state $Verb \rightarrow book\bullet,\ [0,1]$. This new state is then added to the chart entry that *follows* the one currently being processed. The noun sense of *book* never enters the chart since it is not predicted by any rule at this position in the input.

We should note that our version of SCANNER and PREDICTOR differs slightly from the corresponding operations in the original formulation of the algorithm (Earley, 1970). There terminals were treated uniformly as ordinary parts of the grammar by both PREDICTOR and SCANNER. In this approach, a state like $VP \rightarrow \bullet\ Verb\ NP,\ [0,0]$ would trigger predicted states corresponding to any rule that had *Verb* as its left-hand side. In our current example, the state $Verb \rightarrow \bullet\ book$ would be predicted. The original SCANNER would subsequently encounter this predicted state and match the current input token against the predicted token, resulting in a new state with the dot advanced, $Verb \rightarrow book\bullet$.

Unfortunately, this approach is not practical for applications with large lexicons, since states representing *every* word in a given word class would be entered into the chart as soon as that class was predicted. In our current example, states representing

every known verb would be added in addition to the one for *book*. For this reason, our version of the PREDICTOR does not create states representing predictions for individual lexical items. SCANNER makes up for this by explicitly inserting states representing completed lexical items despite the fact that no states in the chart predict them.

COMPLETER

COMPLETER is applied to a state when its dot has reached the right end of the rule. The presence of such a state represents the fact that the parser has successfully discovered a particular grammatical category over some span of the input. The purpose of COMPLETER is to find, and advance, all previously created states that were looking for this grammatical category at this position in the input. COMPLETER then creates states **copying** the older state, advancing the dot over the expected category, and installing the new state in the current chart entry.

In the current example, when the state $NP \rightarrow Det\ Nominal\bullet$, $[1,3]$ is processed, COMPLETER looks for incomplete states ending at position 1 and expecting an *NP*. It finds the states $VP \rightarrow Verb\bullet NP$, $[0,1]$ and $VP \rightarrow Verb\bullet NP\ PP$, $[0,1]$. This results in the addition of the new complete state, $VP \rightarrow Verb\ NP\bullet$, $[0,3]$, and the new incomplete state, $VP \rightarrow Verb\ NP\bullet PP$, $[0,3]$ to the chart.

A Complete Example

Figure 13.14 shows the sequence of states created during the complete processing of (13.7); each row indicates the state number for reference, the dotted rule, the start and end points, and finally the function that added this state to the chart. The algorithm begins by seeding the chart with a top-down expectation for an *S*, that is, by adding a dummy state $\gamma \rightarrow \bullet S, [0,0]$ to Chart[0]. When this state is processed, it is passed to PREDICTOR, leading to the creation of the three states representing predictions for each possible type of *S* and transitively passed to states for all of the left corners of those trees. When the state $VP \rightarrow \bullet\ Verb, [0,0]$ is reached, SCANNER is called and the first word is read. A state representing the verb sense of *Book* is added to the entry for Chart[1]. Note that when the subsequent sentence initial *VP* states are processed, SCANNER will be called again. However, new states are not added since they would be identical to the *Verb* state already in the chart.

When all the states of Chart[0] have been processed, the algorithm moves on to Chart[1], where it finds the state representing the verb sense of *book*. This is a complete state with its dot to the right of its constituent and is therefore passed to COMPLETER. COMPLETER then finds the four previously existing *VP* states expecting a *Verb* at this point in the input. These states are copied with their dots advanced and added to Chart[1]. The completed state corresponding to an intransitive *VP* then leads to the creation of an *S* representing an imperative sentence. Alternatively, the dot in the transitive verb phrase leads to the creation of the three states predicting different forms of *NP*s. The state $NP \rightarrow \bullet\ Det\ Nominal$, $[1,1]$ causes SCANNER to read the word *that* and add a corresponding state to Chart[2].

Moving on to Chart[2], the algorithm finds the state representing the determiner sense of *that*. This complete state leads to the advancement of the dot in the *NP* state

Chart[0]	S0	$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
	S1	$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
	S2	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
	S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
	S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
	S5	$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	Predictor
	S6	$NP \rightarrow \bullet Det\ Nominal$	[0,0]	Predictor
	S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
	S8	$VP \rightarrow \bullet Verb\ NP$	[0,0]	Predictor
	S9	$VP \rightarrow \bullet Verb\ NP\ PP$	[0,0]	Predictor
	S10	$VP \rightarrow \bullet Verb\ PP$	[0,0]	Predictor
	S11	$VP \rightarrow \bullet VP\ PP$	[0,0]	Predictor
Chart[1]	S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
	S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
	S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
	S15	$VP \rightarrow Verb \bullet NP\ PP$	[0,1]	Completer
	S16	$VP \rightarrow Verb \bullet PP$	[0,1]	Completer
	S17	$S \rightarrow VP \bullet$	[0,1]	Completer
	S18	$VP \rightarrow VP \bullet PP$	[0,1]	Completer
	S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
	S20	$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	Predictor
	S21	$NP \rightarrow \bullet Det\ Nominal$	[1,1]	Predictor
	S22	$PP \rightarrow \bullet Prep\ NP$	[1,1]	Predictor
Chart[2]	S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
	S24	$NP \rightarrow Det \bullet Nominal$	[1,2]	Completer
	S25	$Nominal \rightarrow \bullet Noun$	[2,2]	Predictor
	S26	$Nominal \rightarrow \bullet Nominal\ Noun$	[2,2]	Predictor
	S27	$Nominal \rightarrow \bullet Nominal\ PP$	[2,2]	Predictor
Chart[3]	S28	$Noun \rightarrow flight \bullet$	[2,3]	Scanner
	S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
	S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
	S31	$Nominal \rightarrow Nominal \bullet Noun$	[2,3]	Completer
	S32	$Nominal \rightarrow Nominal \bullet PP$	[2,3]	Completer
	S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
	S34	$VP \rightarrow Verb\ NP \bullet PP$	[0,3]	Completer
	S35	$PP \rightarrow \bullet Prep\ NP$	[3,3]	Predictor
	S36	$S \rightarrow VP \bullet$	[0,3]	Completer
	S37	$VP \rightarrow VP \bullet PP$	[0,3]	Completer

Figure 13.14 Chart entries created during an Earley parse of *Book that flight*. Each entry shows the state, its start and end points, and the function that placed it in the chart.

predicted in Chart[1] and also to the predictions for the various kinds of *Nominal*. The first of these causes SCANNER to be called for the last time to process the word *flight*.

Finally, moving on to Chart[3]: the presence of the state representing *flight* leads in succession to the completion of an *NP*, a transitive *VP*, and an *S*. The presence of the state $S \rightarrow VP\bullet$, [0,3] in the last chart entry signals the discovery of a successful parse.

It is useful to contrast this example with the CKY example given earlier. Although Earley managed to avoid adding an entry for the noun sense of *book*, its overall behavior

Chart[1]	S12	*Verb* → *book* •	[0,1]	Scanner
Chart[2]	S23	*Det* → *that* •	[1,2]	Scanner
Chart[3]	S28	*Noun* → *flight* •	[2,3]	Scanner
	S29	*Nominal* → *Noun* •	[2,3]	(S28)
	S30	*NP* → *Det Nominal* •	[1,3]	(S23, S29)
	S33	*VP* → *Verb NP* •	[0,3]	(S12, S30)
	S36	*S* → *VP* •	[0,3]	(S33)

Figure 13.15 States that participate in the final parse of *Book that flight*, including structural parse information.

is clearly much more promiscuous than that of CKY. This promiscuity arises from the purely top-down nature of the predictions that Earley makes. Exercise 13.6 asks you to improve the algorithm by eliminating some of these unnecessary predictions.

Retrieving Parse Trees from a Chart

As with the CKY algorithm, this version of the Earley algorithm is a recognizer, not a parser. Valid sentences will simply leave the state $S \rightarrow \alpha\bullet$, $[0,N]$ in the chart. To retrieve parses from the chart, we need to add an additional field to each state with information about the completed states that generated its constituents.

The information needed to fill these fields can be gathered by a simple change to the COMPLETER function. Recall that COMPLETER creates new states by advancing existing incomplete states when the constituent following the dot has been found. We just change COMPLETER to add a pointer to the older state onto a list of constituent-states for the new state. Retrieving a parse tree from the chart is then merely a matter of following pointers, starting with the state (or states) representing a complete S in the final chart entry. Figure 13.15 shows the chart entries produced by an appropriately updated COMPLETER that participated in the final parse for this example.

13.4.3 Chart Parsing

In both the CKY and Earley algorithms, the order in which events occur (adding entries to the table, reading words, making predictions, etc.) is statically determined by the procedures that make up these algorithms. Unfortunately, dynamically determining the order in which events occur based on the current information is often necessary for a variety of reasons. Fortunately, **chart parsing**, an approach advanced by Martin Kay and his colleagues (Kaplan, 1973; Kay, 1982), permits a more flexible determination of the order in which chart entries are processed. This is accomplished through the use of an explicit *agenda*. In this scheme, as states (called **edges** in this approach) are created, they are added to an agenda that is kept ordered according to a policy that is specified *separately* from the main parsing algorithm. This can be viewed as another instance of a state-space search that we've seen several times before. The FSA and FST recognition and parsing algorithms in Chapters 2 and 3 employed agendas with simple static policies, and the A* decoding algorithm described in Chapter 9 is driven by an agenda that is ordered probabilistically.

Chart parsing

Figure 13.16 presents a generic version of a parser based on such a scheme. The main part of the algorithm consists of a single loop that removes an edge from the front of an agenda, processes it, and then moves on to the next entry in the agenda. When the agenda is empty, the parser stops and returns the chart. The policy used to order the elements in the agenda thus determines the order in which further edges are created and predictions are made.

```
function CHART-PARSE(words, grammar, agenda-strategy) returns chart
  INITIALIZE(chart, agenda, words)
  while agenda
      current-edge ← POP(agenda)
      PROCESS-EDGE(current-edge)
  return(chart)

procedure PROCESS-EDGE(edge)
    ADD-TO-CHART(edge)
    if INCOMPLETE?(edge)
        FORWARD-FUNDAMENTAL-RULE(edge)
    else
        BACKWARD-FUNDAMENTAL-RULE(edge)
    MAKE-PREDICTIONS(edge)

procedure FORWARD-FUNDAMENTAL((A → α • B β, [i, j]))
  for each(B → γ •, [j,k]) in chart
      ADD-TO-AGENDA(A → α B • β, [i,k])

procedure BACKWARD-FUNDAMENTAL((B → γ •, [j,k]))
  for each(A → α • B β, [i, j]) in chart
      ADD-TO-AGENDA(A → α B • β, [i,k])

procedure ADD-TO-CHART(edge)
  if edge is not already in chart then
      Add edge to chart

procedure ADD-TO-AGENDA(edge)
  if edge is not already in agenda then
      APPLY(agenda-strategy, edge, agenda)
```

Figure 13.16 A chart parsing algorithm.

Fundamental rule

The key principle in processing edges in this approach is what Kay termed the **fundamental rule** of chart parsing. The fundamental rule states that when the chart contains two contiguous edges where one of the edges provides the constituent that the other one needs, a new edge should be created that spans the original edges and incorporates the provided material. More formally, the fundamental rule states the following: if the chart contains two edges $A \rightarrow \alpha \bullet B\,\beta, [i, j]$ and $B \rightarrow \gamma \bullet, [j,k]$, then we should add the new edge $A \rightarrow \alpha\, B \bullet \beta\, [i,k]$ to the chart. It should be clear that the fundamental rule is a generalization of the basic table-filling operations found in both the CKY and Earley algorithms.

The fundamental rule is triggered in Fig. 13.16 when an edge is removed from the agenda and passed to the PROCESS-EDGE procedure. Note that the fundamental rule

itself does not specify which of the two edges involved has triggered the processing. PROCESS-EDGE handles both cases by checking to see whether or not the edge in question is complete. If it is complete, then the algorithm looks earlier in the chart to see if any existing edge can be advanced; if it is incomplete, then the algorithm looks later in the chart to see if it can be advanced by any pre-existing edge later in the chart.

The next piece of the algorithm to specify is the method for making predictions based on the edge being processed. There are two key components to making predictions in chart parsing: the events that trigger predictions and the nature of the prediction. These components vary depending on whether we are pursuing a top-down or bottom-up strategy. As in Earley, top-down predictions are triggered by expectations that arise from incomplete edges that have been entered into the chart; bottom-up predictions are triggered by the discovery of completed constituents. Figure 13.17 illustrates how these two strategies can be integrated into the chart parsing algorithm.

```
procedure MAKE-PREDICTIONS(edge)
  if Top-Down and INCOMPLETE?(edge)
    TD-PREDICT(edge)
  elseif Bottom-Up and COMPLETE?(edge)
    BU-PREDICT(edge)

procedure TD-PREDICT((A → α • B β, [i, j]))
  for each(B → γ) in grammar do
    ADD-TO-AGENDA(B → • γ, [j, j])

procedure BU-PREDICT((B → γ •, [i, j]))
  for each(A → B β) in grammar
    ADD-TO-AGENDA(A → B • β, [i, j])
```

Figure 13.17 More of the chart parsing algorithm.

Obviously, we've left out many of the bookkeeping details that would have to be specified to turn this approach into a real parser. Among the details that have to be worked out are how the INITIALIZE procedure gets things started, how and when words are read, how the chart is organized, and how to specify an agenda strategy. Indeed, in describing this approach, Kay (1982) refers to it as an **algorithm schema** rather than an algorithm, since it more accurately specifies an entire family of parsers rather than any particular parser. Exercise 13.7 asks you to explore some of the available choices by implementing various chart parsers.

Algorithm schema

13.5 Partial Parsing

Many language processing tasks do not require complex, complete parse trees for all inputs. For these tasks, a **partial parse**, or **shallow parse**, of input sentences may be sufficient. For example, information extraction systems generally do not extract *all* the possible information from a text: they simply identify and classify the segments in

Partial parse
Shallow parse

a text that are likely to contain valuable information. Similarly, information retrieval systems may index texts according to a subset of the constituents found in them.

There are many different approaches to partial parsing. Some make use of cascades of FSTs, of the kind discussed in Chapter 3, to produce tree-like representations. These approaches typically produce flatter trees than the ones we've been discussing in this chapter and the previous one. This flatness arises from the fact that FST cascade approaches generally defer decisions that may require semantic or contextual factors, such as prepositional phrase attachments, coordination ambiguities, and nominal compound analyses. Nevertheless, the intent is to produce parse trees that link all the major constituents in an input.

Chunking

An alternative style of partial parsing is known as **chunking**. Chunking is the process of identifying and classifying the flat, non-overlapping segments of a sentence that constitute the basic non-recursive phrases corresponding to the major parts-of-speech found in most wide-coverage grammars. This set typically includes noun phrases, verb phrases, adjective phrases, and prepositional phrases; in other words, the phrases that correspond to the content-bearing parts-of-speech. Of course, not all applications require the identification of all of these categories; indeed, the most common chunking task is to simply find all the base noun phrases in a text.

Since chunked texts lack a hierarchical structure, a simple bracketing notation is sufficient to denote the location and the type of the chunks in a given example. The following example illustrates a typical bracketed notation.

(13.8) [$_{NP}$ The morning flight] [$_{PP}$ from] [$_{NP}$ Denver] [$_{VP}$ has arrived.]

This bracketing notation makes clear the two fundamental tasks that are involved in chunking: finding the non-overlapping extents of the chunks and assigning the correct label to the discovered chunks.

Note that in this example all the words are contained in some chunk. This will not be the case in all chunking applications. Many words in any input will often fall outside of any chunk, for example, in systems searching for base *NP*s in their inputs, as in the following:

(13.9) [$_{NP}$ The morning flight] from [$_{NP}$ Denver] has arrived.

The details of what constitutes a syntactic base phrase for any given system varies according to the syntactic theories underlying the system and whether the phrases are being derived from a treebank. Nevertheless, some standard guidelines are followed in most systems. First and foremost, base phrases of a given type do not recursively contain any constituents of the same type. Eliminating this kind of recursion leaves us with the problem of determining the boundaries of the non-recursive phrases. In most approaches, base phrases include the headword of the phrase, along with any pre-head material within the constituent, while crucially excluding any post-head material. Eliminating post-head modifiers from the major categories automatically removes the need to resolve attachment ambiguities. Note that this exclusion does lead to certain oddities, such as *PP*s and *VP*s often consisting solely of their heads. Thus, our earlier example *a flight from Indianapolis to Houston on NWA* is reduced to the following:

(13.10) [$_{NP}$ a flight] [$_{PP}$ from] [$_{NP}$ Indianapolis][$_{PP}$ to][$_{NP}$ Houston][$_{PP}$ on][$_{NP}$ NWA]

13.5.1 Finite-State Rule-Based Chunking

Syntactic base phrases of the kind we're considering can be characterized by finite-state automata (or finite-state rules, or regular expressions) of the kind discussed in Chapters 2 and 3. In finite-state rule-based chunking, rules are hand-crafted to capture the phrases of interest for any particular application. In most rule-based systems, chunking proceeds from left to right, finding the longest matching chunk from the beginning of the sentence and continuing with the first word after the end of the previously recognized chunk. The process continues until the end of the sentence. This is a greedy process and is not guaranteed to find the best global analysis for any given input.

The primary limitation placed on these chunk rules is that they cannot contain any recursion; the right-hand side of the rule cannot reference directly or indirectly the category that the rule is designed to capture. In other words, rules of the form $NP \rightarrow Det\ Nominal$ are fine, but rules such as $Nominal \rightarrow Nominal\ PP$ are not. Consider the following example chunk rules adapted from Abney (1996).

$$\begin{aligned} NP &\rightarrow (DT)\ NN^*\ NN \\ NP &\rightarrow NNP \\ VP &\rightarrow VB \\ VP &\rightarrow Aux\ VB \end{aligned}$$

The process of turning these rules into a single finite-state transducer is the same process we introduced in Chapter 3 to capture spelling and phonological rules for English. Finite-state transducers are created corresponding to each rule and are then unioned together to form a single machine that can be determinized and minimized.

As we saw in Chapter 3, a major benefit of the finite-state approach is the ability to use the output of earlier transducers as inputs to subsequent transducers to form **cascades**. In **partial parsing**, this technique can be used to more closely approximate the output of true context-free parsers. In this approach, an initial set of transducers is used, in the way just described, to find a subset of syntactic base phrases. These base phrases are then passed as input to further transducers that detect larger and larger constituents such as prepositional phrases, verb phrases, clauses, and sentences. Consider the following rules, again adapted from Abney (1996).

$$\begin{aligned} FST_2 \quad PP &\rightarrow IN\ NP \\ FST_3 \quad S &\rightarrow PP^*\ NP\ PP^*\ VP\ PP^* \end{aligned}$$

Combining these two machines with the earlier ruleset results in a three-machine cascade. The application of this cascade to (13.8) is shown in Fig. 13.18.

13.5.2 Machine Learning-Based Approaches to Chunking

As with part-of-speech tagging, an alternative to rule-based processing is to use supervised machine learning techniques to *train* a chunker by using annotated data as a training set. As described earlier in Chapter 6, we can view the task as one of **sequential classification**, where a classifier is trained to label each element of the input

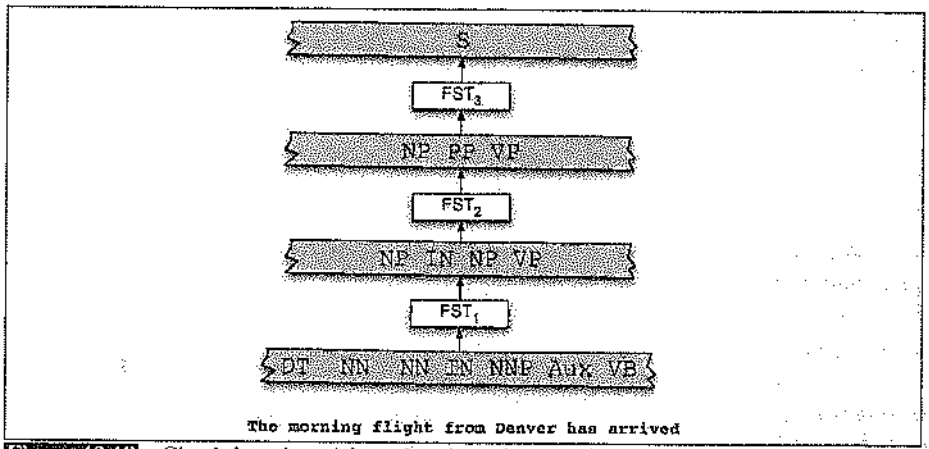


Figure 13.18 Chunk-based partial parsing through a set of finite-state cascades. FST_1 transduces from part-of-speech tags to base noun phrases and verb phrases. FST_2 finds prepositional phrases. Finally, FST_3 detects sentences.

in sequence. Any of the standard approaches to training classifiers apply to this problem. In the work that pioneered this approach, Ramshaw and Marcus (1995) used the transformation-based learning method described in Chapter 5.

The critical first step in such an approach is to be able to view the chunking process in a way that is amenable to sequential classification. A particularly fruitful approach is to treat chunking as a tagging task similar to part-of-speech tagging (Ramshaw and Marcus, 1995). In this approach, a small tagset simultaneously encodes both the segmentation and the labeling of the chunks in the input. The standard way to do this has come to be called **IOB tagging** and is accomplished by introducing tags to represent the beginning (B) and internal (I) parts of each chunk, as well as those elements of the input that are outside (O) any chunk. Under this scheme, the size of the tagset is $(2n+1)$, where n is the number of categories to be classified. The following example shows the bracketing notation of (13.8) on page 451 reframed as a tagging task:

IOB tagging

(13.11) *The morning flight from Denver has arrived*
B_NP I_NP I_NP B_PP B_NP B_VP I_VP

The same sentence with only the base-NPs tagged illustrates the role of the O tags.

(13.12) *The morning flight from Denver has arrived.*
B_NP I_NP I_NP O B_NP O O

Notice that there is no explicit encoding of the end of a chunk in this scheme; the end of any chunk is implicit in any transition from an I or B to a B or O tag. This encoding reflects the notion that when sequentially labeling words, it is generally easier (at least in English) to detect the beginning of a new chunk than it is to know when a chunk has ended. Not surprisingly, a variety of other tagging schemes represent chunks in subtly different ways, including some that explicitly mark the end of constituents. Tjong Kim Sang and Veenstra (1999) describe three variations on this basic tagging scheme and investigate their performance on a variety of chunking tasks.

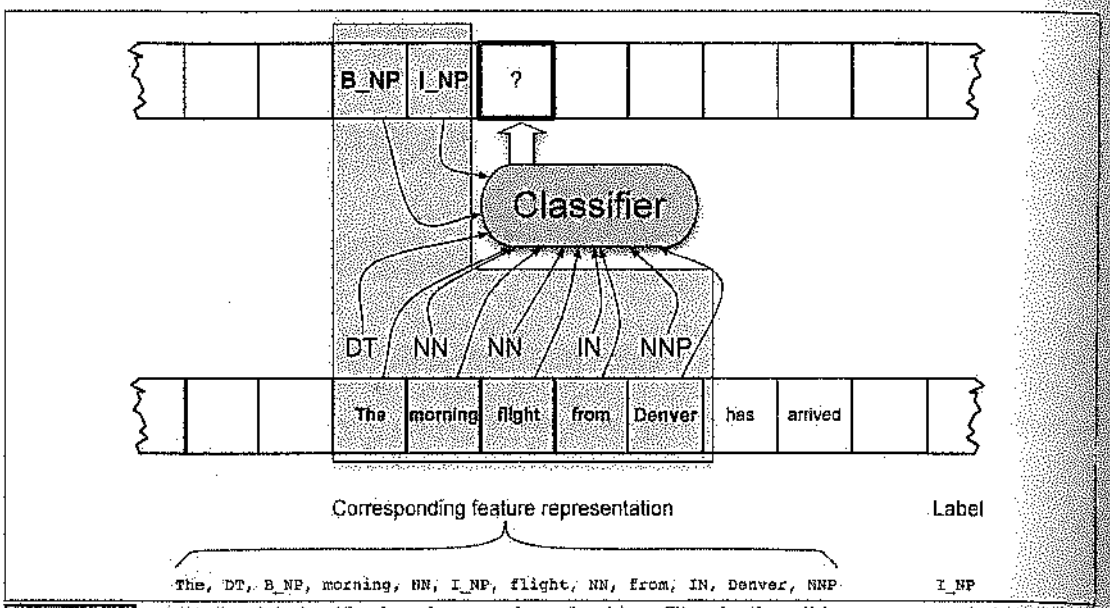


Figure 13.19 A sequential-classifier-based approach to chunking. The chunker slides a context window over the sentence, classifying words as it proceeds. At this point, the classifier is attempting to label *flight*. Features derived from the context typically include the words, part-of-speech tags as well as the previously assigned chunk tags.

Given such a tagging scheme, building a chunker consists of training a classifier to label each word of an input sentence with one of the IOB tags from the tagset. Of course, training requires training data consisting of the phrases of interest delimited and marked with the appropriate category. The direct approach is to annotate a representative corpus. Unfortunately, annotation efforts can be both expensive and time consuming. It turns out that the best place to find such data for chunking is in an existing treebank such as the Penn Treebank described in Chapter 12.

Such treebanks provide a complete parse for each corpus sentence, allowing base syntactic phrases to be extracted from the parse constituents. To find the phrases we're interested in, we just need to know the appropriate non-terminal names in the corpus. Finding chunk boundaries requires finding the head and then including the material to the left of the head, ignoring the text to the right. This is somewhat error-prone since it relies on the accuracy of the head-finding rules described in Chapter 12.

Having extracted a training corpus from a treebank, we must now cast the training data into a form that's useful for training classifiers. In this case, each input can be represented as a set of features extracted from a context window that surrounds the word to be classified. Using a window that extends two words before and two words after the word being classified seems to provide reasonable performance. Features extracted from this window include the words themselves, their parts-of-speech, and the chunk tags of the preceding inputs in the window.

Figure 13.19 illustrates this scheme with the example given earlier. During training, the classifier would be provided with a training vector consisting of the values of 13 features; the two words to the left of the decision point, their parts-of-speech and chunk

tags, the word to be tagged along with its part-of-speech, the two words that follow along with their parts-of speech, and finally the correct chunk tag, in this case, I_NP. During classification, the classifier is given the same vector without the answer and assigns the most appropriate tag from its tagset.

13.5.3 Chunking-System Evaluations

As with the evaluation of part-of-speech taggers, the evaluation of chunkers proceeds by comparing chunker output with gold-standard answers provided by human annotators. However, unlike part-of-speech tagging, word-by-word accuracy measures are not appropriate. Instead, chunkers are evaluated according to the notions of precision, recall, and the F-measure borrowed from the field of information retrieval.

Precision

Precision measures the percentage of system-provided chunks that were correct. Correct here means that both the boundaries of the chunk and the chunk's label are correct. Precision is therefore defined as

$$\textbf{Precision:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of chunks given by system}}$$

Recall

Recall measures the percentage of chunks actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall:} = \frac{\text{Number of correct chunks given by system}}{\text{Total number of actual chunks in the text}}$$

F-measure

The **F-measure** (van Rijsbergen, 1975) provides a way to combine these two measures into a single metric. The F-measure is defined as

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is sometimes called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \tag{13.13}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1}\frac{1}{a_2}\frac{1}{a_3}\cdots\frac{1}{a_n}} \tag{13.14}$$

and hence F-measure is

$$F = \frac{1}{\frac{1}{\alpha P} \times \frac{1}{(1-\alpha)R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{13.15}$$

The best current systems achieve an F-measure of around .96 on the task of base-NP chunking. Learning-based systems designed to find a more complete set of base

phrases, such as the ones given in Fig. 13.20, achieve *F*-measures in the .92 to .94 range. The choice of learning approach seems to have little impact; a wide range of machine learning approaches achieve similar results (Cardie et al., 2000). FST-based systems (Section 13.5.1) achieve *F*-measures ranging from .85 to .92 on this task.

Statistical significance on chunking results can be computed using matched-pair tests such as McNemar's test, or variants such as the Matched-Pair Sentence Segment Word Error (MAPSSWE) test described on page 329.

Factors limiting the performance of current systems include part-of-speech tagging accuracy, inconsistencies in the training data introduced by the process of extracting chunks from parse trees, and difficulty resolving ambiguities involving conjunctions. Consider the following examples that involve pre-nominal modifiers and conjunctions.

(13.16) [$_{NP}$ Late arrivals and departures] are commonplace during winter.

(13.17) [$_{NP}$ Late arrivals] and [$_{NP}$ cancellations] are commonplace during winter.

In the first example, *late* is shared by both *arrivals* and *departures*, yielding a single long base-NP. In the second example, *late* is not shared and modifies *arrivals* alone, thus yielding two base-NPs. Distinguishing these two situations, and others like them, requires access to semantic and context information unavailable to current chunkers.

Label	Category	Proportion (%)	Example
NP	Noun Phrase	51	*The most frequently cancelled flight*
VP	Verb Phrase	20	*may not arrive*
PP	Prepositional Phrase	20	*to Houston*
ADVP	Adverbial Phrase	4	*earlier*
SBAR	Subordinate Clause	2	*that*
ADJP	Adjective Phrase	2	*late*

Figure 13.20 Most frequent base phrases used in the 2000 CONLL shared task. These chunks correspond to the major categories contained in the Penn Treebank.

13.6 Summary

The two major ideas introduced in this chapter are those of **parsing** and **partial parsing**. Here's a summary of the main points we covered about these ideas:

- Parsing can be viewed as a **search** problem.
- Two common architectural metaphors for this search are **top-down** (starting with the root *S* and growing trees down to the input words) and **bottom-up** (starting with the words and growing trees up toward the root *S*).
- **Ambiguity** combined with the **repeated parsing of subtrees** poses problems for simple backtracking algorithms.
- A sentence is **structurally ambiguous** if the grammar assigns it more than one possible parse. Common kinds of structural ambiguity include **PP-attachment**, **coordination ambiguity**, and **noun-phrase bracketing ambiguity**.

- **Dynamic programming** parsing algorithms use a table of partial parses to efficiently parse ambiguous sentences. The **CKY**, **Earley**, and **chart parsing** algorithms all use dynamic programming to solve the repeated parsing of subtrees problem.
- The CKY algorithm restricts the form of the grammar to Chomsky normal form (CNF); the Earley and chart parsers accept unrestricted context-free grammars.
- Many practical problems, including **information extraction** problems, can be solved without full parsing.
- **Partial parsing** and **chunking** are methods for identifying shallow syntactic constituents in a text.
- High-accuracy partial parsing can be achieved either through rule-based or machine learning-based methods.

Bibliographical and Historical Notes

Writing about the history of compilers, Knuth notes:

> In this field there has been an unusual amount of parallel discovery of the same technique by people working independently.

Well, perhaps not unusual, if multiple discovery is the norm (see page 13). But there has certainly been enough parallel publication that this history errs on the side of succinctness in giving only a characteristic early mention of each algorithm; the interested reader should see Aho and Ullman (1972).

Bottom-up parsing seems to have been first described by Yngve (1955), who gave a breadth-first, bottom-up parsing algorithm as part of an illustration of a machine translation procedure. Top-down approaches to parsing and translation were described (presumably independently) by at least Glennie (1960), Irons (1961), and Kuno and Oettinger (1963). Dynamic programming parsing, once again, has a history of independent discovery. According to Martin Kay (personal communication), a dynamic programming parser containing the roots of the CKY algorithm was first implemented by John Cocke in 1960. Later work extended and formalized the algorithm, as well as proving its time complexity (Kay, 1967; Younger, 1967; Kasami, 1965). The related

WFST

well-formed substring table (**WFST**) seems to have been independently proposed by Kuno (1965) as a data structure that stores the results of all previous computations in the course of the parse. Based on a generalization of Cocke's work, a similar data structure had been independently described by Kay (1967, 1973). The top-down application of dynamic programming to parsing was described in Earley's Ph.D. dissertation (Earley, 1968, 1970). Sheil (1976) showed the equivalence of the WFST and the Earley algorithm. Norvig (1991) shows that the efficiency offered by dynamic programming can be captured in any language with a *memoization* function (such as in LISP) simply by wrapping the *memoization* operation around a simple top-down parser.

While parsing via cascades of finite-state automata had been common in the early history of parsing (Harris, 1962), the focus shifted to full CFG parsing quite soon af-

terward. Church (1980) argued for a return to finite-state grammars as a processing model for natural language understanding; other early finite-state parsing models include Ejerhed (1988). Abney (1991) argued for the important practical role of shallow parsing. Much recent work on shallow parsing applies machine learning to the task of learning the patterns; see, for example, Ramshaw and Marcus (1995), Argamon et al. (1998), Munoz et al. (1999).

The classic reference for parsing algorithms is Aho and Ullman (1972); although the focus of that book is on computer languages, most of the algorithms have been applied to natural language. A good programming languages textbook such as Aho et al. (1986) is also useful.

Exercises

13.1 Implement the algorithm to convert arbitrary context-free grammars to CNF. Apply your program to the $\mathscr{L}_1$ grammar.

13.2 Implement the CKY algorithm and test it with your converted $\mathscr{L}_1$ grammar.

13.3 Rewrite the CKY algorithm given in Fig. 13.10 on page 440 so that it can accept grammars that contain unit productions.

13.4 Augment the Earley algorithm of Fig. 13.13 to enable parse trees to be retrieved from the chart by modifying the pseudocode for COMPLETER as described on page 448.

13.5 Implement the Earley algorithm as augmented in the previous exercise. Check it on a test sentence by using the $\mathscr{L}_1$ grammar.

13.6 Alter the Earley algorithm so that it makes better use of bottom-up information to reduce the number of useless predictions.

13.7 Attempt to recast the CKY and Earley algorithms in the chart-parsing paradigm.

13.8 Discuss the relative advantages and disadvantages of partial versus full parsing.

13.9 Implement a more extensive finite-state grammar for noun groups by using the examples given in Section 13.5 and test it on some *NP*s. Use an on-line dictionary with parts-of-speech if available; if not, build a more restricted system by hand.

13.10 Discuss how to augment a parser to deal with input that may be incorrect, for example, containing spelling errors or mistakes arising from automatic speech recognition.