CHAPTER 18 Context-Free Grammars and Constituency Parsing

Because the Night by Bruce Springsteen and Patti Smith
The Fire Next Time by James Baldwin
If on a winter's night a traveler by Italo Calvino
Love Actually by Richard Curtis
Suddenly Last Summer by Tennessee Williams
A Scanner Darkly by Philip K. Dick

Six titles that are not constituents, from Geoffrey K. Pullum on Language Log (who was pointing out their incredible rarity).

One morning I shot an elephant in my pajamas.
How he got into my pajamas I don't know.

Groucho Marx, *Animal Crackers*, 1930

syntax

The study of grammar has an ancient pedigree. The grammar of Sanskrit was described by the Indian grammarian Pāṇini sometime between the 7th and 4th centuries BCE, in his famous treatise the Aṣṭādhyāyī ('8 books'). And our word **syntax** comes from the Greek *sýntaxis*, meaning "setting out together or arrangement", and refers to the way words are arranged together. We have seen syntactic notions in previous chapters like the use of part-of-speech categories (Chapter 17). In this chapter and the next one we introduce formal models for capturing more sophisticated notions of grammatical structure and algorithms for parsing these structures.

Our focus in this chapter is **context-free grammars** and the **CKY algorithm** for parsing them. Context-free grammars are the backbone of many formal models of the syntax of natural language (and, for that matter, of computer languages). Syntactic parsing is the task of assigning a syntactic structure to a sentence. Parse trees (whether for context-free grammars or for the dependency or CCG formalisms we introduce in following chapters) 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 **formal semantic analysis**. And parsers and the grammatical structure they assign a sentence are a useful text analysis tool for text data science applications that require modeling the relationship of elements in sentences.

In this chapter we introduce context-free grammars, give a small sample grammar of English, introduce more formal definitions of context-free grammars and grammar normal form, and talk about **treebanks**: corpora that have been annotated with syntactic structure. We then discuss parse ambiguity and the problems it presents, and turn to parsing itself, giving the famous Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965, Younger 1967), the standard dynamic programming approach to syntactic parsing. The 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. Finally, we'll introduce the standard set of metrics for evaluating parser accuracy.

18.1 Constituency

noun phrase

Syntactic constituency is the idea that groups of words can behave as single units, or constituents. Part of developing a grammar involves building an inventory of the constituents in the language. How do words group together in English? Consider the **noun phrase**, a sequence of words surrounding at least one noun. Here are some examples of noun phrases (thanks to Damon Runyon):

Harry the Horse
the Broadway coppers
they
a high-class spot such as Mindy's
the reason he comes into the Hot Box
three parties from Brooklyn

What evidence do we have that these words group together (or "form constituents")? One piece of evidence is that they can all appear in similar syntactic environments, for example, before a verb.

three parties from Brooklyn *arrive*...
a high-class spot such as Mindy's *attracts*...
the Broadway coppers *love*...
they *sit*

But while the whole noun phrase can occur before a verb, this is not true of each of the individual words that make up a noun phrase. The following are not grammatical sentences of English (recall that we use an asterisk (*) to mark fragments that are not grammatical English sentences):

*from *arrive*...
*the *is*...
*as *attracts*...
*spot *sat*...

Thus, to correctly describe facts about the ordering of these words in English, we must be able to say things like "*Noun Phrases can occur before verbs*". Let's now see how to do this in a more formal way!

18.2 Context-Free Grammars

CFG

A widely used formal system for modeling constituent structure in natural language is the **context-free grammar**, or **CFG**. Context-free grammars are also called **phrase-structure grammars**, and the formalism is equivalent to **Backus-Naur form**, or **BNF**. The idea of basing a grammar on constituent structure dates back to the psychologist Wilhelm Wundt (1900) but was not formalized until Chomsky (1956) and, independently, Backus (1959).

rules

lexicon

NP

A context-free grammar consists of a set of **rules** or **productions**, each of which expresses the ways that symbols of the language can be grouped and ordered together, and a **lexicon** of words and symbols. For example, the following productions express that an **NP** (or **noun phrase**) can be composed of either a *ProperNoun* or a determiner (*Det*) followed by a *Nominal*; a *Nominal* in turn can consist of one or more *Nouns*.[1]

$$\begin{array}{rcl} NP & \rightarrow & Det\ Nominal \\ NP & \rightarrow & ProperNoun \\ Nominal & \rightarrow & Noun \mid Nominal\ Noun \end{array}$$

Context-free rules can be hierarchically embedded, so we can combine the previous rules with others, like the following, that express facts about the lexicon:

$$\begin{array}{rcl} Det & \rightarrow & a \\ Det & \rightarrow & the \\ Noun & \rightarrow & flight \end{array}$$

terminal

non-terminal

The symbols that are used in a CFG are divided into two classes. The symbols that correspond to words in the language ("the", "nightclub") are called **terminal** symbols; the lexicon is the set of rules that introduce these terminal symbols. The symbols that express abstractions over these terminals are called **non-terminals**. In each context-free rule, the item to the right of the arrow (→) is an ordered list of one or more terminals and non-terminals; to the left of the arrow is a single non-terminal symbol expressing some cluster or generalization. The non-terminal associated with each word in the lexicon is its lexical category, or part of speech.

A CFG can be thought of in two ways: as a device for generating sentences and as a device for assigning a structure to a given sentence. Viewing a CFG as a generator, we can read the → arrow as "rewrite the symbol on the left with the string of symbols on the right".

So starting from the symbol:	*NP*
we can use our first rule to rewrite *NP* as:	*Det Nominal*
and then rewrite *Nominal* as:	*Noun*
and finally rewrite these parts-of-speech as:	*a flight*

derivation

parse tree

We say the string *a flight* can be derived from the non-terminal *NP*. Thus, a CFG can be used to generate a set of strings. This sequence of rule expansions is called a **derivation** of the string of words. It is common to represent a derivation by a **parse tree** (commonly shown inverted with the root at the top). Figure 18.1 shows the tree representation of this derivation.

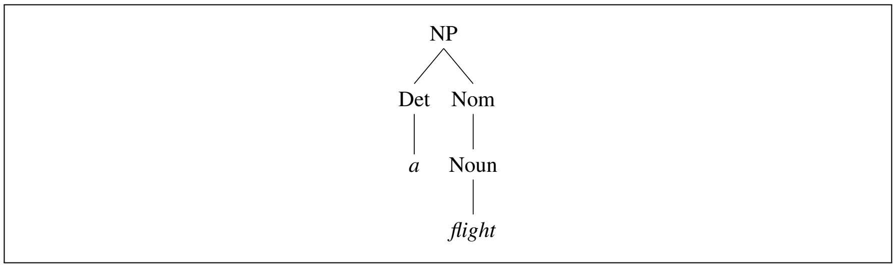


Figure 18.1 A parse tree for "a flight".

dominates

In the parse tree shown in Fig. 18.1, we can say that the node *NP* **dominates** all the nodes in the tree (*Det*, *Nom*, *Noun*, *a*, *flight*). We can say further that it immediately dominates the nodes *Det* and *Nom*.

start symbol

The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**. Each grammar must have one designated start

[1] When talking about these rules we can pronounce the rightarrow → as "goes to", and so we might read the first rule above as "NP goes to Det Nominal".

symbol, which is often called S. Since context-free grammars are often used to define sentences, S is usually interpreted as the “sentence” node, and the set of strings that are derivable from S is the set of sentences in some simplified version of English.

verb phrase

Let’s add a few additional rules to our inventory. The following rule expresses the fact that a sentence can consist of a noun phrase followed by a **verb phrase**:

$S \rightarrow NP\ VP$ I prefer a morning flight

A verb phrase in English consists of a verb followed by assorted other things; for example, one kind of verb phrase consists of a verb followed by a noun phrase:

$VP \rightarrow Verb\ NP$ prefer a morning flight

Or the verb may be followed by a noun phrase and a prepositional phrase:

$VP \rightarrow Verb\ NP\ PP$ leave Boston in the morning

Or the verb phrase may have a verb followed by a prepositional phrase alone:

$VP \rightarrow Verb\ PP$ leaving on Thursday

A prepositional phrase generally has a preposition followed by a noun phrase. For example, a common type of prepositional phrase in the ATIS corpus is used to indicate location or direction:

$PP \rightarrow Preposition\ NP$ from Los Angeles

The *NP* inside a *PP* need not be a location; *PPs* are often used with times and dates, and with other nouns as well; they can be arbitrarily complex. Here are ten examples from the ATIS corpus:

to Seattle
in Minneapolis
on Wednesday
in the evening
on the ninth of July
on these flights
about the ground transportation in Chicago
of the round trip flight on United Airlines
of the AP fifty seven flight
with a stopover in Nashville

Figure 18.2 gives a sample lexicon, and Fig. 18.3 summarizes the grammar rules we’ve seen so far, which we’ll call $\mathscr{L}_0$. Note that we can use the or-symbol | to indicate that a non-terminal has alternate possible expansions.

Noun →	*flights* | *flight* | *breeze* | *trip* | *morning*
Verb →	*is* | *prefer* | *like* | *need* | *want* | *fly* | *do*
Adjective →	*cheapest* | *non-stop* | *first* | *latest*
	| *other* | *direct*
Pronoun →	*me* | *I* | *you* | *it*
Proper-Noun →	*Alaska* | *Baltimore* | *Los Angeles*
	| *Chicago* | *United* | *American*
Determiner →	*the* | *a* | *an* | *this* | *these* | *that*
Preposition →	*from* | *to* | *on* | *near* | *in*
Conjunction →	*and* | *or* | *but*

Figure 18.2 The lexicon for $\mathscr{L}_0$.

We can use this grammar to generate sentences of this “ATIS-language”. We start with S, expand it to *NP VP*, then choose a random expansion of *NP* (let’s say, to

Grammar Rules			**Examples**
S	$\rightarrow$	*NP VP*	I + want a morning flight
NP	$\rightarrow$	*Pronoun*	I
	$\mid$	*Proper-Noun*	Los Angeles
	$\mid$	*Det Nominal*	a + flight
Nominal	$\rightarrow$	*Nominal Noun*	morning + flight
	$\mid$	*Noun*	flights
VP	$\rightarrow$	*Verb*	do
	$\mid$	*Verb NP*	want + a flight
	$\mid$	*Verb NP PP*	leave + Boston + in the morning
	$\mid$	*Verb PP*	leaving + on Thursday
PP	$\rightarrow$	*Preposition NP*	from + Los Angeles

Figure 18.3 The grammar for $\mathscr{L}_0$, with example phrases for each rule.

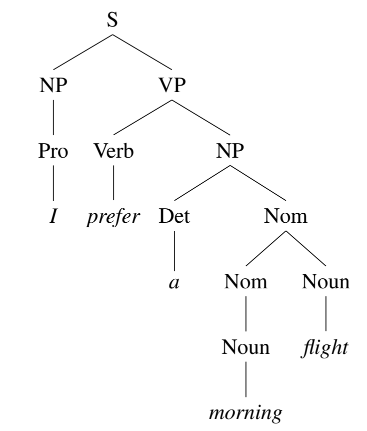


Figure 18.4 The parse tree for "I prefer a morning flight" according to grammar $\mathscr{L}_0$.

I), and a random expansion of *VP* (let's say, to *Verb NP*), and so on until we generate the string *I prefer a morning flight*. Figure 18.4 shows a parse tree that represents a complete derivation of *I prefer a morning flight*.

bracketed notation

We can also represent a parse tree in a more compact format called **bracketed notation**; here is the bracketed representation of the parse tree of Fig. 18.4:

(18.1) $[_S\,[_{NP}\,[_{Pro}\,\text{I}]]\,[_{VP}\,[_V\,\text{prefer}]\,[_{NP}\,[_{Det}\,\text{a}]\,[_{Nom}\,[_N\,\text{morning}]\,[_{Nom}\,[_N\,\text{flight}]]]]]]$

grammatical

ungrammatical

generative grammar

A CFG like that of $\mathscr{L}_0$ defines a formal language. Sentences (strings of words) that can be derived by a grammar are in the formal language defined by that grammar, and are called **grammatical** sentences. Sentences that cannot be derived by a given formal grammar are not in the language defined by that grammar and are referred to as **ungrammatical**. This hard line between "in" and "out" characterizes all formal languages but is only a very simplified model of how natural languages really work. This is because determining whether a given sentence is part of a given natural language (say, English) often depends on the context. In linguistics, the use of formal languages to model natural languages is called **generative grammar** since the language is defined by the set of possible sentences "generated" by the grammar. (Note that this is a different sense of the word 'generate' than when we talk about language models generating text.)

18.2.1 Formal Definition of Context-Free Grammar

We conclude this section with a quick, formal description of a context-free grammar and the language it generates. A context-free grammar G is defined by four parameters: N, Σ, R, S (technically it is a "4-tuple").

N	a set of **non-terminal symbols** (or **variables**)
Σ	a set of **terminal symbols** (disjoint from N)
R	a set of **rules** or productions, each of the form $A \rightarrow \beta$,
	where A is a non-terminal,
	β is a string of symbols from the infinite set of strings $(\Sigma \cup N)^*$
S	a designated **start symbol** and a member of N

For the remainder of the book we adhere to the following conventions when discussing the formal properties of context-free grammars (as opposed to explaining particular facts about English or other languages).

Capital letters like A, B, and S	Non-terminals
S	The start symbol
Lower-case Greek letters like α, β, and γ	Strings drawn from $(\Sigma \cup N)^*$
Lower-case Roman letters like u, v, and w	Strings of terminals

A language is defined through the concept of derivation. One string derives another one if it can be rewritten as the second one by some series of rule applications. More formally, following Hopcroft and Ullman (1979),

directly derives

> if $A \rightarrow \beta$ is a production of R and α and γ are any strings in the set $(\Sigma \cup N)^*$, then we say that $\alpha A \gamma$ **directly derives** $\alpha \beta \gamma$, or $\alpha A \gamma \Rightarrow \alpha \beta \gamma$.

Derivation is then a generalization of direct derivation:

> Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be strings in $(\Sigma \cup N)^*, m \geq 1$, such that

$$\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{m-1} \Rightarrow \alpha_m$$

derives

> We say that α_1 **derives** α_m, or $\alpha_1 \overset{*}{\Rightarrow} \alpha_m$.

We can then formally define the language $\mathscr{L}_G$ generated by a grammar G as the set of strings composed of terminal symbols that can be derived from the designated start symbol S.

$$\mathscr{L}_G = \{w | w \text{ is in } \Sigma^* \text{ and } S \overset{*}{\Rightarrow} w\}$$

syntactic parsing

The problem of mapping from a string of words to its parse tree is called **syntactic parsing**, as we'll see in Section 18.6.

18.3 Treebanks

treebank

A corpus in which every sentence is annotated with a parse tree is called a **treebank**.

Treebanks play an important role in parsing as well as in linguistic investigations of syntactic phenomena.

Penn Treebank

Treebanks are generally made by running a parser over each sentence and then having the resulting parse hand-corrected by human linguists. Figure 18.5 shows sentences from the **Penn Treebank** project, which includes various treebanks in English, Arabic, and Chinese. The Penn Treebank part-of-speech tagset was defined in Chapter 17, but we'll see minor formatting differences across treebanks. The use of LISP-style parenthesized notation for trees is extremely common and resembles the bracketed notation we saw earlier in (18.1). For those who are not familiar with it we show a standard node-and-line tree representation in Fig. 18.6.

```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```

(a)

```
((S
   (NP-SBJ The/DT flight/NN )
   (VP should/MD
     (VP arrive/VB
       (PP-TMP at/IN
         (NP eleven/CD a.m/RB ))
       (NP-TMP tomorrow/NN )))))
```

(b)

Figure 18.5 Parses from the LDC Treebank3 for (a) Brown and (b) ATIS sentences.

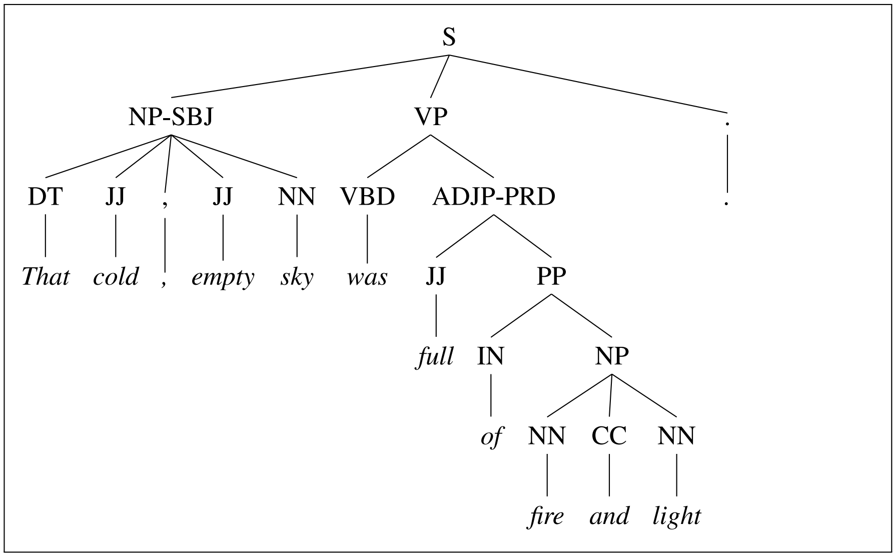


Figure 18.6 The tree corresponding to the Brown corpus sentence in the previous figure.

The sentences in a treebank implicitly constitute a grammar of the language. For example, from the parsed sentences in Fig. 18.5 we can extract the CFG rules shown in Fig. 18.7 (with rule suffixes (-SBJ) stripped for simplicity). The grammar used to parse the Penn Treebank is very flat, resulting in very many rules. For example,

Grammar	Lexicon
S → *NP VP .*	*DT* → *the* | *that*
S → *NP VP*	*JJ* → *cold* | *empty* | *full*
NP → *CD RB*	*NN* → *sky* | *fire* | *light* | *flight* | *tomorrow*
NP → *DT NN*	*CC* → *and*
NP → *NN CC NN*	*IN* → *of* | *at*
NP → *DT JJ , JJ NN*	*CD* → *eleven*
NP → *NN*	*RB* → *a.m.*
VP → *MD VP*	*VB* → *arrive*
VP → *VBD ADJP*	*VBD* → *was* | *said*
VP → *MD VP*	*MD* → *should* | *would*
VP → *VB PP NP*	
ADJP → *JJ PP*	
PP → *IN NP*	

Figure 18.7 CFG grammar rules and lexicon from the treebank sentences in Fig. 18.5.

among the approximately 4,500 different rules for expanding VPs are separate rules for PP sequences of any length and every possible arrangement of verb arguments:

```
VP → VBD PP
VP → VBD PP PP
VP → VBD PP PP PP
VP → VBD PP PP PP PP
VP → VB ADVP PP
VP → VB PP ADVP
VP → ADVP VB PP
```

18.4 Grammar Equivalence and Normal Form

strongly equivalent

weakly equivalent

A formal language is defined as a (possibly infinite) set of strings of words. This suggests that we could ask if two grammars are equivalent by asking if they generate the same set of strings. In fact, it is possible to have two distinct context-free grammars generate the same language. We say that two grammars are **strongly equivalent** if they generate the same set of strings *and* if they assign the same phrase structure to each sentence (allowing merely for renaming of the non-terminal symbols). Two grammars are **weakly equivalent** if they generate the same set of strings but do not assign the same phrase structure to each sentence.

normal form

Chomsky normal form

binary branching

It is sometimes useful to have a **normal form** for grammars, in which each of the productions takes a particular form. For example, a context-free grammar is in **Chomsky normal form** (CNF) (Chomsky, 1963) if it is ϵ-free and if in addition each production is either of the form $A \rightarrow B\ C$ or $A \rightarrow a$. That is, the right-hand side of each rule either has two non-terminal symbols or one terminal symbol. Chomsky normal form grammars are **binary branching**, that is they have binary trees (down to the prelexical nodes). We make use of this binary branching property in the CKY parsing algorithm in Section 18.6.

Any context-free grammar can be converted into a weakly equivalent Chomsky normal form grammar. For example, a rule of the form

$$A \rightarrow B\ C\ D$$

can be converted into the following two CNF rules (Exercise 18.1 asks the reader to

Grammar	Lexicon
S → NP VP	*Det → that | this | the | a*
S → Aux NP VP	*Noun → book | flight | meal | money*
S → VP	*Verb → book | include | prefer*
NP → Pronoun	*Pronoun → I | she | me*
NP → Proper-Noun	*Proper-Noun → Houston | United*
NP → Det Nominal	*Aux → does*
Nominal → Noun	*Preposition → from | to | on | near | through*
Nominal → Nominal Noun	
Nominal → Nominal PP	
VP → Verb	
VP → Verb NP	
VP → Verb NP PP	
VP → Verb PP	
VP → VP PP	
PP → Preposition NP	

Figure 18.8 The $\mathscr{L}_1$ miniature English grammar and lexicon.

formulate the complete algorithm):

$$
\begin{aligned}
A &\rightarrow B\ X \\
X &\rightarrow C\ D
\end{aligned}
$$

Sometimes using binary branching can actually produce smaller grammars. For example, the sentences that might be characterized as

```
VP -> VBD NP PP*
```

are represented in the Penn Treebank by this series of rules:

```
VP → VBD NP PP
VP → VBD NP PP PP
VP → VBD NP PP PP PP
VP → VBD NP PP PP PP PP
...
```

but could also be generated by the following two-rule grammar:

```
VP → VBD NP PP
VP → VP PP
```

Chomsky-adjunction

The generation of a symbol A with a potentially infinite sequence of symbols B with a rule of the form `A → A B` is known as **Chomsky-adjunction**.

18.5 Ambiguity

structural ambiguity

Ambiguity is the most serious problem faced by syntactic parsers. Chapter 17 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 18.8, which adds a few rules to the $\mathscr{L}_0$ grammar.

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*

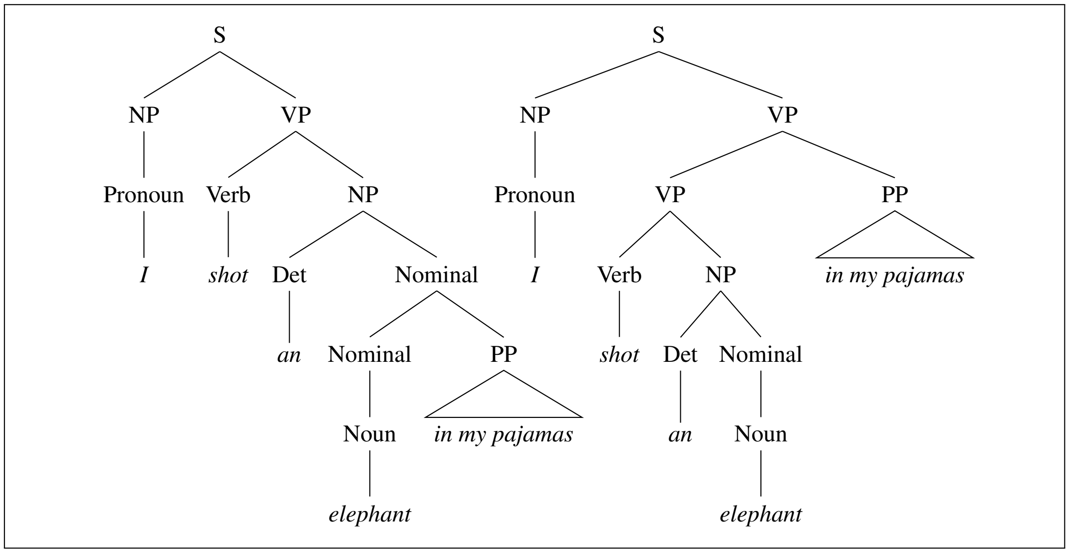


Figure 18.9 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

Crackers is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or a part of the verb phrase headed by *shot*. Figure 18.9 illustrates these two analyses of Marx's line using rules from $\mathscr{L}_1$.

attachment ambiguity

PP-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**: the preposition phrase can be attached either as part of the NP or as part of the VP. 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*:

(18.2) 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:

(18.3) 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*).

syntactic disambiguation

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**.

18.6 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 subproblems. The complete table has the solution to all the subproblems needed to solve the problem as a whole. In the case of syntactic parsing, these subproblems represent parse trees for all the constituents detected in the input.

chart parsing

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.

18.6.1 Conversion to Chomsky Normal Form

The CKY algorithm requires grammars to first be in Chomsky Normal Form (CNF). Recall from Section 18.4 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 $\textit{INF-VP} \rightarrow \textit{to VP}$ would be replaced by the two rules $\textit{INF-VP} \rightarrow \textit{TO VP}$ and $\textit{TO} \rightarrow \textit{to}$.

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 $A \overset{*}{\Rightarrow} B$ 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:

$$\begin{aligned} A &\rightarrow X1\,\gamma \\ X1 &\rightarrow B\,C \end{aligned}$$

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 18.10 shows the results of applying this entire conversion procedure to the $\mathscr{L}_1$ grammar introduced earlier on page 9. 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 18.10 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.

18.6.2 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

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

Figure 18.10 $\mathscr{L}_1$ 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.

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. 18.11.

(18.4) 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 this entry (i.e., the cells to the left and the cells below) have already been filled. The algorithm given in Fig. 18.12 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. 18.11 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 online processing, 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. 18.12 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 18.13 illustrates the general case of filling cell $[i,j]$.

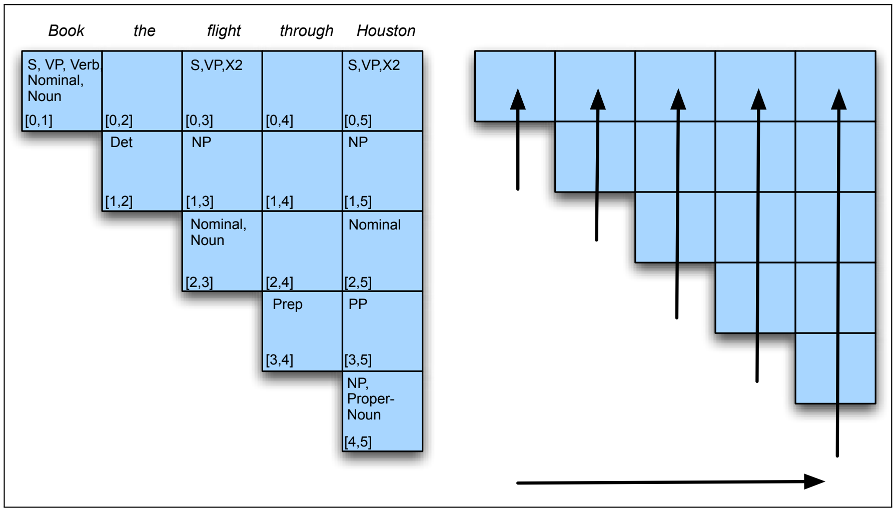


Figure 18.11 Completed parse table for *Book the flight through Houston.*

```
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 18.12 The CKY algorithm.

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 18.14 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* → *Verb NP PP* rule, now captured indirectly with the *VP* → *X2 PP* rule.

18.6.3 CKY Parsing

The algorithm given in Fig. 18.12 is a recognizer, not a parser. That is, it can tell us whether a valid parse exists for a given sentence based on whether or not if finds an S in cell $[0,n]$, but it can't provide the derivation, which is the actual job for a parser. 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. 18.14), the second change is to permit multiple versions of the same non-terminal to be entered into the table (again as shown in Fig. 18.14). With these changes, the completed table contains all the possible parses for a given input. Returning an arbitrary single

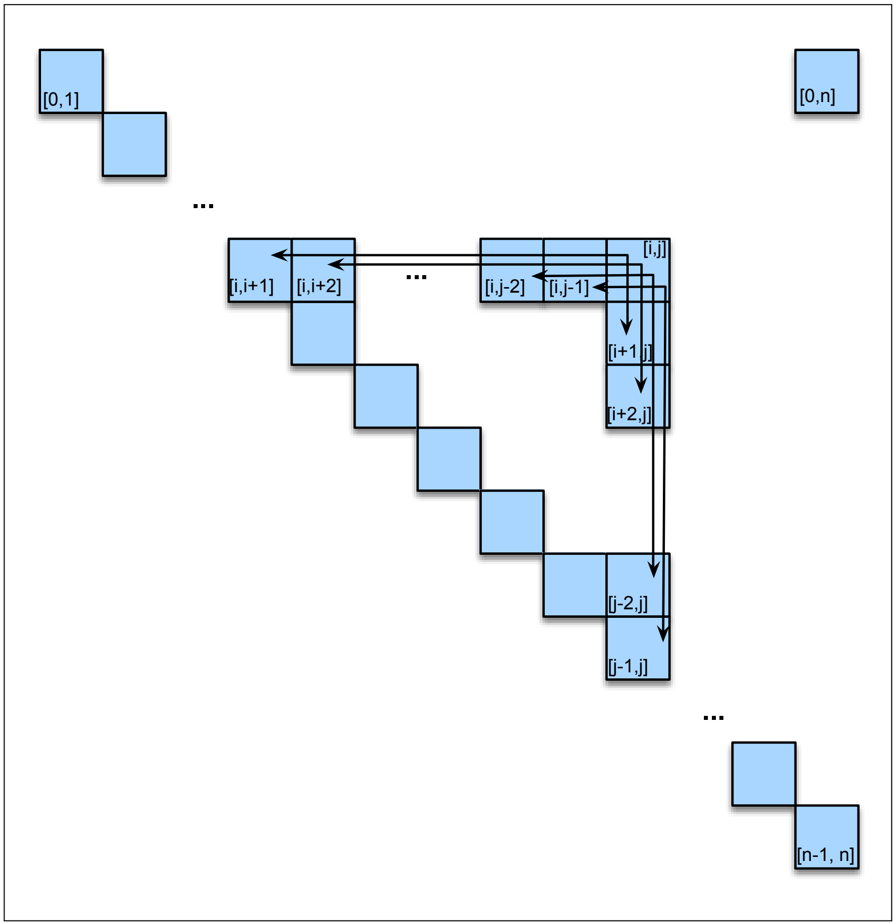


Figure 18.13 All the ways to fill the $[i, j]$th cell in the CKY table.

parse consists of choosing an S from cell $[0,n]$ and then recursively retrieving its component constituents from the table. Of course, instead of returning every parse for a sentence, we usually want just the best parse; we'll see how to do that in the next section.

18.6.4 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. The returned CNF trees may not be consistent with the original grammar built by the grammar developers, and 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 18.3 asks you to make this change. Many of the probabilistic parsers presented in Appendix C use the CKY algorithm altered in

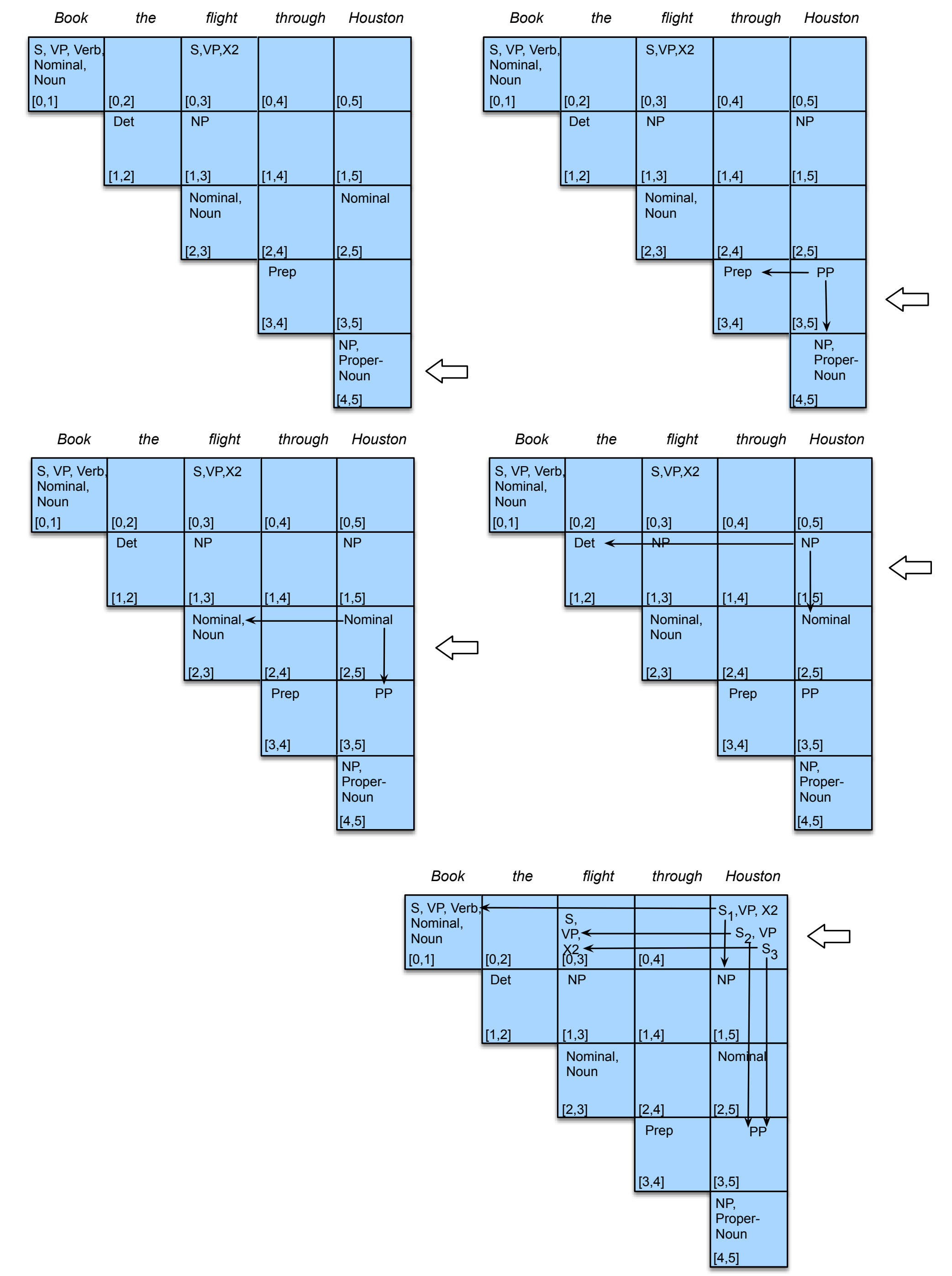


Figure 18.14 Filling the cells of column 5 after reading the word *Houston*.

just this manner.

18.7 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). This parser learns to map a span of words to a constituent, and, like CKY, hierarchically combines larger and larger spans to build the parse-tree bottom-up. But unlike classic CKY, this parser doesn't use the hand-written grammar to constrain what constituents can be combined, instead just relying on the learned neural representations of spans to encode likely combinations.

18.7.1 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 system to assign a score $s(i,j,l)$ to this constituent span.

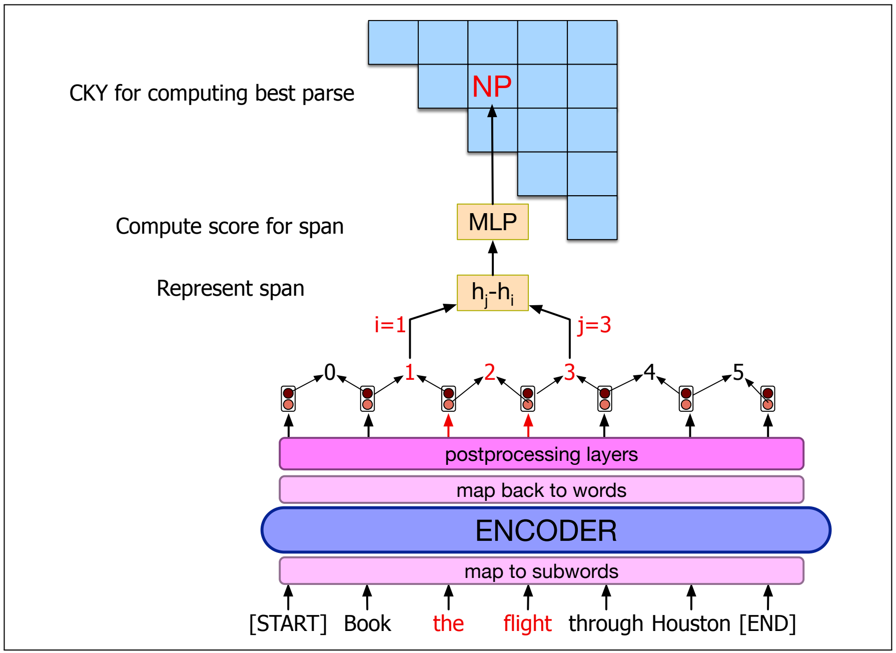
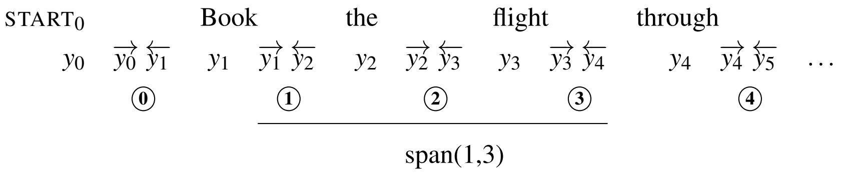


Figure 18.15 A simplified outline of computing the span score for the span *the flight* with the label NP.

Fig. 18.15 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 first subword unit as the representation for the entire word; using the last subword unit, or the sum of all the subword units are also common. 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 used 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 word output y_t into a (leftward-pointing) value for spans ending at this fencepost, $\overleftarrow{y}_t$, and a (rightward-pointing) value $\overrightarrow{y}_t$ 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$(1,3)$:


START$_0$
Book
the
flight
through
y_0
$\overrightarrow{y_0}$ $\overleftarrow{y_1}$
y_1
$\overrightarrow{y_1}$ $\overleftarrow{y_2}$
y_2
$\overrightarrow{y_2}$ $\overleftarrow{y_3}$
y_3
$\overrightarrow{y_3}$ $\overleftarrow{y_4}$
y_4
$\overrightarrow{y_4}$ $\overleftarrow{y_5}$
...
0
1
2
3
4
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:

$$v(i,j) = [\overrightarrow{y_j} - \overrightarrow{y_i}\ ;\ \overleftarrow{y_{j+1}} - \overleftarrow{y_{i+1}}] \tag{18.5}$$

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:

$$s(i,j,\cdot) = \mathbf{W}_2\,\mathrm{ReLU}(\mathrm{LayerNorm}(\mathbf{W}_1\mathbf{v}(i,j))) \tag{18.6}$$

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

18.7.2 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:

$$T = \{(i_t, j_t, l_t) : t = 1, \ldots, |T|\} \tag{18.7}$$

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:

$$s(T) = \sum_{(i,j,l)\in T} s(i,j,l) \tag{18.8}$$

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

$$\hat{T} = \operatorname*{argmax}_{T} s(T) \tag{18.9}$$

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 of 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:

$$s_{\text{best}}(i, i+1) = \max_{l} s(i, i+1, l) \tag{18.10}$$

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

$$\begin{aligned} s_{\text{best}}(i, j) &= \max_{l} s(i, j, l) \\ &+ \max_{k} [s_{\text{best}}(i, k) + s_{\text{best}}(k, j)] \end{aligned} \tag{18.11}$$

Note that the parser is using the max label for span (i, j) + the max labels for spans (i, k) and (k, j) without worrying about whether those decisions make sense given a grammar. The role of the grammar in classical parsing is to help constrain possible combinations of constituents (NPs like to be followed by VPs). By contrast, the neural model seems to learn these kinds of contextual constraints during its mapping from spans to non-terminals.

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).

18.8 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:

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

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

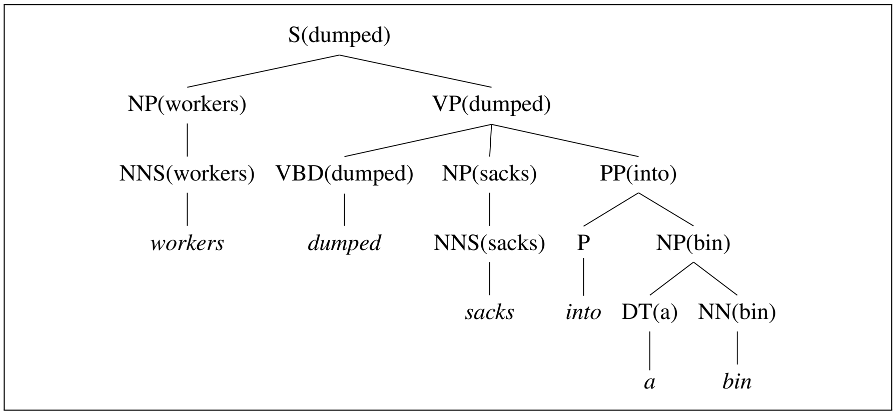


Figure 18.16 A lexicalized tree from Collins (1999).

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

$$F_1 = \frac{2PR}{P+R} \quad (18.12)$$

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

18.9 Heads and Head-Finding

Syntactic constituents can be associated with a lexical **head**; *N* is the head of an *NP*, *V* is the head of a *VP*. This idea of a head for each constituent dates back to Bloomfield 1914, and is central to the dependency grammars and dependency parsing we'll introduce in Chapter 19. Indeed, heads can be used as a way to map between constituency and dependency parses. Heads are also important in probabilistic parsing (Appendix C) and in constituent-based grammar formalisms like Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994)..

In one simple model of lexical heads, each context-free rule is associated with a head (Charniak 1997, Collins 1999). The head is the word in the phrase that is grammatically the most important. Heads are passed up the parse tree; thus, each non-terminal in a parse tree is annotated with a single word, which is its lexical head. Figure 18.16 shows an example of such a tree from Collins (1999), in which each non-terminal is annotated with its head.

For the generation of such a tree, each CFG rule must be augmented to identify one right-side constituent to be the head child. The headword for a node is then set to the headword of its head child. Choosing these head children is simple for textbook examples (*NN* is the head of *NP*) but is complicated and indeed controversial for most phrases. (Should the complementizer *to* or the verb be the head of an infinite verb phrase?) Modern linguistic theories of syntax generally include a component that defines heads (see, e.g., (Pollard and Sag, 1994)).

An alternative approach to finding a head is used in most practical computational systems. Instead of specifying head rules in the grammar itself, heads are identified dynamically in the context of trees for specific sentences. In other words, once a sentence is parsed, the resulting tree is walked to decorate each node with the appropriate head. Most current systems rely on a simple set of handwritten rules, such as a practical one for Penn Treebank grammars given in Collins (1999) but developed originally by Magerman (1995). For example, the rule for finding the head of an *NP* is as follows (Collins, 1999, p. 238):

- If the last word is tagged POS, return last-word.
- Else search from right to left for the first child which is an NN, NNP, NNPS, NX, POS, or JJR.
- Else search from left to right for the first child which is an NP.
- Else search from right to left for the first child which is a $, ADJP, or PRN.
- Else search from right to left for the first child which is a CD.
- Else search from right to left for the first child which is a JJ, JJS, RB or QP.
- Else return the last word

Selected other rules from this set are shown in Fig. 18.17. For example, for *VP* rules of the form $VP \rightarrow Y_1 \cdots Y_n$, the algorithm would start from the left of $Y_1 \cdots Y_n$ looking for the first Y_i of type TO; if no TOs are found, it would search for the first Y_i of type VBD; if no VBDs are found, it would search for a VBN, and so on. See Collins (1999) for more details.

Parent	**Direction**	**Priority List**
ADJP	Left	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	Right	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
PRN	Left	
PRT	Right	RP
QP	Left	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
S	Left	TO IN VP S SBAR ADJP UCP NP
SBAR	Left	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
VP	Left	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP

Figure 18.17 Some head rules from Collins (1999). The head rules are also called a **head percolation table**.

18.10 Summary

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

- In many languages, groups of consecutive words act as a group or a **constituent**, which can be modeled by **context-free grammars** (which are also known as **phrase-structure grammars**).
- A context-free grammar consists of a set of **rules** or **productions**, expressed over a set of **non-terminal** symbols and a set of **terminal** symbols. Formally, a particular **context-free language** is the set of strings that can be **derived** from a particular **context-free grammar**.

- **Structural ambiguity** is a significant problem for parsers. Common sources of structural ambiguity include **PP-attachment** and **coordination 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).
- The basic CKY algorithm compactly represents all possible parses of the sentence but doesn't choose a single best parse.
- Choosing a single parse from all possible parses (**disambiguation**) can be done by **neural constituency parsers**.
- Span-based neural constituency parses 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.
- 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.

Historical Notes

According to Percival (1976), the idea of breaking up a sentence into a hierarchy of constituents appeared in the *Völkerpsychologie* of the groundbreaking psychologist Wilhelm Wundt (Wundt, 1900):

> *...den sprachlichen Ausdruck für die willkürliche Gliederung einer Gesammtvorstellung in ihre in logische Beziehung zueinander gesetzten Bestandteile*
>
> [the linguistic expression for the arbitrary division of a total idea into its constituent parts placed in logical relations to one another]

Wundt's idea of constituency was taken up into linguistics by Leonard Bloomfield in his early book *An Introduction to the Study of Language* (Bloomfield, 1914). By the time of his later book, *Language* (Bloomfield, 1933), what was then called "immediate-constituent analysis" was a well-established method of syntactic study in the United States. By contrast, traditional European grammar, dating from the Classical period, defined relations between *words* rather than constituents, and European syntacticians retained this emphasis on such **dependency** grammars, the subject of Chapter 19. (And indeed, both dependency and constituency grammars have been in vogue in computational linguistics at different times).

American Structuralism saw a number of specific definitions of the immediate constituent, couched in terms of their search for a "discovery procedure": a methodological algorithm for describing the syntax of a language. In general, these attempt to capture the intuition that "The primary criterion of the immediate constituent is the degree in which combinations behave as simple units" (Bazell, 1952/1966, p. 284). The most well known of the specific definitions is Harris' idea of distributional similarity to individual units, with the *substitutability* test. Essentially, the method proceeded by breaking up a construction into constituents by attempting to substitute simple structures for possible constituents—if a substitution of a simple form, say, *man*, was substitutable in a construction for a more complex set (like *intense young man*), then the form *intense young man* was probably a constituent. Harris's test was the beginning of the intuition that a constituent is a kind of equivalence class.

The **context-free grammar** was a formalization of this idea of hierarchical constituency defined in Chomsky (1956) and further expanded upon (and argued against) in Chomsky (1957) and Chomsky (1956/1975). Shortly after Chomsky's initial work, the context-free grammar was reinvented by Backus (1959) and independently by Naur et al. (1960) in their descriptions of the ALGOL programming language; Backus (1996) noted that he was influenced by the productions of Emil Post and that Naur's work was independent of his (Backus') own. After this early work, a great number of computational models of natural language processing were based on context-free grammars because of the early development of efficient parsing algorithms.

Dynamic programming parsing has a history of independent discovery. According to the late 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

The earliest disambiguation algorithms for parsing were based on **probabilistic context-free grammars**, first worked out by Booth (1969) and Salomaa (1969); see Appendix C for more history. Neural methods were first applied to parsing at around the same time as statistical parsing methods were developed (Henderson, 1994). In the earliest work neural networks were used to estimate some of the probabilities for statistical constituency parsers (Henderson, 2003, 2004; Emami and Jelinek, 2005) . The next decades saw a wide variety of neural parsing algorithms, including recursive neural architectures (Socher et al., 2011, 2013), encoder-decoder models (Vinyals et al., 2015; Choe and Charniak, 2016), and the idea of focusing on spans (Cross and Huang, 2016). For more on the span-based self-attention approach we describe in this chapter see Stern et al. (2017), Gaddy et al. (2018), Kitaev and Klein (2018), and Kitaev et al. (2019). See Chapter 19 for the parallel history of neural dependency parsing.

probabilistic context-free grammars

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.

Exercises

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

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

18.3 Rewrite the CKY algorithm given in Fig. 18.12 on page 14 so that it can accept grammars that contain unit productions.

18.4 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.

18.5 Implement the PARSEVAL metrics described in Section 18.8. Next, use a parser and a treebank, compare your metrics against a standard implementation. Analyze the errors in your approach.

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

Backus, J. W. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Information Processing: Proceedings of the International Conference on Information Processing, Paris*. UNESCO.

Backus, J. W. 1996. Transcript of question and answer session. In R. L. Wexelblat, ed., *History of Programming Languages*, page 162. Academic Press.

Bazell, C. E. 1952/1966. The correspondence fallacy in structural linguistics. In E. P. Hamp, F. W. Householder, and R. Austerlitz, eds, *Studies by Members of the English Department, Istanbul University (3), reprinted in Readings in Linguistics II (1966)*, 271–298. University of Chicago Press.

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

Bloomfield, L. 1914. *An Introduction to the Study of Language*. Henry Holt and Company.

Bloomfield, L. 1933. *Language*. University of Chicago Press.

Booth, T. L. 1969. Probabilistic representation of formal languages. *IEEE Conference Record of the 1969 Tenth Annual Symposium on Switching and Automata Theory*.

Charniak, E. 1997. Statistical parsing with a context-free grammar and word statistics. *AAAI*.

Choe, D. K. and E. Charniak. 2016. Parsing as language modeling. *EMNLP*.

Chomsky, N. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chomsky, N. 1956/1975. *The Logical Structure of Linguistic Theory*. Plenum.

Chomsky, N. 1957. *Syntactic Structures*. Mouton.

Chomsky, N. 1963. Formal properties of grammars. In R. D. Luce, R. Bush, and E. Galanter, eds, *Handbook of Mathematical Psychology*, volume 2, 323–418. Wiley.

Collins, M. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia.

Cross, J. and L. Huang. 2016. Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles. *EMNLP*.

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.

Emami, A. and F. Jelinek. 2005. A neural syntactic language model. *Machine learning*, 60(1):195–227.

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

Harris, Z. S. 1946. From morpheme to utterance. *Language*, 22(3):161–183.

Henderson, J. 1994. *Description Based Parsing in a Connectionist Network*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Henderson, J. 2003. Inducing history representations for broad coverage statistical parsing. *HLT-NAACL-03*.

Henderson, J. 2004. Discriminative training of a neural network statistical parser. *ACL*.

Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

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

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

Kay, M. 1967. Experiments with a powerful parser. *COLING*.

Kay, M. 1973. The MIND system. In R. Rustin, ed., *Natural Language Processing*, 155–188. Algorithmics Press.

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

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

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

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

Magerman, D. M. 1995. Statistical decision-tree models for parsing. *ACL*.

Naur, P., J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijnagaarden, and M. Woodger. 1960. Report on the algorithmic language ALGOL 60. *CACM*, 3(5):299–314. Revised in CACM 6:1, 1-17, 1963.

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

Percival, W. K. 1976. On the historical source of immediate constituent analysis. In J. D. McCawley, ed., *Syntax and Semantics Volume 7, Notes from the Linguistic Underground*, 229–242. Academic Press.

Pollard, C. and I. A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

Salomaa, A. 1969. Probabilistic and weighted grammars. *Information and Control*, 15:529–544.

Sekine, S. and M. Collins. 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.

Socher, R., J. Bauer, C. D. Manning, and A. Y. Ng. 2013. Parsing with compositional vector grammars. *ACL*.

Socher, R., C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. 2011. Parsing natural scenes and natural language with recursive neural networks. *ICML*.

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

Vinyals, O., Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. 2015. Grammar as a foreign language. *NeurIPS*.

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

Wundt, W. 1900. *Völkerpsychologie: eine Untersuchung der Entwicklungsgesetze von Sprache, Mythus, und Sitte*. W. Engelmann, Leipzig. Band II: Die Sprache, Zweiter Teil.

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