CHAPTER

13 Constituency Parsing

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

Syntactic parsing is the task of assigning a syntactic structure to a sentence. This chapter focuses on constituency structures, those assigned by context-free grammars of the kind described in Chapter 12. In the next chapter we'll introduce dependency parses, an alternative kind of parse structure,

Parse trees can be used in applications such as **grammar checking**: sentence that cannot be parsed may have grammatical errors (or at least be hard to read). Parse trees can be an intermediate stage of representation for **semantic analysis** (as we show in Chapter 16) and thus play a role in applications like **question answering**. For example to answer the question

Which flights to Denver depart before the Seattle flight?

we'll need to know that the questioner wants a list of flights going to Denver, not flights going to Seattle, and parse structure (knowing that *to Denver* modifies *flights*, and *which flights to Denver* is the subject of the *depart*) can help us.

We begin by discussing ambiguity and the problems it presents, and then give the Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965, Younger 1967), the standard dynamic programming approach to syntactic parsing. We've already seen other dynamic programming algorithms like minimum edit distance (Chapter 2) and Viterbi (Chapter 8).

The vanilla CKY algorithm returns an efficient representation of the set of parse trees for a sentence, but doesn't tell us **which** parse tree is the right one. For that, we need to augment CKY with scores for each possible constituent. We'll see how to do this with neural span-based parsers. And we'll introduce other methods like **supertagging** for parsing CCG, **partial parsing methods**, for use in situations in which a superficial syntactic analysis of an input may be sufficient, and the standard set of metrics for evaluating parser accuracy.

13.1 Ambiguity

structural ambiguity

Ambiguity is the most serious problem faced by syntactic parsers. Chapter 8 introduced the notions of **part-of-speech ambiguity** and **part-of-speech disambiguation**. Here, we introduce a new kind of ambiguity, called **structural ambiguity**, illustrated with a new toy grammar $\mathscr{L}_1$, shown in Figure 13.1, which adds a few rules to the $\mathscr{L}_0$ grammar from the last chapter.

Structural ambiguity occurs when the grammar can assign more than one parse to a sentence. Groucho Marx's well-known line as Captain Spaulding in *Animal Crackers* is ambiguous because the phrase *in my pajamas* can be part of the *NP*

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

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

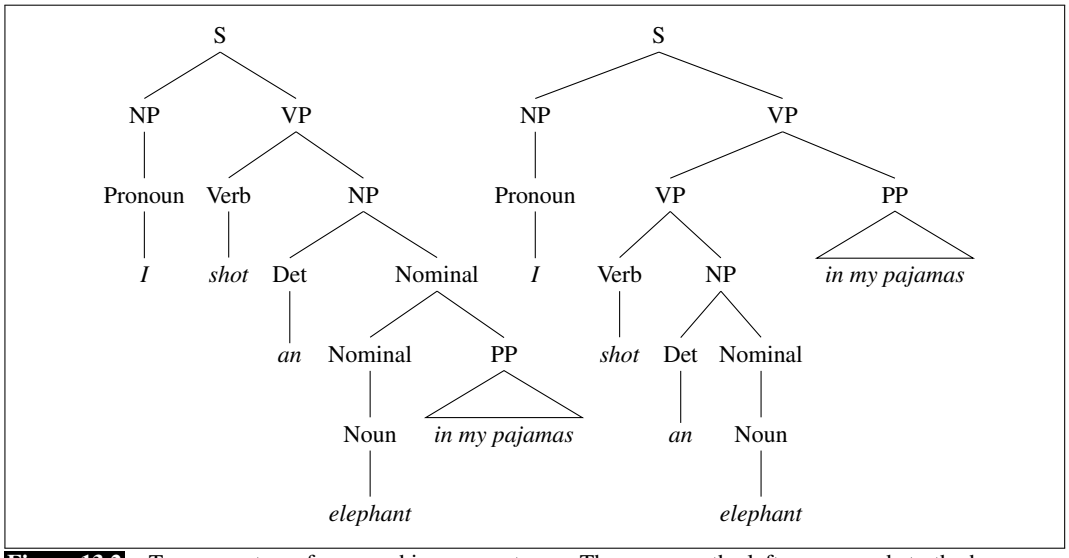


Figure 13.2 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

headed by *elephant* or a part of the verb phrase headed by *shot*. Figure 13.2 illustrates these two analyses of Marx's line using rules from $\mathscr{L}_1$.

attachment ambiguity

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

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

coordination ambiguity

In **coordination ambiguity** 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, like the following news sentence from the Brown corpus:

(13.2) 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 grammatically correct but semantically unreasonable parses for naturally occurring sentences is an irksome problem that affects all parsers. Fortunately, the CKY algorithm below is designed to efficiently handle structural ambiguities. And as we'll see in the following section, we can augment CKY with neural methods to choose a single correct parse by **syntactic disambiguation**.

syntactic disambiguation

13.2 CKY Parsing: A Dynamic Programming Approach

Dynamic programming provides a powerful framework for addressing the problems caused by ambiguity in grammars. Recall that a dynamic programming approach systematically fills in a table of solutions to sub-problems. The complete table has the solution to all the sub-problems needed to solve the problem as a whole. In the case of syntactic parsing, these sub-problems represent parse trees for all the constituents detected in the input.

The dynamic programming advantage arises from the context-free nature of our grammar rules — once a constituent has been discovered in a segment of the input we can record its presence and make it available for use in any subsequent derivation that might require it. This provides both time and storage efficiencies since subtrees can be looked up in a table, not reanalyzed. This section presents the Cocke-Kasami-Younger (CKY) algorithm, the most widely used dynamic-programming based approach to parsing. **Chart parsing** (Kaplan 1973, Kay 1982) is a related approach, and dynamic programming methods are often referred to as **chart parsing** methods.

chart parsing

13.2.1 Conversion to Chomsky Normal Form

The CKY algorithm requires grammars to first 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 non-terminals or to a single terminal. 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.

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-hand side, 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, resulting in the following new rules:

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

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.

Figure 13.3 shows the results of applying this entire conversion procedure to the $\mathscr{L}_1$ grammar introduced earlier on page 2. 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 13.3 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.

$\mathscr{L}_1$ **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 13.3 $\mathscr{L}_1$ 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.

13.2.2 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 two-dimensional matrix can be used to encode the structure of an entire tree. For a sentence of length n, we will work with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix. Each cell $[i, j]$ in this matrix contains the 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 $_0\ Book\ _1\ that\ _2\ flight\ _3$). These gaps are often called **fenceposts**, on the metaphor of the posts between segments of fencing. It follows then that the cell that represents the entire input resides in position $[0, n]$ in the matrix.

fenceposts

Since each non-terminal entry in our table has two daughters in the parse, it follows that for each constituent represented by an entry $[i, j]$, 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. 13.4.

(13.3) Book the flight through Houston.

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

Given this setup, CKY recognition consists 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

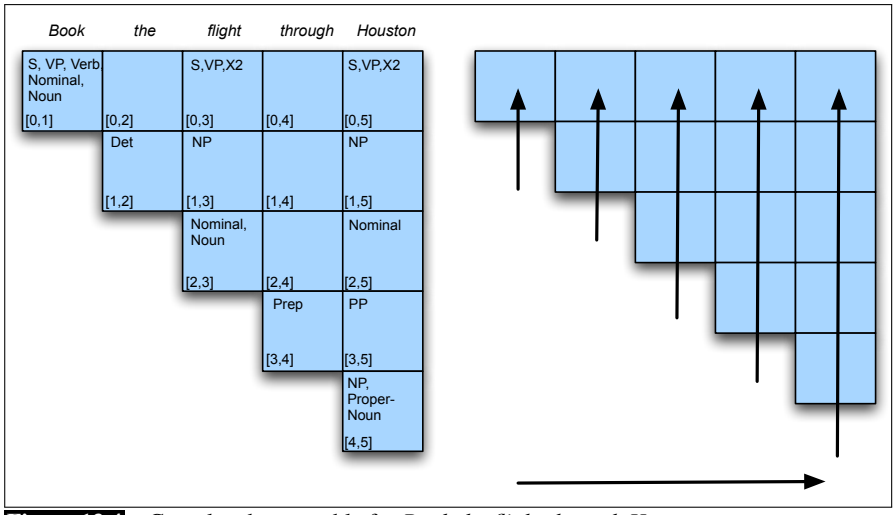


Figure 13.4 Completed parse table for *Book the flight through Houston.*

this entry (i.e., the cells to the left and the cells below) have already been filled. The algorithm given in Fig. 13.5 fills the upper-triangular matrix a column at a time working from left to right, with each column filled from bottom to top, as the right side of Fig. 13.4 illustrates. 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 processing, since filling the columns from left to right corresponds to processing each word one at a time.

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

for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  for all $\{A \mid A \rightarrow words[j] \in grammar\}$
    $table[j-1, j] \leftarrow table[j-1, j] \cup A$
  for $i \leftarrow$ **from** $j-2$ **down to** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      for all $\{A \mid A \rightarrow BC \in grammar$ **and** $B \in table[i,k]$ **and** $C \in table[k,j]\}$
        $table[i,j] \leftarrow table[i,j] \cup A$

Figure 13.5 The CKY algorithm.

The outermost loop of the algorithm given in Fig. 13.5 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 column j. Figure 13.6 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 13.7 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

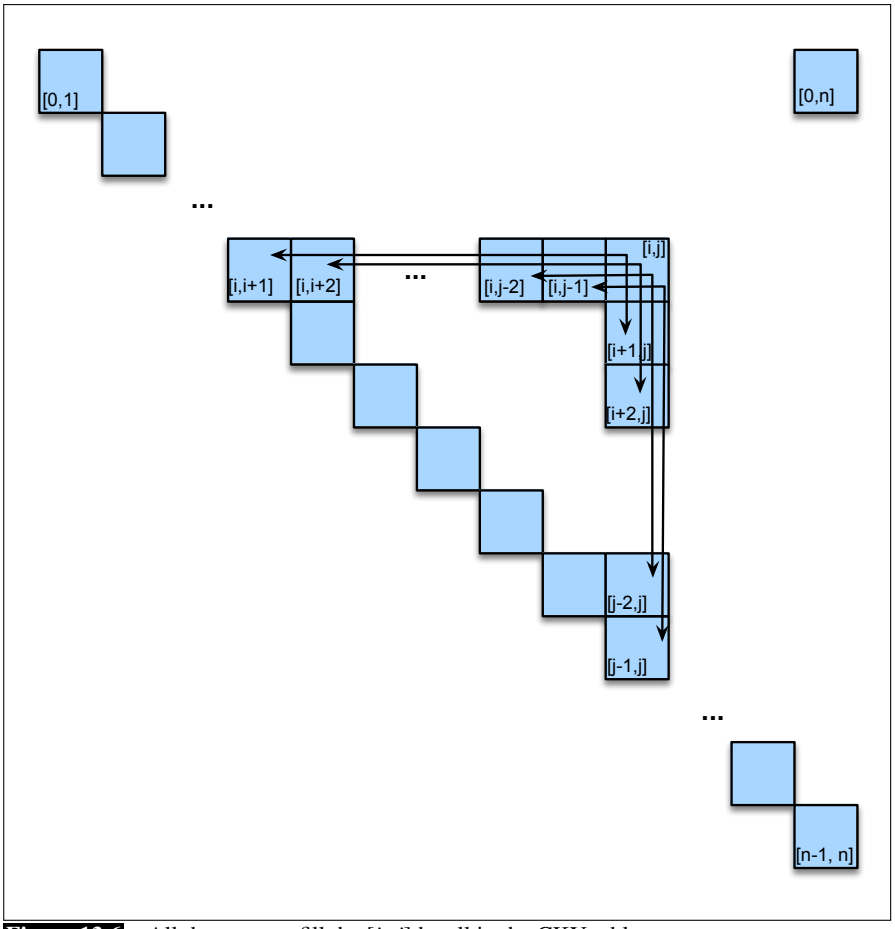


Figure 13.6 All the ways to fill the $[i, j]$th cell in the CKY table.

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.

13.2.3 CKY Parsing

The algorithm given in Fig. 13.5 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. 13.7), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 13.7). 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.

Returning every parse for a sentence may not be useful, since there may be an exponential number of parses. We'll see in the next section how to retrieve only the best parse.

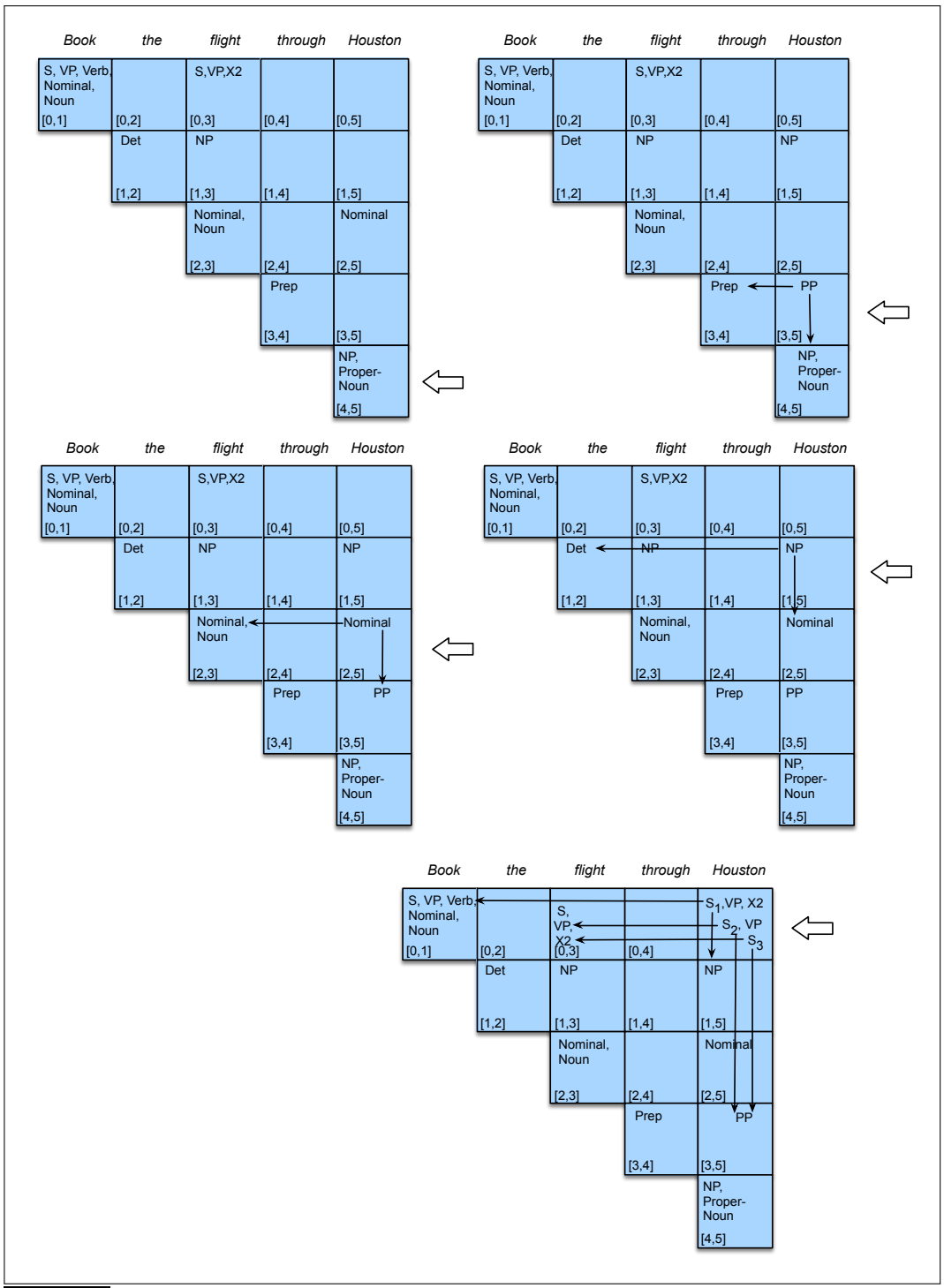


Figure 13.7 Filling the cells of column 5 after reading the word *Houston*.

13.2.4 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 probabilistic parsers presented in Appendix C use the CKY algorithm altered in just this manner.

13.3 Span-Based Neural Constituency Parsing

While the CKY parsing algorithm we've seen so far does great at enumerating all the possible parse trees for a sentence, it has a large problem: it doesn't tell us which parse is the correct one! That is, it doesn't **disambiguate** among the possible parses. To solve the disambiguation problem we'll use a simple neural extension of the CKY algorithm. The intuition of such parsing algorithms (often called **span-based constituency parsing**, or **neural CKY**), is to train a neural classifier to assign a score to each constituent, and then use a modified version of CKY to combine these constituent scores to find the best-scoring parse tree. Here we'll describe a version of the algorithm from Kitaev et al. (2019).

13.3.1 Computing Scores for a Span

span

Let's begin by considering just the constituent (we'll call it a **span**) that lies between fencepost positions i and j with non-terminal symbol label l. We'll build a classifier to assign a score $s(i, j, l)$ to this constituent span.

Fig. 13.8 sketches the architecture. The input word tokens are embedded by passing them through a pretrained language model like BERT. Because BERT operates on the level of subword (wordpiece) tokens rather than words, we'll first need to convert the BERT outputs to word representations. One standard way of doing this is to simply use the last subword unit as the representation for the word (using the first subword unit seems to work equivalently well). The embeddings can then be passed through some postprocessing layers; Kitaev et al. (2019), for example, use 8 Transformer layers.

The resulting word encoder outputs y_t are then use to compute a span score. First, we must map the word encodings (indexed by word positions) to span encodings (indexed by fenceposts). We do this by representing each fencepost with two separate values; the intuition is that a span endpoint to the right of a word represents different information than a span endpoint to the left of a word. We convert each

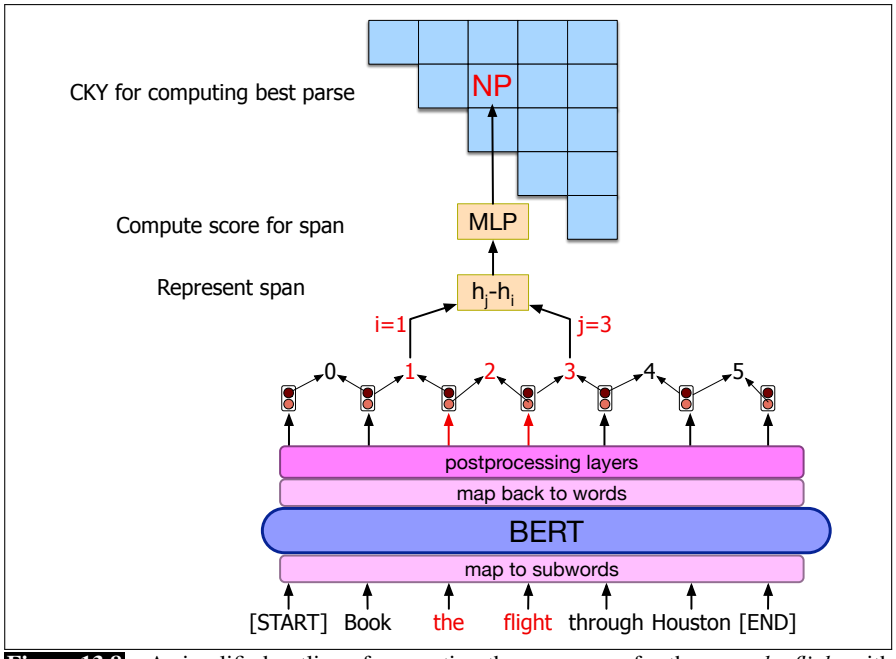
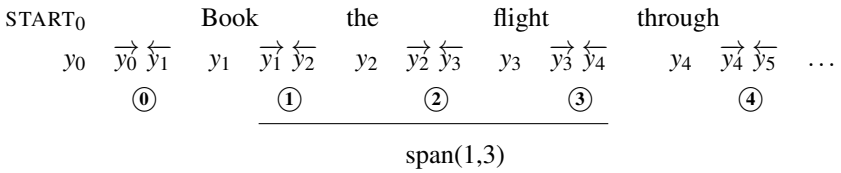


Figure 13.8 A simplified outline of computing the span score for the span *the flight* with the label NP.

word output y_t into a (leftward-pointing) value for spans ending at this fencepost, $\overrightarrow{y}_t$, and a (rightward-pointing) value $\overleftarrow{y}_t$ for spans beginning at this fencepost, by splitting y_t into two halves. Each span then stretches from one double-vector fencepost to another, as in the following representation of *the flight*, which is span(1,3):



A traditional way to represent a span, developed originally for RNN-based models (Wang and Chang, 2016), but extended also to Transformers, is to take the difference between the embeddings of its start and end, i.e., representing span (i, j) by subtracting the embedding of i from the embedding of j. Here we represent a span by concatenating the difference of each of its fencepost components:

$$v(i, j) = [\overrightarrow{y}_j - \overrightarrow{y}_i \; ; \; \overleftarrow{y}_{j+1} - \overleftarrow{y}_{i+1}] \tag{13.4}$$

The span vector v is then passed through an MLP span classifier, with two fully-connected layers and one ReLU activation function, whose output dimensionality is the number of possible non-terminal labels:

$$s(i, j, \cdot) = \mathbf{W}_2 \, \text{ReLU}(\text{LayerNorm}(\mathbf{W}_1 \mathbf{v}(i, j))) \tag{13.5}$$

The MLP then outputs a score for each possible non-terminal.

13.3.2 Integrating Span Scores into a Parse

Now we have a score for each labeled constituent span $s(i, j, l)$. But we need a score for an entire parse tree. Formally a tree T is represented as a set of $|T|$ such labeled

spans, with the t^{th} span starting at position i_t and ending at position j_t, with label l_t:

$$T = \{(i_t, j_t, l_t) : t = 1, \ldots, |T|\} \tag{13.6}$$

Thus once we have a score for each span, the parser can compute a score for the whole tree $s(T)$ simply by summing over the scores of its constituent spans:

$$s(T) = \sum_{(i,j,l) \in T} s(i, j, l) \tag{13.7}$$

And we can choose the final parse tree as the tree with the maximum score:

$$\hat{T} = \underset{T}{\text{argmax}}\, s(T) \tag{13.8}$$

The simplest method to produce the most likely parse is to greedily choose the highest scoring label for each span. This greedy method is not guaranteed to produce a tree, since the best label for a span might not fit into a complete tree. In practice, however, the greedy method tends to find trees; in their experiments Gaddy et al. (2018) finds that 95% of predicted bracketings form valid trees.

Nonetheless it is more common to use a variant the CKY algorithm to find the full parse. The variant defined in Gaddy et al. (2018) works as follows. Let's define $s_{\text{best}}(i, j)$ as the score of the best subtree spanning (i, j). For spans of length one, we choose the best label:

$$s_{\text{best}}(i, i+1) = \max_{l} s(i, i+1, l) \tag{13.9}$$

For other spans (i, j), the recursion is:

$$\begin{aligned} s_{\text{best}}(i, j) &= \max_{l} s(i, j, l) \\ &+ \max_{k} [s_{\text{best}}(i, k) + s_{\text{best}}(k, j)] \end{aligned} \tag{13.10}$$

For more details on span-based parsing, including the margin-based training algorithm, see Stern et al. (2017), Gaddy et al. (2018), Kitaev and Klein (2018), and Kitaev et al. (2019).

13.4 Evaluating Parsers

PARSEVAL

The standard tool for evaluating parsers that assign a single parse tree to a sentence is the **PARSEVAL** metrics (Black et al., 1991). The PARSEVAL metric measures how much the **constituents** in the hypothesis parse tree look like the constituents in a hand-labeled, **reference** parse. PARSEVAL thus requires a human-labeled reference (or "gold standard") parse tree for each sentence in the test set; we generally draw these reference parses from a treebank like the Penn Treebank.

A constituent in a hypothesis parse C_h of a sentence s is labeled correct if there is a constituent in the reference parse C_r with the same starting point, ending point, and non-terminal symbol. We can then measure the precision and recall just as for tasks we've seen already like named entity tagging:

$$\textbf{labeled recall:} = \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of correct constituents in reference parse of } s}$$

$$\textbf{labeled precision:} = \frac{\text{\# of correct constituents in hypothesis parse of } s}{\text{\# of total constituents in hypothesis parse of } s}$$

As usual, we often report a combination of the two, F_1:

$$F_1 = \frac{2PR}{P+R} \tag{13.11}$$

We additionally use a new metric, crossing brackets, for each sentence s:

cross-brackets: the number of constituents for which the reference parse has a bracketing such as ((A B) C) but the hypothesis parse has a bracketing such as (A (B C)).

For comparing parsers that use different grammars, the PARSEVAL metric includes a canonicalization algorithm for removing information likely to be grammar-specific (auxiliaries, pre-infinitival "to", etc.) and for computing a simplified score (Black et al., 1991). The canonical implementation of the PARSEVAL metrics is called **evalb** (Sekine and Collins, 1997).

evalb

13.5 Partial Parsing

partial parse
shallow parse

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 a text that are likely to contain valuable information.

chunking

One kind 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 content-word parts-of-speech: noun phrases, verb phrases, adjective phrases, and prepositional phrases. The task of finding all the base noun phrases in a text is particularly common. 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:

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

This bracketing notation makes clear the two fundamental tasks that are involved in chunking: segmenting (finding the non-overlapping extents of the chunks) and labeling (assigning the correct tag to the discovered chunks). Some input words may not be part of any chunk, particularly in tasks like base *NP*:

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

What constitutes a syntactic base phrase depends on the application (and whether the phrases come 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 obviates the need to resolve attachment ambiguities. This exclusion does lead to certain oddities, such as *PP*s and *VP*s often consisting solely of their heads. Thus *a flight from Indianapolis to Houston* would be reduced to the following:

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

Chunking Algorithms Chunking is generally done via supervised learning, training a BIO sequence labeler of the sort we saw in Chapter 8 from annotated training data. Recall that in BIO tagging, we have a tag for the beginning (B) and inside (I) of each chunk type, and one for tokens outside (O) any chunk. The following example shows the bracketing notation of (13.12) on page 12 reframed as a tagging task:

(13.15) *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.

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

Since annotation efforts are expensive and time consuming, chunkers usually rely on existing treebanks like the Penn Treebank, extracting syntactic phrases from the full parse constituents of a sentence, finding the appropriate heads 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.

Given a training set, any sequence model can be used to chunk: CRF, RNN, Transformer, etc. As with the evaluation of named-entity taggers, the evaluation of chunkers proceeds by comparing chunker output with gold-standard answers provided by human annotators, using precision, recall, and F_1.

13.6 CCG Parsing

Lexicalized grammar frameworks such as CCG pose problems for which the phrase-based methods we've been discussing are not particularly well-suited. To quickly review, CCG consists of three major parts: a set of categories, a lexicon that associates words with categories, and a set of rules that govern how categories combine in context. Categories can be either atomic elements, such as S and NP, or functions such as $(S\backslash NP)/NP$ which specifies the transitive verb category. Rules specify how functions, their arguments, and other functions combine. For example, the following rule templates, **forward** and **backward function application**, specify the way that functions apply to their arguments.

$$X/Y \;\; Y \;\Rightarrow\; X$$
$$Y \;\; X\backslash Y \;\Rightarrow\; X$$

The first rule applies a function to its argument on the right, while the second looks to the left for its argument. The result of applying either of these rules is the category specified as the value of the function being applied. For the purposes of this discussion, we'll rely on these two rules along with the **forward** and **backward composition** rules and **type-raising**, as described in Chapter 12.

13.6.1 Ambiguity in CCG

As is always the case in parsing, managing ambiguity is the key to successful CCG parsing. The difficulties with CCG parsing arise from the ambiguity caused by the large number of complex lexical categories combined with the very general nature of

the grammatical rules. To see some of the ways that ambiguity arises in a categorial framework, consider the following example.

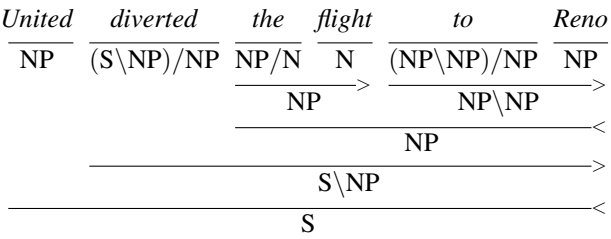
(13.17) United diverted the flight to Reno.

Our grasp of the role of *the flight* in this example depends on whether the prepositional phrase *to Reno* is taken as a modifier of *the flight*, as a modifier of the entire verb phrase, or as a potential second argument to the verb *divert*. In a context-free grammar approach, this ambiguity would manifest itself as a choice among the following rules in the grammar.

$$\begin{aligned} \textit{Nominal} &\rightarrow \textit{Nominal PP} \\ \textit{VP} &\rightarrow \textit{VP PP} \\ \textit{VP} &\rightarrow \textit{Verb NP PP} \end{aligned}$$

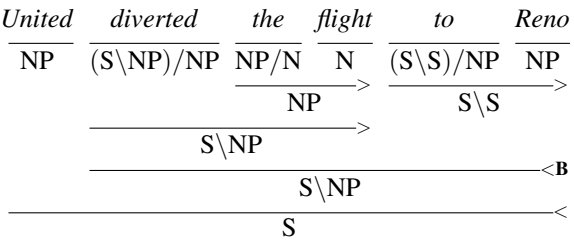
In a phrase-structure approach we would simply assign the word *to* to the category *P* allowing it to combine with *Reno* to form a prepositional phrase. The subsequent choice of grammar rules would then dictate the ultimate derivation. In the categorial approach, we can associate *to* with distinct categories to reflect the ways in which it might interact with other elements in a sentence. The fairly abstract combinatoric rules would then sort out which derivations are possible. Therefore, the source of ambiguity arises not from the grammar but rather from the lexicon.

Let's see how this works by considering several possible derivations for this example. To capture the case where the prepositional phrase *to Reno* modifies *the flight*, we assign the preposition *to* the category $(NP\backslash NP)/NP$, which gives rise to the following derivation.

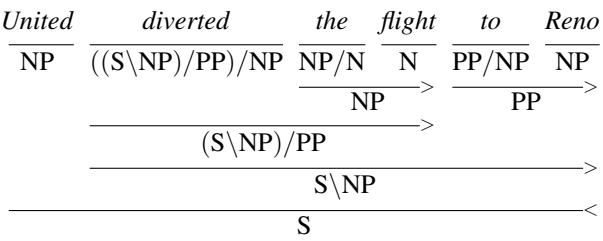


Here, the category assigned to *to* expects to find two arguments: one to the right as with a traditional preposition, and one to the left that corresponds to the *NP* to be modified.

Alternatively, we could assign *to* to the category $(S\backslash S)/NP$, which permits the following derivation where to Reno modifies the preceding verb phrase.



A third possibility is to view *divert* as a ditransitive verb by assigning it to the category $((S\backslash NP)/PP)/NP$, while treating *to Reno* as a simple prepositional phrase.



While CCG parsers are still subject to ambiguity arising from the choice of grammar rules, including the kind of spurious ambiguity discussed in Chapter 12, it should be clear that the choice of lexical categories is the primary problem to be addressed in CCG parsing.

13.6.2 CCG Parsing Frameworks

Since the rules in combinatory grammars are either binary or unary, a bottom-up, tabular approach based on the CKY algorithm should be directly applicable to CCG parsing. Unfortunately, the large number of lexical categories available for each word, combined with the promiscuity of CCG's combinatoric rules, leads to an explosion in the number of (mostly useless) constituents added to the parsing table. The key to managing this explosion of zombie constituents is to accurately assess and exploit the most likely lexical categories possible for each word — a process called supertagging.

The following sections describe two approaches to CCG parsing that make use of supertags. Section 13.6.4, presents an approach that structures the parsing process as a heuristic search through the use of the A* algorithm. The following section then briefly describes a more traditional classifier-based approach that manages the search space complexity through the use of **adaptive supertagging** — a process that iteratively considers more and more tags until a parse is found.

13.6.3 Supertagging

supertagging

Chapter 8 introduced the task of part-of-speech tagging, the process of assigning the correct lexical category to each word in a sentence. **Supertagging** is the corresponding task for highly lexicalized grammar frameworks, where the assigned tags often dictate much of the derivation for a sentence.

CCG supertaggers rely on treebanks such as CCGbank to provide both the overall set of lexical categories as well as the allowable category assignments for each word in the lexicon. CCGbank includes over 1000 lexical categories, however, in practice, most supertaggers limit their tagsets to those tags that occur at least 10 times in the training corpus. This results in a total of around 425 lexical categories available for use in the lexicon. Note that even this smaller number is large in contrast to the 45 POS types used by the Penn Treebank tagset.

As with traditional part-of-speech tagging, the standard approach to building a CCG supertagger is to use supervised machine learning to build a sequence labeler from hand-annotated training data. To find the most likely sequence of tags given a sentence, it is most common to use a neural sequence model, either RNN or Transformer.

It's also possible, however, to use the CRF tagging model described in Chapter 8, using similar features; the current word w_i, its surrounding words within l words, local POS tags and character suffixes, and the supertag from the prior timestep,

training by maximizing log-likelihood of the training corpus and decoding via the Viterbi algorithm as described in Chapter 8.

Unfortunately the large number of possible supertags combined with high per-word ambiguity leads the naive CRF algorithm to error rates that are too high for practical use in a parser. The single best tag sequence $\hat{T}$ will typically contain too many incorrect tags for effective parsing to take place. To overcome this, we instead return a probability distribution over the possible supertags for each word in the input. The following table illustrates an example distribution for a simple sentence, in which each column represents the probability of each supertag for a given word *in the context of the input sentence*. The "..." represent all the remaining supertags possible for each word.

United	serves	Denver
N/N: 0.4	$(S\backslash NP)/NP$: 0.8	NP: 0.9
NP: 0.3	N: 0.1	N/N: 0.05
S/S: 0.1	...	...
$S\backslash S$: .05		
...		

To get the probability of each possible word/tag pair, we'll need to sum the probabilities of all the supertag sequences that contain that tag at that location. This can be done with the forward-backward algorithm that is also used to train the CRF, described in Appendix A.

13.6.4 CCG Parsing using the A* Algorithm

The A* algorithm is a heuristic search method that employs an agenda to find an optimal solution. Search states representing partial solutions are added to an agenda based on a cost function, with the least-cost option being selected for further exploration at each iteration. When a state representing a complete solution is first selected from the agenda, it is guaranteed to be optimal and the search terminates.

The A* cost function, $f(n)$, is used to efficiently guide the search to a solution. The f-cost has two components: $g(n)$, the exact cost of the partial solution represented by the state n, and $h(n)$ a heuristic approximation of the cost of a solution that makes use of n. When $h(n)$ satisfies the criteria of not overestimating the actual cost, A* will find an optimal solution. Not surprisingly, the closer the heuristic can get to the actual cost, the more effective A* is at finding a solution without having to explore a significant portion of the solution space.

When applied to parsing, search states correspond to edges representing completed constituents. Each edge specifies a constituent's start and end positions, its grammatical category, and its f-cost. Here, the g component represents the current cost of an edge and the h component represents an estimate of the cost to complete a derivation that makes use of that edge. The use of A* for phrase structure parsing originated with Klein and Manning (2003), while the CCG approach presented here is based on the work of Lewis and Steedman (2014).

Using information from a supertagger, an agenda and a parse table are initialized with states representing all the possible lexical categories for each word in the input, along with their f-costs. The main loop removes the lowest cost edge from the agenda and tests to see if it is a complete derivation. If it reflects a complete derivation it is selected as the best solution and the loop terminates. Otherwise, new states based on the applicable CCG rules are generated, assigned costs, and entered

into the agenda to await further processing. The loop continues until a complete derivation is discovered, or the agenda is exhausted, indicating a failed parse. The algorithm is given in Fig. 13.9.

```
function CCG-ASTAR-PARSE(words) returns table or failure

  supertags ← SUPERTAGGER(words)
  for i ← from 1 to LENGTH(words) do
      for all {A | (words[i], A, score) ∈ supertags}
          edge ← MAKEEDGE(i − 1, i, A, score)
          table ← INSERTEDGE(table, edge)
          agenda ← INSERTEDGE(agenda, edge)
  loop do
      if EMPTY?(agenda) return failure
      current ← POP(agenda)
      if COMPLETEDPARSE?(current) return table
      table ← INSERTEDGE(chart, edge)
      for each rule in APPLICABLERULES(edge) do
          successor ← APPLY(rule, edge)
          if successor not ∈ in agenda or chart
              agenda ← INSERTEDGE(agenda, successor)
          else if successor ∈ agenda with higher cost
              agenda ← REPLACEEDGE(agenda, successor)
```

Figure 13.9 A*-based CCG parsing.

Heuristic Functions

Before we can define a heuristic function for our A* search, we need to decide how to assess the quality of CCG derivations. We'll make the simplifying assumption that the probability of a CCG derivation is just the product of the probability of the supertags assigned to the words in the derivation, ignoring the rules used in the derivation. More formally, given a sentence S and derivation D that contains supertag sequence T, we have:

$$P(D,S) = P(T,S) \tag{13.18}$$

$$= \prod_{i=1}^{n} P(t_i|s_i) \tag{13.19}$$

To better fit with the traditional A* approach, we'd prefer to have states scored by a cost function where lower is better (i.e., we're trying to minimize the cost of a derivation). To achieve this, we'll use negative log probabilities to score derivations; this results in the following equation, which we'll use to score completed CCG derivations.

$$P(D,S) = P(T,S) \tag{13.20}$$

$$= \sum_{i=1}^{n} -\log P(t_i|s_i) \tag{13.21}$$

Given this model, we can define our f-cost as follows. The f-cost of an edge is the sum of two components: $g(n)$, the cost of the span represented by the edge, and

$h(n)$, the estimate of the cost to complete a derivation containing that edge (these are often referred to as the **inside** and **outside costs**). We'll define $g(n)$ for an edge using Equation 13.21. That is, it is just the sum of the costs of the supertags that comprise the span.

For $h(n)$, we need a score that approximates but *never overestimates* the actual cost of the final derivation. A simple heuristic that meets this requirement assumes that each of the words in the outside span will be assigned its *most probable supertag*. If these are the tags used in the final derivation, then its score will equal the heuristic. If any other tags are used in the final derivation the f-cost will be higher since the new tags must have higher costs, thus guaranteeing that we will not overestimate.

Putting this all together, we arrive at the following definition of a suitable f-cost for an edge.

$$\begin{aligned} f(w_{i,j},t_{i,j}) &= g(w_{i,j}) + h(w_{i,j}) \qquad (13.22)\\ &= \sum_{k=i}^{j} -\log P(t_k|w_k) + \\ &\quad \sum_{k=1}^{i-1} \min_{t\in tags}(-\log P(t|w_k)) + \sum_{k=j+1}^{N} \min_{t\in tags}(-\log P(t|w_k)) \end{aligned}$$

As an example, consider an edge representing the word *serves* with the supertag N in the following example.

(13.23) United serves Denver.

The g-cost for this edge is just the negative log probability of this tag, $-log_{10}(0.1)$, or 1. The outside h-cost consists of the most optimistic supertag assignments for *United* and *Denver*, which are N/N and NP respectively. The resulting f-cost for this edge is therefore 1.443.

An Example

Fig. 13.10 shows the initial agenda and the progress of a complete parse for this example. After initializing the agenda and the parse table with information from the supertagger, it selects the best edge from the agenda — the entry for *United* with the tag N/N and f-cost 0.591. This edge does not constitute a complete parse and is therefore used to generate new states by applying all the relevant grammar rules. In this case, applying forward application to *United: N/N* and *serves: N* results in the creation of the edge *United serves: N[0,2], 1.795* to the agenda.

Skipping ahead, at the third iteration an edge representing the complete derivation *United serves Denver, S[0,3], .716* is added to the agenda. However, the algorithm does not terminate at this point since the cost of this edge (.716) does not place it at the top of the agenda. Instead, the edge representing *Denver* with the category *NP* is popped. This leads to the addition of another edge to the agenda (type-raising *Denver*). Only after this edge is popped and dealt with does the earlier state representing a complete derivation rise to the top of the agenda where it is popped, goal tested, and returned as a solution.

The effectiveness of the A* approach is reflected in the coloring of the states in Fig. 13.10 as well as the final parsing table. The edges shown in blue (including all the initial lexical category assignments not explicitly shown) reflect states in the search space that never made it to the top of the agenda and, therefore, never

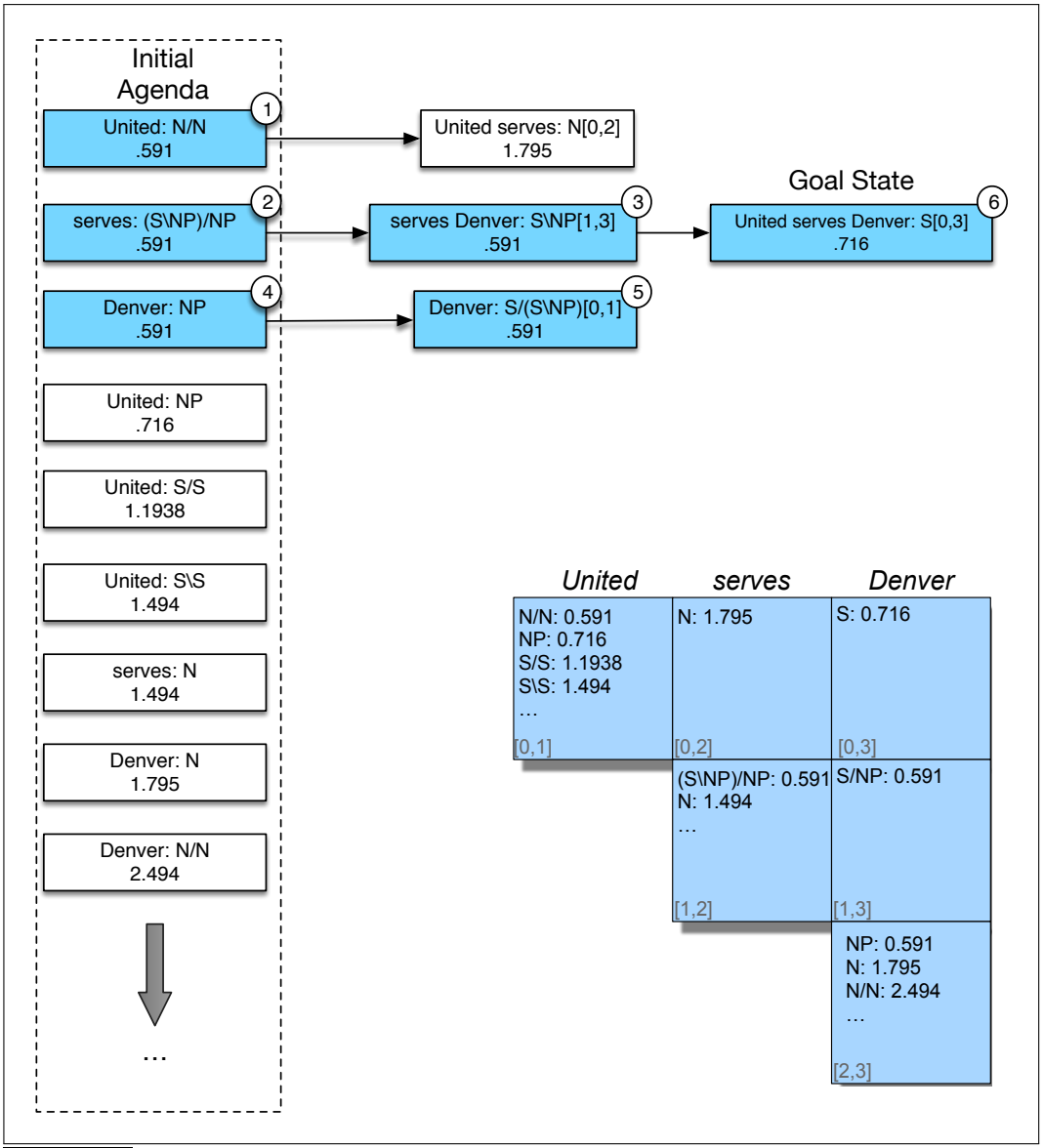


Figure 13.10 Example of an A* search for the example "United serves Denver". The circled numbers on the blue boxes indicate the order in which the states are popped from the agenda. The costs in each state are based on f-costs using negative log_{10} probabilities.

contributed any edges to the final table. This is in contrast to the PCKY approach where the parser systematically fills the parse table with all possible constituents for all possible spans in the input, filling the table with myriad constituents that do not contribute to the final analysis.

13.7 Summary

This chapter introduced constituency parsing. Here's a summary of the main points:

- **Structural ambiguity** is a significant problem for parsers. Common sources of structural ambiguity include **PP-attachment**, **coordination ambiguity**, and **noun-phrase bracketing ambiguity**.
- **Dynamic programming** parsing algorithms, such as **CKY**, use a table of partial parses to efficiently parse ambiguous sentences.
- **CKY** restricts the form of the grammar to Chomsky normal form (CNF).
-
- Parsers are evaluated with three metrics: **labeled recall**, **labeled precision**, and **cross-brackets**.
- **Partial parsing** and **chunking** are methods for identifying shallow syntactic constituents in a text. They are solved by sequence models trained on syntactically-annotated data.

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, since multiple discovery is the norm in science (see page **??**). 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 **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 in Kay (1967) (and Kay 1973). The top-down application of dynamic programming to parsing was described in Earley's Ph.D. dissertation (Earley 1968, Earley 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.

WFST

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 afterward. 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).

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. 13.5 on page 6 so that it can accept grammars that contain unit productions.

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

13.5 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.

13.6 Implement the PARSEVAL metrics described in Section 13.4. Next, use a parser and a treebank, compare your metrics against a standard implementation. Analyze the errors in your approach.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, Vol. 1. Prentice Hall.

Black, E., Abney, S. P., Flickinger, D., Gdaniec, C., Grishman, R., Harrison, P., Hindle, D., Ingria, R., Jelinek, F., Klavans, J. L., Liberman, M. Y., Marcus, M. P., Roukos, S., Santorini, B., and Strzalkowski, T. (1991). A procedure for quantitatively comparing the syntactic coverage of English grammars. *Proceedings DARPA Speech and Natural Language Workshop*.

Church, K. W. (1980). *On Memory Limitations in Natural Language Processing* Master's thesis, MIT. Distributed by the Indiana University Linguistics Club.

Earley, J. (1968). *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Earley, J. (1970). An efficient context-free parsing algorithm. *CACM 6*(8), 451–455.

Ejerhed, E. I. (1988). Finding clauses in unrestricted text by finitary and stochastic methods. *ANLP*.

Gaddy, D., Stern, M., and Klein, D. (2018). What's going on in neural constituency parsers? an analysis. *NAACL HLT*.

Glennie, A. (1960). On the syntax machine and the construction of a universal compiler. Tech. rep. No. 2, Contr. NR 049-141, Carnegie Mellon University (at the time Carnegie Institute of Technology), Pittsburgh, PA.

Harris, Z. S. (1962). *String Analysis of Sentence Structure*. Mouton, The Hague.

Irons, E. T. (1961). A syntax directed compiler for ALGOL 60. *CACM 4*, 51–55.

Kaplan, R. M. (1973). A general syntactic processor. Rustin, R. (Ed.), *Natural Language Processing*, 193–241. Algorithmics Press.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Tech. rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Kay, M. (1967). Experiments with a powerful parser. *Proc. 2eme Conference Internationale sur le Traitement Automatique des Langues*.

Kay, M. (1973). The MIND system. Rustin, R. (Ed.), *Natural Language Processing*, 155–188. Algorithmics Press.

Kay, M. (1982). Algorithm schemata and data structures in syntactic processing. Allén, S. (Ed.), *Text Processing: Text Analysis and Generation, Text Typology and Attribution*, 327–358. Almqvist and Wiksell, Stockholm.

Kitaev, N., Cao, S., and Klein, D. (2019). Multilingual constituency parsing with self-attention and pre-training. *ACL*.

Kitaev, N. and Klein, D. (2018). Constituency parsing with a self-attentive encoder. *ACL*.

Klein, D. and Manning, C. D. (2003). A* parsing: Fast exact Viterbi parse selection. *HLT-NAACL*.

Kuno, S. (1965). The predictive analyzer and a path elimination technique. *CACM 8*(7), 453–462.

Kuno, S. and Oettinger, A. G. (1963). Multiple-path syntactic analyzer. Popplewell, C. M. (Ed.), *Information Processing 1962: Proceedings of the IFIP Congress 1962*. North-Holland.

Lewis, M. and Steedman, M. (2014). A* ccg parsing with a supertag-factored model. *EMNLP*.

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics 17*(1), 91–98.

Sekine, S. and Collins, M. (1997). The evalb software. http://cs.nyu.edu/cs/projects/proteus/evalb.

Sheil, B. A. (1976). Observations on context free parsing. *SMIL: Statistical Methods in Linguistics 1*, 71–109.

Stern, M., Andreas, J., and Klein, D. (2017). A minimal span-based neural constituency parser. *ACL*.

Wang, W. and Chang, B. (2016). Graph-based dependency parsing with bidirectional lstm. *ACL*.

Yngve, V. H. (1955). Syntax and the problem of multiple meaning. Locke, W. N. and Booth, A. D. (Eds.), *Machine Translation of Languages*, 208–226. MIT Press.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control 10*, 189–208.