Chapter 4

Parsing and PCFGs

4.1 Introduction

In natural languages like English, words combine to form phrases, which can themselves combine to form other phrases. For example, in the sentence "Sam thinks Sandy likes the book", the words '*the*' and '*book*' combine to form the noun phrase (NP) '*the book*', which combines with the verb '*likes*' to form the verb phrase (VP) '*likes the book*', which in turn combines with '*Sandy*' to form the embedded clause or sentence (S) '*Sandy likes the book*'. Parsing, which is the process of recovering this kind of structure from a string of words, is the topic of this chapter.

4.1.1 Phrase-structure trees

It's natural to represent this kind of recursive structure as a *tree*, as in Figure 4.1. They are called *phrase-structure trees* because they show how the words and phrases combine to form other phrases.

Notice that trees such as this are usually drawn upside-down, with the *root node* at the top of the tree. The *leaf nodes* of the tree (at the bottom, of course!), which are labeled with words, are also known as *terminal nodes.* The sequence of labels on the terminal nodes is called the *terminal yield* or just the *yield* of the tree; this is the string that the tree describes. The nodes immediately dominating (i.e., above) the terminal nodes are called *preterminal nodes*; they are labeled with the word's part-of-speech (the same parts-of-speech that we saw in the previous chapter). The *phrasal nodes* are above the preterminals; they are labeled with the *category* of the phrase (e.g., '*NP*', '*VP*', etc.). As you would expect, the *nonterminal nodes* are all of the nodes in the tree except the terminal nodes.

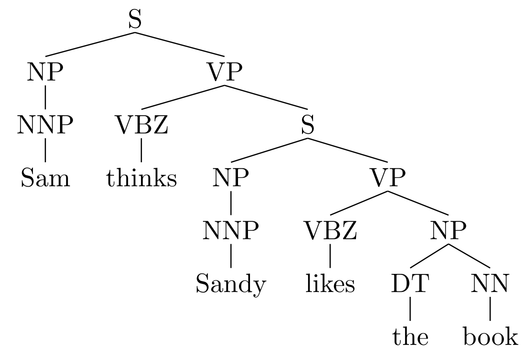

Figure 4.1: A phrase-structure tree for the sentence "Sam thinks Sandy likes the book". The labels on the nonterminal nodes indicate the category of the corresponding phrase, e.g., '*the book*' is an '*NP*' (Noun Phrase).

Parsing is the process of identifying this kind of structure for a sentence. It is one of the best-understood areas of computational linguistics. Arguably *the* best. The literature is very large, and, at least if you are parsing newspaper articles, several good parsers are downloadable from the web.

The best "real world" use of parsing today is in machine translation, for translating between languages with radically different word orders. One might try to view translation as a kind of "rotation" of phrase structure trees (viewed as a kind of mobile sculpture). In the last couple of years this has been shown to work, and now major English-Japanese MT programs use this approach.

This success to some degree justifies the emphasis on parsing, but there was never much doubt (at least among "parsing people") that the area would one day come into wide usage. The reason is simple: people can understand endless new sentences. We conclude from this that we must understand by building the meaning of whole sentences out of the meaning of sentence parts. Syntactic parsing (to a first approximation) tells us what those parts are, and roughly in what order they combine. (Examples like Figure 4.1 should make this seem plausable.) Unfortunately we computational linguists know little of "meanings" and how they combine. When we do, the importance of parsing will be much more obvious.

It's convenient to have a standard (one dimensional) notation for writing phrase structure trees, and one common one is based on the bracketted expressions of the programming language Lisp. Figure 4.2 gives the same tree as in Figure 4.1 in bracket notation.

```
(S (NP (NNP Sam))
   (VP (VBZ thinks)
       (S (NP (NNP Sandy))
          (VP (VBZ likes)
              (NP (DT the)
                  (NN book))))))
```

Figure 4.2: The phrase structure tree of Figure 4.1 for '*Sam thinks Sandy likes the book*' in bracket notation. Note that the indentation is purely for aesthetic reasons; the structure is indicated by the opening and closing brackets.

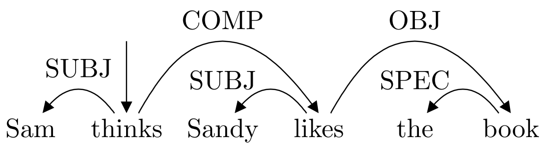

Figure 4.3: A dependency tree for the sentence "Sam thinks Sandy likes the book". The labels on the arcs indicate the type of dependency involved, e.g., '*Sandy*' is the subject of '*likes*'.

4.1.2 Dependency trees

The phrase-structure tree does not directly indicate all of the structure that the sentence has. For example, most phrases consist of a *head* and zero or more *dependents*. Continuing with the previous example, '*likes*' might be analysed as the head of the VP '*likes the book*' and the S '*Sandy likes the book*', where the NPs '*the book*' and '*Sandy*' are both dependents of '*likes*'.

A *dependency tree* makes these dependencies explicit. The nodes in the dependency tree are the words of the sentence, and there is an arc from each head to the heads of all of its dependent phrases. Figure 4.3 depicts a dependency tree for '*Sam thinks Sandy likes the book*'. Sometimes the dependency arcs are labeled to indicate the type of dependency involved; e.g., '*SUBJ*' indicates a subject, '*OBJ*' indicates a direct object and '*COMP*' indicates a verbal complement.

Of course the phrase-structure tree and the dependency tree for a sentence are closely related. For example, it is straightforward to map a phrase-structure tree to an unlabeled dependency tree if one can identify the head

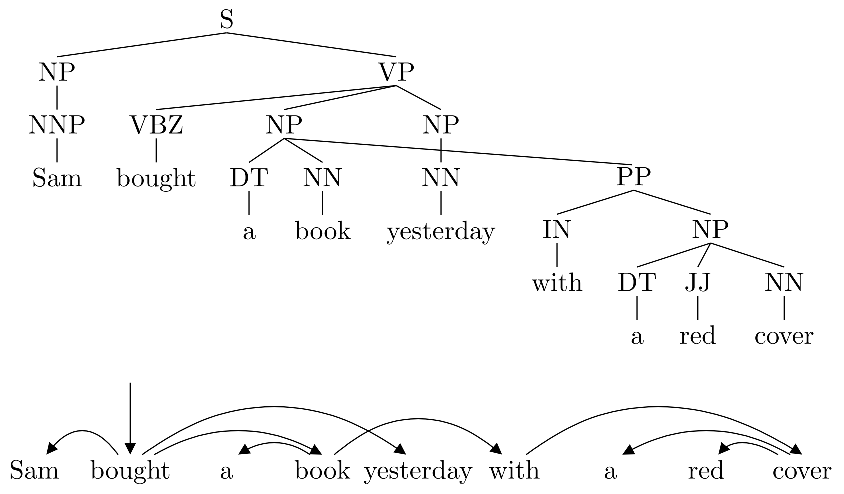

Figure 4.4: A discontinuous structure in English

of each phrase.

There is much more to say about both phrase structure and dependency structure, but we only have space to make a few comments here. Most importantly, there is no agreement on the correct grammar even for a well-studied language like English. For example, what's the structure of a sentence like '*the more, the merrier*'?

It is not even clear that trees are the best representation of phrase structure or dependency structure. For example, some sentences have *discontinuous phrases*, which often correspond to *nonprojective dependency structures* (i.e., dependency structures with crossing dependencies). While these are more common in languages with relatively free word order such as Czech, there are English constructions which plausibly involve such discontinuities. However, as we will see in this chapter, there are such significant computational advantages to only considering structures that can be represented as trees that it is common to do so.

Example 4.1: Figure 4.4. shows an English example involving nonprojective structure. The phrase '*a book with a red cover*' is discontinuous in this example because it is interrupted by '*yesterday*'. Such cases are called "nonprojective" because when projected onto the plane they shows up as "crossing lines" in the phrase structure tree and the dependency structure.

In the face of puzzles and disagreements there are two ways to go, try to resolve them, or ignore them. Work in statistical parsing has mostly done the latter. Dedicated linguists and computational linguists have roughed out grammars for some languages and then hired people to apply their grammar to a corpus of sentences. The result is called a *tree bank.* To the degree that most grammatical formalisms tend to capture the same regularities this can still be a successful strategy even if no one formalism is widely preferred over the rest. This seems to be the case.

4.2 Probabilistic context-free grammars

Probabilistic Context-Free Grammars (PCFGs), are a simple model of phrase-structure trees. We start by explaining what a formal language and a grammar are, and then present *context-free grammars* Context-Free Grammars and PCFGs (their probabilistic counterpart).

4.2.1 Languages and grammars

A *formal language* is a mathematical abstraction of a language. It is defined in terms of a *terminal vocabulary*, which is a finite set $\mathcal{V}$ of *terminal symbols* (or terminals for short), which are the atomic elements out of which the expressions of the language are constructed. For example $\mathcal{V}$ might be a set of English words, or it could be the characters 'a'–'z' (e.g., if we wanted to model the way that words are built out of letters).

Given a set $\mathcal{V}$ of terminals, $\mathcal{W} = \mathcal{V}^\star$ is the set of all finite sequences or strings whose elements are members of $\mathcal{V}$. ($\mathcal{V}^\star$ also includes the *empty string* ϵ). A *language* is a subset of $\mathcal{W}$. A *grammar* is a finite specification of a language. (A language can contain an infinite number of strings, or even if it is finite, it can contain so many strings that it is not practical to list them all). A *probabilistic language* is a probability distribution over $\mathcal{W}$, and a *probabilistic grammar* is a finite specification of a probabilistic language. (We will often drop the "probabilistic" when clear from the context).

A grammar may also provide other information about the strings in the language. For example, it is common in computational linguistics to use probabilistic grammars whose support is $\mathcal{W}$ (i.e., they assign non-zero probability to every string in $\mathcal{W}$), and whose primary purpose is to associate strings with their phrase-structure trees.

$$\begin{aligned}
\mathcal{V} &= \{\text{book}, \text{likes}, \text{Sandy}, \text{Sam}, \text{the}, \text{thinks}\} \\
\mathcal{N} &= \{\text{DT}, \text{NNP}, \text{NP}, \text{S}, \text{VBZ}, \text{VP}\} \\
S &= \text{S} \\
\mathcal{R} &= \left\{ \begin{array}{ll}
\text{DT} \rightarrow \text{the} & \text{NN} \rightarrow \text{book} \\
\text{NNP} \rightarrow \text{Sam} & \text{NNP} \rightarrow \text{Sandy} \\
\text{NP} \rightarrow \text{NNP} & \text{NP} \rightarrow \text{DT NN} \\
\text{S} \rightarrow \text{NP VP} & \text{VBZ} \rightarrow \text{likes} \\
\text{VBZ} \rightarrow \text{thinks} & \text{VP} \rightarrow \text{VBZ NP} \\
\text{VP} \rightarrow \text{VBZ S} &
\end{array} \right\}
\end{aligned}$$

Figure 4.5: A context-free grammar which generates the phrase-structure tree depicted in Figure 4.1. The start symbol is 'S'.

4.2.2 Context-free grammars

A context-free grammar is perhaps the simplest possible model of phrase-structure trees. Formally, a *context-free grammar* (CFG) is a quadruple $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R})$, where $\mathcal{V}$ and $\mathcal{N}$ are disjoint finite sets of *terminal symbols* and *nonterminal symbols* respectively, $S \in \mathcal{N}$ is a distinguished nonterminal called *start symbol*, and $\mathcal{R}$ is a finite set of *rules* or *productions*. A rule $A \rightarrow \beta$ consists of a parent nonterminal $A \in \mathcal{N}$ and children $\beta \in (\mathcal{N} \cup \mathcal{V})^\star$. Figure 4.5 contains an example of a context-free grammar.

A production of the form $A \rightarrow \epsilon$, where ϵ is the empty string, are called an *epsilon rule*. A CFG that does not contain any epsilon rules is *epsilon-free*. While everything we say about CFGs in this chapter generalizes to CFGs that contain epsilon productions, they do complicate the mathematical and computational treatment, so for simplicity we will assume that our grammars are epsilon-free.

Context-free grammars were originally thought of as a rewriting system, which explains the otherwise curious nomenclature and notation. The rewriting process starts with a string that contains only the start symbol 'S'. Each rewriting step consists of replacing some occurence of a nonterminal A in the string with β, where $A \rightarrow \beta \in \mathcal{R}$ is a rule in the grammar. The rewriting process terminates when the string contains no nonterminals, so it is not possible to apply any more rewriting rules. The set of strings that can be produced in this way is the language that the grammar specifies or *generates*.

S
NP VP
NNP VP
Sam VP
Sam VBZ S
Sam thinks S
Sam thinks NP VP
Sam thinks Sandy VP
Sam thinks Sandy VBZ NP
Sam thinks Sandy likes NP
Sam thinks Sandy likes DT NN
Sam thinks Sandy likes the NN
Sam thinks Sandy likes the book

Figure 4.6: A derivation of '*Sam thinks Sandy likes the book*' using the context-free grammar presented in Figure 4.5 on page 106.

Example 4.2: Figure 4.6 gives a derivation of the string "Sam thinks Sandy likes the book" using the grammar of Figure 4.5.

The derivational view of context-free grammars also explains why they are called "context-free". A context-sensitive grammar differs from a context-free one in that the rules come with additional restrictions on the contexts in which they can apply.

Even though historically context-free grammars were first described as rewriting systems, it is probably more useful to think of them as specifying or generating a set of phrase-structure trees. The rules of a context-free grammar can be seen as specifying the possible *local trees* that the tree can contain, where a local tree is a subtree that consists of a parent node and its sequence of children nodes.

Given a tree bank we first divide it into a train/development/test split, and then using the training set to specify the grammar (by reading off the local trees), and when testing, accepting whatever the tree bank says as the gold standard.

In more detail, a context-free grammar $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R})$ generates a tree t iff t's root node is labeled S, its leaf nodes are labeled with terminals from $\mathcal{V}$, and for each local tree ℓ in t, if ℓ's parent node is labeled A and its children are labeled β, then $\mathcal{R}$ contains a rule $A \to \beta$. The set of all trees that G generates is $\mathcal{T}_G$, where we will drop the subscript when clear from the context. G generates a string $w \in \mathcal{W}$ iff G generates a tree that has w as its terminal yield.

Example 4.3: The CFG in Figure 4.5 on page 106 generates the phrase-structure tree depicted in Figure 4.1 on page 102, as well as an infinite number of other trees, including trees for rather bizarre sentences such as '*the book thinks Sam*' as well as impeccable ones such as '*Sandy likes Sam*'.

*Parsing*is the process of taking a CFG G and a string w and returning the subset $\mathcal{T}_G(w)$ of the trees in $\mathcal{T}_G$ that have w as their yield. Note that $\mathcal{T}_G(w)$ can contain an infinite number of trees if $\mathcal{R}$ has epsilon productions or unary productions (i.e., productions of the form $A \to B$ for $A, B \in N$), and even if $\mathcal{T}_G(w)$ is finite its size can grow exponentially with the length of w, so we may be forced to return some kind of finite description of $\mathcal{T}_G(w)$ such as the *packed parse forest* of section 4.3.

One way to think about a CFG is as a kind of "plugging system". We imagine that our terminal and nonterminal symbols are different plug shapes, and that our goal is to somehow connect up a sequence of terminal plugs to a single socket labeled with the start symbol S. Each rule $A \to \beta \in \mathcal{R}$ is a kind of adaptor (maybe a bit like a surge suppressor board) that has a sequence of sockets β and a plug A. When parsing our goal is to find a way of connecting up all of the terminals via the rules such that there are no unconnected plugs or sockets, and everything winds up plugged into the start socket S.

4.2.3 Probabilistic context-free grammars

Probabilistic Context-Free Grammars (PCFGs) extend context-free grammars by associating a probability $\rho_{A\to\beta}$ with each rule $A \to \beta \in \mathcal{R}$ in the grammar. Informally, $\rho_{A\to\beta}$ is the conditional probability that the nonterminal A expands to β. The probability of a tree generated by the PCFG is just the product of the probabilities of the rules used to derive that tree. PCFGs are generative probability models in the sense we described in Section 1.3.3. The above description of how trees are generated in a CFG is their generative story.

More formally, a PCFG G is a quintuple $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \boldsymbol{\rho})$ where $(\mathcal{V}, \mathcal{N}, S, \mathcal{R})$ is a CFG and $\boldsymbol{\rho}$ is a vector of real numbers in $[0, 1]$ that satisfies:

$$\sum_{A\to\beta\in\mathcal{R}_A} \rho_{A\to\beta} \;=\; 1 \tag{4.1}$$

where $\mathcal{R}_A = \{A \to \beta \in R\}$ is the set of all rules in $\mathcal{R}$ whose parent is A. This condition is natural if we interpret $\rho_{A\to\beta}$ as the conditional probability

$\rho_{\text{DT}\to\text{the}}$	$= 1.0$	$\rho_{\text{NN}\to\text{book}}$	$= 1.0$
$\rho_{\text{NNP}\to\text{Sam}}$	$= 0.7$	$\rho_{\text{NNP}\to\text{Sandy}}$	$= 0.3$
$\rho_{\text{NP}\to\text{NNP}}$	$= 0.2$	$\rho_{\text{NP}\to\text{DT NN}}$	$= 0.8$
$\rho_{\text{S}\to\text{NP VP}}$	$= 1.0$	$\rho_{\text{VBZ}\to\text{likes}}$	$= 0.4$
$\rho_{\text{VBZ}\to\text{thinks}}$	$= 0.6$	$\rho_{\text{VP}\to\text{VBZ NP}}$	$= 0.9$
$\rho_{\text{VP}\to\text{VBZ S}}$	$= 0.1$		

Figure 4.7: The rule probability vector $\boldsymbol{\rho}$ which, when combined with CFG in Figure 4.5 on page 106, specifies a PCFG which generates the tree depicted in Figure 4.1 on page 102 with probability approximately 3×10^{-4}.

of A expanding to β. It simply says that the probabiltities of all of the rules expanding A sum to one. Figure 4.7 gives an example of $\boldsymbol{\rho}$ for the CFG presented earlier in Figure 4.5 on page 106.

A PCFG G defines a probability distribution $\mathrm{P}_G(T)$ over trees $T \in \mathcal{T}_G$ as follows:

$$\mathrm{P}_G(T = t) \;=\; \prod_{A\to\beta\in R} {\rho_{A\to\beta}}^{n_{A\to\beta}(t)}$$

where $n_{A\to\beta}(t)$ is the number of times the local tree with parent labeled A and children labeled β appears in t. (This is equivalent to saying that the probability of a tree is the product of the probabilities of all the local trees that make it up.) If $\mathcal{T}_G(w)$ is the set of trees generated by G with yield w, then we define $\mathrm{P}_G(w)$ to be the sum of the probability of the trees in $\mathcal{T}_G(w)$, i.e.,

$$\mathrm{P}_G(W = w) \;=\; \sum_{t\in\mathcal{T}_G(w)} \mathrm{P}(T = t) \tag{4.2}$$

Finally, note that P_G may not define a properly normalized probability distribution on $\mathcal{T}$ or $\mathcal{W}$. Figure 4.8 presents a simple PCFG for which $\sum_{t\in\mathcal{T}} \mathrm{P}(t) < 1$; intuitively this is because this grammar puts its mass on "infinite trees". Fortunately it has been shown that such cases cannot occur for a very wide class of grammars, including all the ones we allude to in this book.

4.2.4 HMMs as a kind of PCFG

PCFGs are strictly more expressive than the HMMs we saw in the previous chapter. That is, for each HMM there is a PCFG that generates the same

$$\begin{aligned}
\mathcal{V} &= \{\mathrm{x}\} \\
\mathcal{N} &= \{\mathrm{S}\} \\
S &= \mathrm{S} \\
\mathcal{R} &= \{\mathrm{S} \to \mathrm{S\ S} \qquad \mathrm{S} \to \mathrm{x}\} \\
\boldsymbol{\rho} &= (\rho_{\mathrm{S}\to\mathrm{S\ S}} = 0.7,\ \rho_{\mathrm{S}\to\mathrm{x}} = 0.3)
\end{aligned}$$

Figure 4.8: A PCFG for which the tree "probabilities" P(t) do not sum to 1. Informally, this is because this grammar puts non-zero mass on infinite trees.

language (with the same probabilities) as the HMM. This parallelism goes further; the algorithms we presented for HMMs in the last chapter can be viewed as special cases of the algorithms for PCFGs that we present below.

Recall that an HMM H is defined in terms of a set of states $\mathcal{Y}$, a state-to-state transition matrix $\boldsymbol{\sigma}$, where $\sigma_{y,y'}$ is the probability of a transition from y to y', and a state-to-output matrix $\boldsymbol{\tau}$, where $\tau_{y,v}$ is the probability of emitting output v from state y. Given an HMM G, we can define a PCFG G_H that generates the same probabilistic language as H as follows. Let $\mathcal{Y}'$ be the set of HMM states except for the begin and end states '$\triangleright$' and '$\triangleleft$'. Then set $G_H = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \rho)$ where $\mathcal{V}$ is the same terminal vocabulary as the HMM, $\mathcal{N} = \{S\} \cup \{A_y, B_y : y \in \mathcal{Y}'\}$ where A_y and B_y are unique symbols distinct from S and each element of $\mathcal{V}$. $\mathcal{R}$ consists of all rules of the form $S \to A_y$, $A_y \to B_y\ A_{y'}$, $A_y \to B_y$ and $B_y \to v$, for all $y, y' \in \mathcal{Y}'$ and $v \in \mathcal{V}$, and $\boldsymbol{\rho}$ is defined as follows:

$$\begin{array}{lcllcl}
\rho_{S\to A_y} & = & \sigma_{\triangleright,y} & \rho_{A_y\to B_y\ A_{y'}} & = & \sigma_{y,y'} \\
\rho_{A_y\to B_y} & = & \sigma_{y,\triangleleft} & \rho_{B_y\to v} & = & \tau_{y,v}
\end{array}$$

Example 4.4: Figure 4.9 shows a parse tree (writen in the form normal for a parse tree) generated from the PCFG corresponding to the example HMM presented in Section 3.3 on page 74. However, the corespondence to HMMs is perhaps clearer if we write the parse tree "on its side" as in Figure 4.10. This figure should also make the above probability rules clearer — e.g, why $\rho_{S\to A_y} = \sigma_{\triangleright,y}$.

Grammars such as these are called *right-linear* ("linear" means that every rule contains at most one nonterminal symbol, and "right-linear" means that that this nonterminal appears rightmost in the list of the rule's children). We've just seen that every HMM generates the same language as

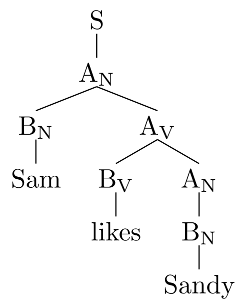

Figure 4.9: A parse tree generated by the PCFG corresponding to the HMM presented in the previous chapter.

$$\begin{tikzcd}
S \arrow[r] & A_N \arrow[r] \arrow[d] & A_V \arrow[r] \arrow[d] & A_N \arrow[d] \\
 & B_N \arrow[d] & B_V \arrow[d] & B_N \arrow[d] \\
 & \text{Sam} & \text{likes} & \text{Sandy}
\end{tikzcd}$$

$$\begin{tikzcd}
\triangleleft \arrow[r] & N \arrow[r] \arrow[d] & V \arrow[r] \arrow[d] & N \arrow[r] \arrow[d] & \triangleleft \\
 & \text{Sam} & \text{likes} & \text{Sandy} &
\end{tikzcd}$$

Figure 4.10: The same parse tree writen on its side, along with the Bayes net for the corresponding HMM

some right-linear PCFG, and it's possible to show that every right-linear PCFG generates the same language as some HMM, so HMMs and right-linear PCFGs can express the same class of languages.

4.2.5 Binarization of PCFGs

Many of the dynamic programming algorithms for CFGs and PCFGs require that their rules be in a special form that permits efficient processing. We will say that a (P)CFG is *binarized* iff all of its productions are all instances of the following schemata:

$A \to v$	:	$A \in \mathcal{N}, v \in \mathcal{V}$	(terminal rules)
$A \to B\ C$	:	$A, B, C \in \mathcal{N}$	(binary rules)
$A \to B$	:	$A, B \in \mathcal{N}$	(unary rules)

Grammars in which all rules are either terminal rules or binary rules are said to be in *Chomsky normal form*. It turns out that for every PCFG G without epsilon rules there is another PCFG G' in Chomsky normal form that generates the same language as G.

Binarized PCFGs are less restrictive that Chomsky normal form because they also permit unary rules (in addition to terminal and binary rules). It turns out that every epsilon-free PCFG G has a corresponding binarized PCFG G' that generates the same language as G, and that the trees generated by G' correspond 1-to-1 with the trees of G. This means that we can map the trees generated by G' to trees generated by G. So in our algorithms below, given a grammar G we find an equivalent binarized grammar G' which we use in algorithm, and then map its output back to the trees of original grammar G.

There are many ways of converting an arbitrary PCFG G to an equivalent binarized PCFG G', but we only describe one of the simplest algorithms here. The key idea is to replace a rule with three or more symbols on the right with several binary rules that accomplish the same thing. So the rule '$A \to B\ C\ D$' would be replaced by

A → B_C D
B_C → B C

The new symbol '*B_C*' can only expand one way, so whenever we use the first of these rules, we always end up with the correct three symbols in our parse.

More formally the binarized PCFG $G' = (\mathcal{V}, \mathcal{N}', S, \mathcal{R}', \rho')$ has the same terminal vocabulary and start symbol as $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \rho)$, but G' has

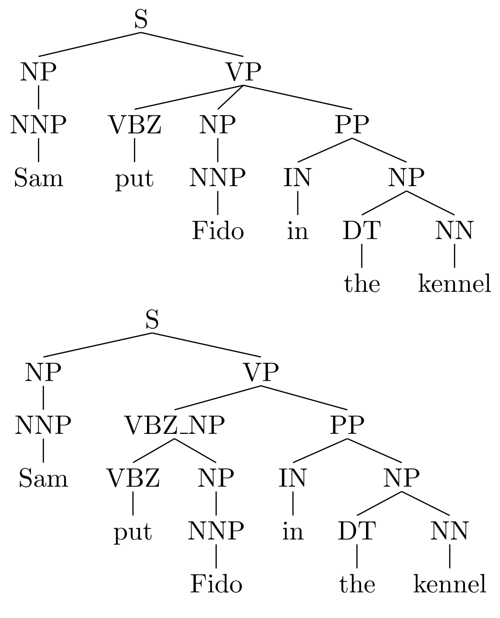

Figure 4.11: A tree generated by a non-binary CFG, and a tree generated by its binarized counterpart

many more nonterminal symbols and rules than G does, e.g., B_C. So the non-terminal set $\mathcal{N}'$ consist of $\mathcal{N}$ plus all *non-empty proper prefixes* of the rules of G. (A proper prefix of a string β is a prefix of β that does not include all of β). We'll write $\beta_{1:i}$ for a symbol concatenating $\beta_1, \ldots, \beta_i$ of β. This method is called *right-branching binarization* because all of the branching occurs along the right-hand-side spine.

Figure 4.11 depicts a tree generated by a non-binarized CFG as well as its binarized counterpart.

More precisely, the rules for defining a binarized grammar G' is given in Figure Figure 4.12. Unary and binary rules probabilities go through untouched. Interestingly, larger rules retain their exactly probabilities as well. So the probability for, say, NP $\rightarrow$ DT_JJ_JJ NN ("the large yellow block") is unchanged from that of the original rule (NP $\rightarrow$ DT JJ JJ NN). The probability of the extra rules introduced by the process, e.g., (DT_JJ_JJ $\rightarrow$ DT_JJ JJ) are all one. This makes sense since there is only one thing that DT_JJ_JJ can expand into, and thus the probability that it will expand that way is one. This means that the sequence of rule expansions that are required in the binarization case have the same probability as the single

$$
\begin{aligned}
\mathcal{N}' &= \mathcal{N} \cup \{\beta_{1:i} : A \to \beta \in \mathcal{R}, 1 < i < |\beta|\} \\
\mathcal{R}' &= \left\{ \begin{array}{lcl} A \to v & : & A \to v \in \mathcal{R} \\ A \to B & : & A \to B \in \mathcal{R} \\ A \to \beta_{1:n-1}\ \beta_n & : & A \to \beta \in \mathcal{R}, |\beta| > 1 \\ \beta_{1:i} \to \beta_{1:i-1}\ \beta_i & : & A \to \beta \in \mathcal{R}, 1 < i < |\beta| \end{array} \right\} \\
\boldsymbol{\rho}' &= \left\{ \begin{array}{lcl} \rho'_{A\to v} = \rho_{A\to v} & : & A \to v \in \mathcal{R} \\ \rho'_{A\to B} = \rho_{A\to B} & : & A \to B \in \mathcal{R} \\ \rho'_{A\to\beta_{1:n-1}\ \beta_n} = \rho_{A\to\beta} & : & A \to \beta \in \mathcal{R} \\ \rho'_{\beta_{1:i}\to\beta_{1:i-1}\ \beta_i} = 1 & : & A \to \beta \in \mathcal{R}, 1 < i < |\beta| \end{array} \right\}
\end{aligned}
$$

Figure 4.12: Rules for defining a binarized PCFG from an unbinarized version

probability in the unbinarized grammar. Also note that different $n$ary rules may share the helper binarized grammar rules. e.g., the rule (NP $\rightarrow$ DT JJ JJ NNS) (NNS is plural common noun) has the binarization (NP $\rightarrow$ DT_JJ_JJ NN), and make use of the rules for expanding DT_JJ_JJ.

Lastly, two points that we come back to in Section 4.7. First, as should already be obvious, binarization dramatically increases the size of the non-terminal vocabulary. Secondly, for the particular binarization method we choose, a new binary non-terminals may never appear as the second non-terminal in a binary rule, only the first. This has implications for efficient parsing.

4.3 Parsing with PCFGs

This section describes how to parse a string $w \in \mathcal{W}$ with a PCFG G. In practice most probabilistic grammars are designed to generate most or all strings in $\mathcal{W}$, so it's not interesting to ask whether w is generated by G. Similarly, the set $\mathcal{T}_G(w)$ will typically include a very large number of trees, many of which will have extremely low probability. Instead, we typically want to find the most likely tree $\hat{t}(w)$ for a string w, or perhaps the set of the m most-likely trees (i.e., the m trees in $\mathcal{T}_G(w)$ with maximum probability).

$$
\begin{aligned}
\hat{t}(w) &= \underset{t\in\mathcal{T}_G(w)}{\operatorname{argmax}}\ \mathrm{P}_G(T = t \mid W = w) \\
&= \underset{t\in\mathcal{T}_G(w)}{\operatorname{argmax}}\ \mathrm{P}_G(T = t, W = w)
\end{aligned}
$$

(However, if we intend to use G as a *language model*, i.e., to predict the probability of a string w, we want to implicitly sum over all $\mathcal{T}_G(w)$ to calculate $\mathrm{P}_G(w)$, as in equation 4.2).

If w is very short and G is sufficiently simple we might be able to enumerate $\mathcal{T}_G(w)$ and identify $\hat{t}(w)$ exhaustively. However, because $\mathcal{T}_G(w)$ can be infinite (and even when it is finite, its size can be exponential in the length of w), in general this is not a practical method.

Instead, we show how to find $\hat{t}(w)$ using a dynamic programming algorithm that directly generalizes the algorithm used to find the most likely state sequence in an HMM given in Section 3.3 on page 74. This algorithm requires that G be a binarized PCFG. We described in the previous section how to map an arbitrary PCFG to an equivalent binarized one, so from here on we will assume that G is binarized. (Note that there are efficient dynamic programming parsing algorithms that do not require G to be binarized, but these usually perform an implicit binarization "on the fly").

It actually simplifies CFG parsing algorithms to use "computer science" zero-based indexing for strings. That is, a string $\boldsymbol{w}$ of length n is made up of elements indexed from 0 to $n-1$, i.e., $\boldsymbol{w} = (w_0, w_1, \ldots, w_{n-1})$. A subsequence or "slice" is defined to be $\boldsymbol{w}_{i,j} = (w_i, \ldots, w_{j-1})$, i.e., it goes up to, but doesn't include element w_j. This in turn is most easily visualized if you think of the indicies as enumerating the positions *between* the words.

Example 4.5: The sentence "Sam likes Sandy" with the indicies 0 to 3 would look like this:

```
      Sam     likes     Sandy
   0       1         2         3
```

So, e.g., $w_{1,3}$ = "likes Sandy".

Just as in the HMM case, we first address the problem of finding the probability $\mu(\boldsymbol{w}) = \max_{t \in \mathcal{T}(w)} \mathrm{P}(t)$ of the most likely parse $\hat{t}(\boldsymbol{w})$ of $\boldsymbol{w}$. Then we describe a method for reading out the most likely tree $\hat{t}(\boldsymbol{w})$ from the dynamic programming parsing table used to compute $\mu(\boldsymbol{w})$.

For a given nonterminal $A \in \mathcal{N}$, let $\mu_A(i,k)$ be the probability of the most likely parse of $\boldsymbol{w_{i,k}}$ when the subtree is rooted in A. Given a string $\boldsymbol{w}$ to parse, we recursively compute $\mu_A(i,k)$ for all for each $A \in \mathcal{N}$ and $0 \leq i < j \leq |\boldsymbol{w}|$. (We use the positions i to k because in a moment we introduce a position j, $i < j < k$.)

The traditional visualization for this is a *chart*. Charts are two dimensional arrays of (i,k). Figure 4.13 shows an empty chart for a three word string, e.g., "Sam likes Sandy". Each diamond is a *cell* of the chart, and

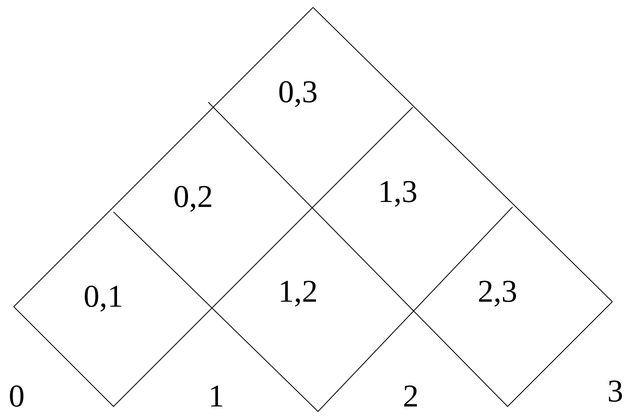

Figure 4.13: Chart for a string of length three

coresponds to possible span of words. So the bottom left-hand cell will be filled with terminal and non-terminals that span the words zero to one — e.g., the first word of the sentence "Sandy". In the same way, the top center cell spans the words zero to three. It is always the top center that contains the start symbol if the string is in the language of the PCFG we are using. Note that the instructions to fill the chart from bottom up coresponds to the pseudo-code:

```
1. for $l = 1$ to $L$
    (a) for $s = 0$ to $L - l$
        i. fill cell$(s, s + l)$
```

Filling a cell coresponds to finding the possible terminal and non-terminals that can span each terminal string (e.g., the third dimension) and determining their $\mu_A(i,k)$. We want to fill bottom up because filling a cell requires the entries for all of the cells beneath it. Figure 4.14 shows our chart when the grammar of Figure 4.5. Note that charts are what we earlier refered to as a *packed parse forest* in that they compress an exponential number of possible trees into a data structure of size N^2, where N is the length of the sentence.

We begin with the case where the grammar is in Chomsky normal form (CNF), i.e., all rules are terminal or binary, and consider unary rules later. Our algorithm computes the values of $\mu_A(i,k)$ from smaller substrings to larger ones. Consider the substrings of $\boldsymbol{w}$ of length 1, i.e., where $k = i + 1$.

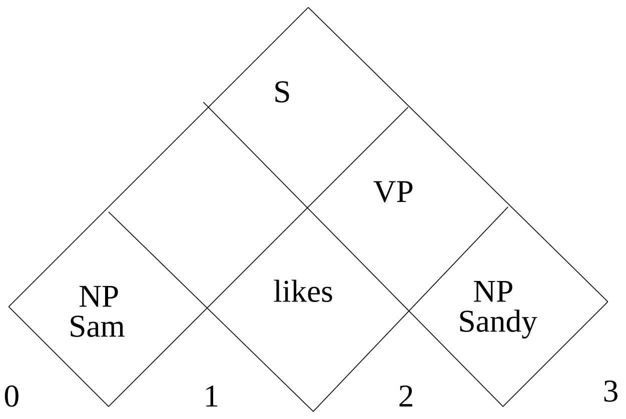

Figure 4.14: Chart appropriately filled for "Sam likes Sandy"

If the grammar is in CNF then these can only be generated by a terminal rule, as the yield of any tree containing a binary branch must contain at least two words. Therefore for $i = 0, \ldots, |\boldsymbol{w}| - 1$:

$$\mu_A(i, i+1) \quad = \quad \rho_{A \to w_i} \tag{4.3}$$

Now we turn to the substrings $\boldsymbol{w}_{i,k}$ of length greater than 1, so $k - i > 1$. All of their parse trees must start with a binary branching node, which must be generated by some rule $(A \to B\ C) \in \mathcal{R}$. This means that there must be a subtree with root labeled B and yield $\boldsymbol{w}_{i,j}$ and another subtree with root labeled C and yield $\boldsymbol{w}_{j,k}$. This yields the following equation for $\boldsymbol{\mu}$ when $k - i > 1$,

$$\mu_A(i, k) \quad = \quad \max_{\substack{j\,:\,i<j<k \\ A \to B\,C \in \mathcal{R}_A}} \quad \rho_{A \to B\ C}\ \mu_B(i, j)\ \mu_C(j, k) \tag{4.4}$$

In words, we look for the combination of rule and mid-point which gives us the maximum probability for (A) spanning i, k. These probabilities are simply the probabilities of the two sub-components times the probability of the rule that joints them to form an A.

Equations 4.3 and 4.4 can be used bottom-up (working from shorter substrings to longer) to fill in the table $\boldsymbol{\mu}$. After the table is complete, the probability of the most likely parse is $\mu_S(0, |w|)$. Finding the most likely parse is straightforward if you associate each entry $\mu_A(i, k)$ with "back pointers" to the $\mu_B(i, j)$ and $\mu_C(j, k)$ that maximize (4.4).

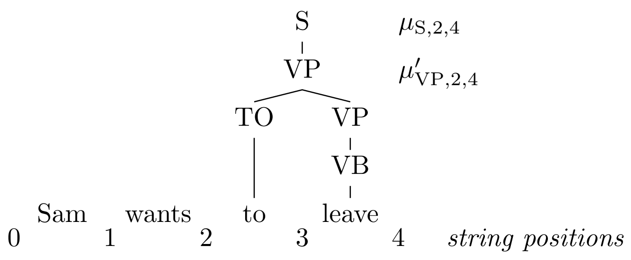

Figure 4.15: This figure shows the relationship between $\boldsymbol{\mu}'$ (which only considers trees starting with a terminal or binary rule) and $\boldsymbol{\mu}$ (which considers all trees).

Unfortunately unary rules complicate this simple picture somewhat. If we permit unary rules the subtree responsible for maximizing $\mu_A(i,k)$ can consist of a chain of up to $|\mathcal{N}|-1$ unary branching nodes before the binary or terminal rule is applied. We handle this by introducing $\mu'_A(i,k)$, which is the probability of the most likely parse rooted of $\boldsymbol{w}_{i,k}$ whose root is labeled A and expands with a binary or terminal rule. Figure 4.15 shows the relationship between $\boldsymbol{\mu}$ and $\boldsymbol{\mu}'$. Then:

$$\mu'_A(i,i+1) = \rho_{A\to w_i} \tag{4.5}$$

$$\mu'_A(i,k) = \max_{\substack{j\,:\,i<j<k \\ A\to B\,C\in\mathcal{R}_A}} \rho_{A\to B\;C}\;\mu_B(i,j)\;\mu_C(j,k) \tag{4.6}$$

$$\mu_A(i,k) = \max\left(\mu'_A(i,k),\ \max_{A\to B\in\mathcal{R}_A} \rho_{A\to B}\;\mu_B(i,k)\right) \tag{4.7}$$

Again, the computation needs to be arranged to fill in shorter substrings first. For each A, i and j combination, (4.5) or (4.6) only needs to be applied once. But unless we know that the grammar constrains the order of nonterminals in unary chains, (4.7) may have to be repeatedly applied up to $|\mathcal{N}|-1$ times for each i,j combination, sweeping over all unary rules, because each application adds one additional story to the unary chain.

There are some obvious optimizations here. For example, no nonterminal produced by binarization can ever appear in a unary chain, so they can be ignored during the construction of unary chains. If there are no unary chains of height ℓ for a particular i,j then there are no chains of height $\ell+1$, so there's no point looking for them. One can optimize still further: there's no point in considering a unary rule $A \to B$ to build a chain of height $\ell+1$

unless there is a unary chain with root B of height ℓ. And there's no need to actually maintain two separate tables $\boldsymbol{\mu'}$ and $\boldsymbol{\mu}$: one can first of all compute $\mu'_A(i,k)$ using (4.5) or (4.6), and then update those values to $\mu_A(i,k)$ using (4.7).

On the other hand, it possible that you already have, say, a constituent Z built from a chain of length two, but then find a better way to build it with a chain of height three. Even if this is the only thing added at three, you have to go on to four, because there could be some other constituent that will now use the more proable Z to improve its probability. To put it another way, you keep going until no non-terminal increases its μ.

4.4 Estimating PCFGs

This section describes methods for estimating PCFGs from data. We consider two basic problems here. First, we consider the supervised case where the training data consists of parse trees. Then we consider the unsupervised case where we are given a CFG (i.e., we are told the rules but not their probabilities) and have to estimate the rule probabilities from a set of strings generated by the CFG. (Learning the rules themselves, rather than just their probabilities, from strings alone is still very much an open research problem).

4.4.1 Estimating PCFGs from parse trees

This section considers the problem: given a sequence $\boldsymbol{t} = (t_1, \ldots, t_n)$ of parse trees, estimate the PCFG $\hat{G}$ that might have produced $\boldsymbol{t}$. The nonterminals $\mathcal{N}$, terminals $\mathcal{V}$, start symbol S and set of rules $\mathcal{R}$ used to generate $\boldsymbol{t}$ can be read directly off the local trees of $\boldsymbol{t}$, so all that remains is to estimate the rule probabilities $\boldsymbol{\rho}$. We'll use the Maximum Likelihood principle to estimate these. This supervised estimation problem is quite straight-forward because the data is fully observed, but we go through it explicitly here because it serves as the basis of the Expectation-Maximization unsupervised algorithm discussed in the next section.

It's straightforward to show that the likelihood of $\boldsymbol{t}$ is:

$$\begin{aligned} L_{\boldsymbol{t}}(\boldsymbol{\rho}) &= \mathrm{P}_{\boldsymbol{\rho}}(\boldsymbol{t}) \\ &= \prod_{A \to \beta \in \mathcal{R}} \rho_{A \to \beta}^{n_{A \to \beta}(\boldsymbol{t})} \end{aligned} \tag{4.8}$$

where $n_{A\to\beta}(\boldsymbol{t})$ is the number of times that the local tree $A \to \beta$ appears in the sequence of trees $\boldsymbol{t}$.

Since $\boldsymbol{\rho}$ satisfies the normalization constraint (4.1) on page 108, (4.8) is a product of multinomials, one for each nonterminal $A \in \mathcal{N}$. Using fairly simple analysis one can show that the maximum likelihood estimate is:

$$\hat{\rho}_{A \to \beta} \quad = \quad \frac{n_{A \to \beta}(\boldsymbol{t})}{n_A(\boldsymbol{t})} \tag{4.9}$$

where $n_A(\boldsymbol{t}) = \sum_{A \to \beta \in \mathcal{R}_A} n_{A \to \beta}(\boldsymbol{t})$ is the number of nodes labeled A in $\boldsymbol{t}$.

4.4.2 Estimating PCFGs from strings

We now turn to the much harder problem of estimating the rule probabilities $\boldsymbol{\rho}$ from strings, rather than parse trees. Now we face an estimation problem with hidden variables. As we have done with other hidden variable estimation problems, we will tackle this one using Expectation-Maximization (EM).

The basic idea is that given a training corpus $\boldsymbol{w}$ of strings and an initial estimate $\boldsymbol{\rho}^{(0)}$ of the rule probabilities, at least conceptually we use $\boldsymbol{\rho}^{(0)}$ to compute the distribution $\mathrm{P}(\boldsymbol{t}|\boldsymbol{w})$ over possible parses for $\boldsymbol{w}$, and from that distribution compute the expected value $\mathrm{E}[n_{A \to \beta}|\boldsymbol{w}]$ of the statistics needed in the MLE equation (4.9) to produce an improved estimate $\boldsymbol{\rho}$. We iterate this process, ultimately converging on at least a local maximum of the likelihood.

The key quantity we need to compute in the EM algorithm are the expected rule counts:

$$\begin{aligned}
\mathrm{E}[n_{A \to \beta} \mid \boldsymbol{w}] \quad &= \quad \sum_{\boldsymbol{t} \in \mathcal{T}_G(\boldsymbol{w})} n_{A \to \beta}(\boldsymbol{t}) \; \mathrm{P}(\boldsymbol{t} \mid \boldsymbol{w}) \\
&= \quad \frac{1}{\mathrm{P}(w)} \sum_{\boldsymbol{t} \in \mathcal{T}_G(\boldsymbol{w})} n_{A \to \beta}(\boldsymbol{t}) \; \mathrm{P}(\boldsymbol{t})
\end{aligned}$$

If the sentences in $\boldsymbol{w}$ are all very short it may be practical to enumerate all of their parses and compute these expectations this way. But in general this will not be possible, and we will need to use a more efficient algorithm. The rest of this section describes the *forward-backward algorithm*, which is a dynamic programming algorithm for computing these expectations for binarized grammars.

To keep matters simple, we will focus on computing the expected counts from a single string $\boldsymbol{w} = (w_0, \ldots, w_{n-1})$. In practice we would compute the expected counts for each sentence separately, and sum them to get the expectations for the corpus as a whole.

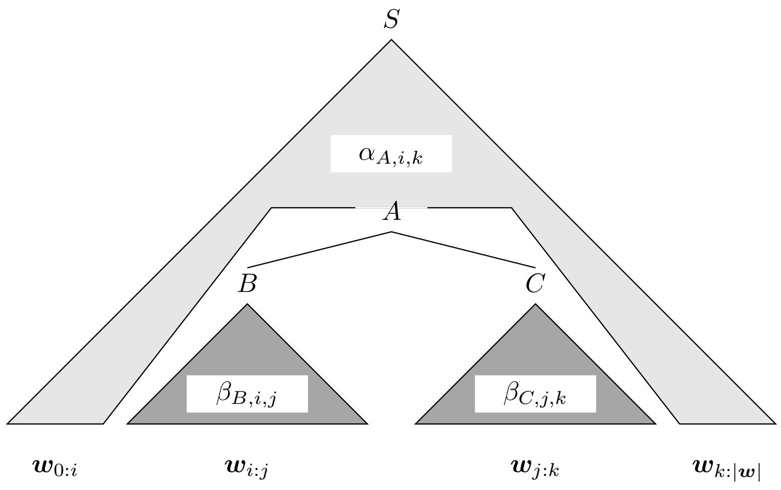

Figure 4.16: The parts of the tree contributing to the inside probabilities $\beta_A(i,j)$ and the outside scores $\alpha_A(i,j)$, as used to calculate $\mathrm{E}[n_{A \to B\,C}(i,j,k)]$.

4.4.3 The inside-outside algorithm for CNF PCFGs

Just as we did for parsing, we'll first describe the algorithm for grammars in Chomsky Normal Form (i.e., without unary rules), and then describe the extension required to handle unary rules.

Generalizing what we did for HMMs, we first describe how to compute something more specific than the expectations we require, namely the expected number $\mathrm{E}[n_{A \to BC}(i,j,k)|\boldsymbol{w}]$ of times a rule $A \to B\ C$ to expand an A spanning from i to k into a B spanning from i to j and a C spanning from j to k. The expectations we require are obtained by summing over all string positions i, j and k.

The quantities we need to compute are the *inside* and *outside* "probabilities" $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ respectively. These generalize the Backward and the Forward probabilities used in the EM algorithm for HMMs, which is why we use the same symbols for them. The scare quotes are there because in grammars with unary rules outside "probabilities" can be larger than one, and because of this we will refer to outside *scores* instead of "probabilities". Figure 4.16 depicts the parts of a tree that contribute to the inside probabilities and outside scores.

The inside probability $\beta_A(i,j)$ is the probability of an A expanding to

$\boldsymbol{w}_{i,j}$, which is the sum of the probability of all trees with root labeled A and yield $\boldsymbol{w}_{i,j}$, i.e.,

$$\begin{aligned} \beta_A(i,j) &= \mathrm{P}_A(\boldsymbol{w}_{i:j}) \\ &= \sum_{t \in \mathcal{T}_A(\boldsymbol{w}_{i:j})} \mathrm{P}(t) \end{aligned}$$

The outside socre $\alpha_A(i,j)$ is somewhat harder to understand. It is the sum of the probability of all trees whose yield is $\boldsymbol{w}_{0,i} A \boldsymbol{w}_{j:|\boldsymbol{w}|}$, i.e., the string $\boldsymbol{w}$ with $\boldsymbol{w}_{i,j}$ replaced by A. It's called the outside probability because it counts only the part of the tree outside of A. Intuitively, it is the sum of the probability of all trees generating $\boldsymbol{w}$ that include an A expanding to $\boldsymbol{w}_{i,j}$, not including the subtree dominated by A.

The reason why the outside scores can be greater than one with unary rules is that the tree fragments being summed over aren't disjoint, i.e., a tree fragment and its extension wind up being counted. Consider the grammar with rules $\mathrm{S} \rightarrow \mathrm{S}$ and $\mathrm{S} \rightarrow x$, with $\rho_{\mathrm{S}\rightarrow\mathrm{S}} = \rho_{\mathrm{S}\rightarrow x} = 0.5$. The outside trees whose probability is summed to compute $\alpha_{\mathrm{S}}(0,1)$ consist of unary chains whose nodes are all labeled S, so $\alpha_{\mathrm{S}}(0,1) = 1 + 1/2 + 1/4 + \ldots = 2$. On the other hand, $\beta_{\mathrm{S}}(0,1)$ involves summing the probability of similar unary chains which terminate in an x, so $\beta_{\mathrm{S}}(0,1) = 1/2 + 1/4 + \ldots = 1$ as expected (since 'x' is the only string this grammar generates). For this section, however we are assuming Chomsky Normal Form grammars, which do not have unay rules. This simplifies things. In particular the outside scores are now probabilities.

Since the grammar is in Chomsky Normal Form, $\boldsymbol{\beta}$ satisfies the following:

$$\begin{aligned} \beta_A(i,i+1) &= \rho_{A\rightarrow w_i} \\ \beta_A(i,k) &= \sum_{\substack{A\rightarrow B\ C \in \mathcal{R}_A \\ j\,:\,i<j<k}} \rho_{A\rightarrow B\ C}\ \beta_B(i,j)\ \beta_C(j,k) \text{ if } k-i>1 \end{aligned}$$

These can be used to compute $\boldsymbol{\beta}$ in a bottom-up fashion by iterating from smaller to larger substrings, just like we did for $\boldsymbol{\mu}$ in section 4.3 on page 114. It's easy to see that $\beta_S(0,|\boldsymbol{w}|) = \mathrm{P}(\boldsymbol{w})$, i.e., the probability of the string $\boldsymbol{w}$.

The outside probability $\boldsymbol{\alpha}$ are somewhat more complex.

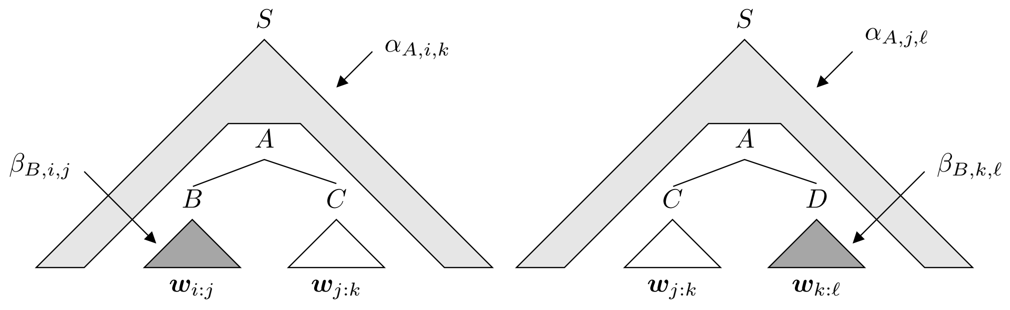

Figure 4.17: The two cases corresponding to the terms in the sum in (4.11) on page 123. Note that the insides of siblings B and D are outside C but inside A, so their inside probabilities are multiplied by the outside scores of A and the rule that connects them.

$$\alpha_A(0, |\boldsymbol{w}|) = 1 \text{ if } A = \text{S and 0 otherwise} \tag{4.10}$$

$$\alpha_C(j,k) = \sum_{\substack{A \to B\ C \in \mathcal{R} \\ i\,:\,0 \leq i < j}} \rho_{A \to B\ C}\ \alpha_A(i,k)\ \beta_B(i,j) + \sum_{\substack{A \to C\ D \in \mathcal{R} \\ \ell\,:\,k < \ell \leq |\boldsymbol{w}|}} \rho_{A \to C\ D}\ \alpha_A(j,\ell)\ \beta_D(k,\ell) \tag{4.11}$$

In order to understand these equations, it helps to recognize that if t is a tree generated by a grammar in Chomsky Normal Form then every nonterminal node in t is either the root node, and therefore has its α specified by (4.10), or is either a right child or a left child of some other nonterminal, and (4.11) sums over these alternatives. In more detail, (4.11) says that the outside score for the smaller constituent C consists of the sum of the outside scores for the larger constituent A times the inside probability of its sibling (either B or D) times the probability of the rule that connects them. Figure 4.17 depicts the structures concerned in (4.11).

These two equations can be used to compute $\boldsymbol{\alpha}$, iterating from longer strings to shorter. The first equation initializes the procedure by setting the outside probability for the root S node to 1 (all other labels have outside probability 0).

Once we have computed $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the expected counts are given by:

$$\mathrm{E}[n_{A\to w_i}(i,i+1) \mid \boldsymbol{w}] = \frac{\alpha_{A,i,i+1}\ \rho_{A\to w_i}}{\mathrm{P}(\boldsymbol{w})} \tag{4.12}$$

$$\mathrm{E}[n_{A\to B\ C}(i,j,k) \mid \boldsymbol{w}] = \frac{\alpha_{A,i,k}\ \rho_{A\to B\ C}\ \beta_{B,i,j}\ \beta_{C,j,k}}{\mathrm{P}(\boldsymbol{w})} \tag{4.13}$$

The expected rule counts that we need to reestimate $\boldsymbol{\rho}$ are obtained by summing over all combinations of string positions, i.e.,

$$\begin{aligned}\mathrm{E}[n_{A\to w} \mid \boldsymbol{w}] &= \sum_{i=0}^{|\boldsymbol{w}|-1} \mathrm{E}[n_{A\to w}(i,i+1) \mid \boldsymbol{w}] \\ \mathrm{E}[n_{A\to B\ C} \mid \boldsymbol{w}] &= \sum_{\substack{i,j,k:\\ 0\le i<j<k\le|\boldsymbol{w}|}} \mathrm{E}[n_{A\to B\ C}(i,j,k) \mid \boldsymbol{w}]\end{aligned}$$

4.4.4 The inside-outside algorithm for binarized grammars

This section sketches how the inside-outside algorithm can be extended to binarized grammars with unary rules. The key to this is calculating the scores (i.e., the product of rule probabilities) of all unary chains with root labeled A and root labeled B for all $A, B \in \mathcal{N}$.

In the computation of the most likely parse $\hat{t}(\boldsymbol{w})$ in section 4.3 on page 114 we explicitly enumerated the relevant unary chains, and were certain that the process would terminate because we could provide an upper bound on their length. But here we want to sum over all unary chains, including the low probability ones, and if the grammar permits *unary cycles* (i.e., nontrivial unary chains whose root and leaf have the same label) then there are infinitely many such chains.

Perhaps surprisingly, it turns out that there is a fairly simple way to compute the sum of the probability of all possible unary chains using matrix inversion. We start by defining a matrix $\mathbf{U} = \mathbf{U}_{A,B}$ whose indices are nonterminals $A, B \in \mathcal{N}$ of the binarized grammar G. (In order to use standard matrix software you'll need to map these to natural numbers, of course).

We set $\mathbf{U}_{A,B} = \rho_{A\to B}$ to be the probability of a unary chain of length one starting with A and ending with B. Then the probability of chains of length two is given by $\mathbf{U}^2$ (i.e., the probability of a chain of length two starting with an A and ending with a B is $\mathbf{U}^2_{A,B}$), and in general the probability of chains of length k is given by $\mathbf{U}^k$. Then the sum $\mathbf{S}$ of the probabilities of all such chains is given by:

$$\begin{aligned}\mathbf{S} &= \mathbf{1} + \mathbf{U} + \mathbf{U}^2 + \ldots \\ &= (\mathbf{1} - \mathbf{U})^{-1}\end{aligned}$$

where $\mathbf{1}$ is the identity matrix (corresponding to unary chains of length zero) and the superscript '-1' denotes matrix inversion.

With the matrix $\mathbf{S}$ in hand, we now proceed to calculate the inside probabilities $\boldsymbol{\beta}$ and outside scores $\boldsymbol{\alpha}$. Just as we did in section 4.3 on page 114, we introduce auxiliary matrices $\boldsymbol{\beta}'$ and $\boldsymbol{\alpha}'$ that only sum over trees whose root expands with a terminal or binary rule, and then compute their unary closure.

The equations for the inside probabilites are as follows:

$$\begin{aligned}\beta'_A(i,i+1) &= \rho_{A\to w_i} \\ \beta'_A(i,k) &= \sum_{\substack{A\to B\ C\in\mathcal{R}_A \\ j\,:\,i<j<k}} \rho_{A\to B\ C}\ \beta_B(i,j)\ \beta_C(j,k) \text{ if } k-i>1 \\ \beta_A(i,j) &= \sum_{B\in\mathcal{N}} \mathbf{S}_{A,B}\ \beta'_B(i,j)\end{aligned}$$

Note that the equation for $\boldsymbol{\beta}$ is *not* recursive, so it only needs to be computed once for each combination of A, i and j (unlike the corresponding unary closure equation for $\boldsymbol{\mu}$).

The equations for the outside scores are also fairly simple extensions of those for Chomsky Normal Form grammars, except that the unary chains grow *downward*, of course.

$$\begin{aligned}\alpha'_A(0,|\boldsymbol{w}|) &= 1 \text{ if } A = \mathrm{S} \text{ and } 0 \text{ otherwise} \\ \alpha'_C(j,k) &= \sum_{\substack{A\to B\ C\in\mathcal{R} \\ i\,:\,0\le i<j}} \rho_{A\to B\ C}\ \alpha_A(i,k)\ \beta_B(i,j) \\ &\quad + \sum_{\substack{A\to C\ D\in\mathcal{R} \\ \ell\,:\,k<\ell\le|\boldsymbol{w}|}} \rho_{A\to C\ D}\ \alpha_A(j,\ell)\ \beta_D(k,\ell) \\ \alpha_B(i,j) &= \sum_{A\in\mathcal{N}} \mathbf{S}_{A,B}\ \alpha'_A(i,j)\end{aligned}$$

Interestingly, the equations (4.12–4.13) we gave on page 124 are correct for binarized grammars as well, so all we need is the corresponding equation

```
(ROOT
 (SQ (MD Would)
   (NP (NNS participants))
   (VP (VP (VB work) (ADVP (JJ nearby)))
       (CC or)
       (VP (VP (VB live) (PP (IN in) (NP (NN barracks))))
           (CC and)
           (VP (VB work))
             (PP (IN on) (NP (JJ public) (NNS lands)))))
    (. ?)))
```

Figure 4.18: A correct tree as specified by the tree bank

for unary rules.

$$\mathrm{E}[n_{A\rightarrow B}(i,j) \mid \boldsymbol{w}] = \frac{\alpha_A(i,j)\ \rho_{A\rightarrow B}\ \beta_B(i,j)}{\mathrm{P}(\boldsymbol{w})} \tag{4.14}$$

4.5 Scoring Parsers

The point of a parser is to produce "correct" trees. In this section we describe how the field measures this, and ultimately assigns a single number, say, 0.9 (out of 1.0) to grade a parsers performance. Rather than do this in the abstract, however, it would be more interesting to do this for a particular parser, say one built directly using the technology we described above.

To start we first need to introduce the *Penn treebank*, a corpus of about 40,000 sentences (one million words) from the Wall-Street Journal, each assigned a phrase structure by hand. Figure 4.18 is a (slightly shortened) tree from this corpus. You might notice the start symbol is not S, but $ROOT$ because many strings in the tree bank are not complete sentences. Also notice that punctuation is included in the tree. It is a good idea to parsing with punctuation left in, as it gives good clues to the correct parse structure, and parsers uniformly work better with the punctuation than without it.

After binarizing the trees we can find Maximum Likelyhood rule probabilities as described in Section 4.4.1. We then build a basic parser as layed out above, and parse a test set we separated in advance. The question we now answer is, having done this, what is the parser's "score" and what does it mean?

```
(ROOT
 (SQ (MD Would)
   (NP (NNS participants))
   (VP (VB work)
       (ADJP (JJ nearby) (CC or) (JJ live))
        (PP (IN in)
            (NP (NP (NN barracks) (CC and) (NN work))
                (PP (IN on) (NP (JJ public) (NNS lands))))))
   (. ?)))
```

Figure 4.19: An incorrect tree for the example in Figure 4.18

The standard numerical measure of parser performance is something called *labeled precision/recall f-measure*. It is a number between zero and 1, with 1 being perfect. We now unpack this name.

First we find the total number of correct phrasal constituents in all of the test set. So in Figure 4.18 there are 13 phrasal labels (and fourteen pretermals). We do not count ROOT, because it is impossible to get wrong, so we have 12. A test-set of a thousand sentences might have a total of twenty five thousand such nodes. We then count how many the parser got correct. If it produced exactly the above tree, it would be twelve, but in fact, it produces the tree in Figure 4.19 How many nodes are correct there?

To assign a number, we first need to define what makes a constituent in one tree the "same" as one in another tree with the same yield. Our formal definition is that two trees are the same if and only if they have the same label, starting point in the string, and ending point in the string. Note that according to this definition we cound as correct the VP(2,13) found in Figures 4.18 and 4.19 even though they do not have the same sub-constituents. The idea is that in such a case the sub-constituents will be marked wrong, but we do now allow the mistakes at that level to count against the VP one level up. The required label identity of the two constituents is what makes this "*labeled* precision/recall f-measure" So here there are four correct constituents in Figure 4.19 out of eight total.

So let C be the total constituents correct, G be the number in the tree-bank tree, and H the number in the parser tree. We define *precision* (P) as C/H and *recall* (R) is C/G. F-measure(F) is the harmonic mean of the two $(2{\cdot}P{\cdot}R)/(P+R)$. Note that if $P=R$ then $P=R=F$, or equalently F is the average of P and R. But as P and R diverge, F tends to the smaller of the two. So suppose our parser produces a tree with only one phrasal category, and it is correct, then $P = 1$, $R = 1/13$, and $F \approx 0.16$

When you build the just-described parser and test it we get $C = 18151$,$G = 27007$, and $H = 25213$, so $F = .695$ This means that almost one out of three constituents are incorrect. The example of Figure 4.19 was deliberatly picked to show you how bad things can get. The parser decided to use "work" as a noun, and "live" as an adjective, and these make for unusual combinations, such as one saying, in effect, that the participants are working "nearby or live" presumably as opposed to far-away or dead.

Fortunately, this is a very simple model, and statistical parsing has moved way beyond this. A modern parser gets about .90 F, and in particular gets the above example completely correct.

4.6 Estimating better grammars from treebanks

Context-free grammars are called "context-free" because the choice of rule expanding each non-terminal is independent of everything already generated in the tree except for the label on the parent being expanded. You can think of this label as classifying the rest of the tree into one of $|\mathcal{N}|$ states. Or to put it another way, a context-free symbol is an information barrior between what is going on outside the constituent and its inside. Because a CFG only constrains the labels on nodes in local trees, if we want to capture a a longer-range dependency between nonlocal parts of the tree, it must be broken down into a sequence of local dependencies between between the labels on nodes in the same local trees. The nonterminal labels then serve as "communication channels" responsible for propagating the dependency through the appropriate sequence of local trees. This process would start with a single state such as NP, and end up with several variants, NP1, NP2, etc. Thus the process is called *state-splitting*.

The classic example of state-splitting is *parent annotation.* where the parent category of each nonterminal is encoded in its label. So, if the string is a complete sentence the parse starts out with `(ROOT (S ...))` But treebank S nodes under ROOT (almost) always end in a final punctuation mark, and thus these S's are quite different from, e.g., subordinate clases. (A *subordinate clause* is a sentence-like structure that We can better model this is we have the symbol "S^ROOT" — an S underneath ROOT. In fact there are a lot of situations where parent annotation allows us to capture important regularities. For example, subject noun phrases, e.g., $\text{NP}\hat{\text{S}}$, are more likely to be pronouns than object NPs (NP^VP). Figure 4.20 also shows the result of parent-annotating the S and NP nonterminals.

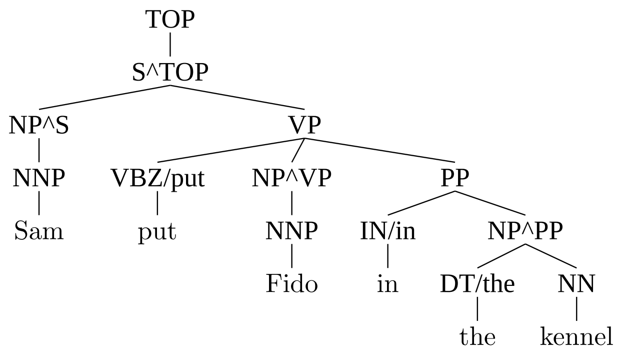

Figure 4.20: The tree produced by a state-splitting tree-transformation of the one depicted in Figure 4.11 on page 113. All verb, preposition and determiner preterminals are lexicalized, and S and NP nonterminals are parent-annotated. A new start symbol TOP has been introduced because not all trees in the treebank have a root labeled S.

Another important way to use state splitting is suggested by the fact that many words prefer to appear in phrases expanded with particular rules. For example, '*put*' prefers to appear under a VP that also contains an NP and PP (as in the example in Figure 4.11 on page 113), while '*sleep*' prefers to appear without any following NP (i.e., '*Sam slept*' is a much better sentence than '*Sam slept Fido in the kennel*', but '*Sam put Fido in the kennel*' is much better than '*Sam put*'). However, in PCFGs extracted from the Penn Treebank and similar treebanks, there is a preterminal intervening between the word and the phrase (VBZ), so the grammar cannot capture this dependency. However, we can modify the grammar so that it does by splitting the preterminals so that they encode the terminal word. This is sometimes called "lexicalization". (In order to avoid sparse data problems in estimation you might choose to do this only for a small number of high-frequency words). For grammars estimated from a treebank training corpus, one way to do this is to transform the trees from which the grammar is estimated. Figure 4.20 shows this as well.

A second way to handle sparse data problems cause by state splitting is to smooth, as we did in Chapter 1. So we might estimate the probability of a parent anotated constituent by blending it with the unannoted version. So if $R_1 = VP^{VP} \to VBDNP^{VP}$ we could smooth it with $R_2 = VP \to VBDNP$

as follows:

$$\widetilde{P}_{R_1} = \frac{n_{R_1}(\boldsymbol{d}) + \beta\, \widetilde{P}_{R_2}}{n_{VB^{VB}}(\boldsymbol{d}) + \beta}$$

4.7 Programming A Parser

Section 4.3 layed out the basics of probabilistic CFG parsing. When we left it, it looked something like this: The basic data structure is the grammar, a set of rules where a rule r is the fourtuple $< lhs, rhs1, rhs2, prob >$. The chart is an N by N array of Cells. A Cell is a set of Constituents, and a constituent is a four tuple $< label, \rho_1, \rho_2, \mu >$. A constituent coresponding to a terminal symbol would have $\mu = 1$ and $\rho_1 = \rho_x = NULL$. (No back pointers).

The parsing algorithm thus is this:

```
Function: parse($w_{0,L}$)

1. read in binarized tree-bank grammar.
2. for $\ell = 1$ to $L$
   (a) for $s = 0$ to $(L - \ell)$
       i. fill $\mathcal{C}(s, s + \ell)$
3. if $Root(0, N) \in \mathcal{C}(0, N)$
   (a) return $debinarized(\rho_{Root(0,n)})$
   (b) else return NULL

Function: fill($\mathcal{C}(i, k)$)

1. if $k = (i + 1)$
   (a) add constituent $c =< word, NULL, NULL, 1 >$ to $\mathcal{C}$
2. for $j = (i + 1)$ to $(k - 1)$
   (a) for $c_1 \in \mathcal{C}(i, j)$ and $c_2 \in \mathcal{C}(j, k)$ and rule $r =< l, lab_{c_1}, lab_{c_2}, p >$
       i. create constituent $c =< lhs, c_1, c_2, \mu_c = (p_r \cdot \mu_{c_1} \cdot \mu_{c_2}) >$
       ii. If there is no $c' \in$cell with $lab_{c'} = lhs_r$ and $\mu_{c'} \geq \mu_c$
       iii. add $c$ to $\mathcal{C}(i, k)$.
3. While adding new or higher $\mu$ constituents to $\mathcal{C}(i, k)$
```

```
(a) for $c \in \mathcal{C}$ and $r =< lhs, lab_c, NULL, p >$
    i. create constituent $n =< lhs_r, c, NULL, \mu_n = (\mu_c \cdot p_r) >$
    ii. if there is no $c' \in \mathcal{C}$ $lab_n = lab_{c'}$ and $\mu_{c'} >= \mu_n$
        A. add n to cell.
```

In words, we fill up the chart always working on smaller spans before larger ones. For each cell with a span of one we first add the word from the string. Then for larger spans we use the binarized grammar to create new constituents from pairs of smaller constituents. And for any size cell, we add constituents from unary rules repeatedly untill there are no more to add.

So far in this book we have refrained from delving below the typical level for psuedo-code. But here there is one issue that can drastically affect performance, and will be far from obvious to the reader. It has to do with the triple loop inside FILL, to the effect "for all rules, left constituents and right constituents" when you think about this you realize that these three loops can be in any order. That is, we could first find a left-constituent, then find rules that match this left constituent, and finally look for the rule's right constituent in the chart. If we find it we apply to rule to create a new constituent of the type specified by the left-hand side of the rule. But we could do this in the reverse order, first right, then rule, then left – or first left, then right, then rule. And it turns out that this can make a *big* difference in the speed of your program. In particular, do not use the ordering left, right, rule.

To see this, we first note that for a normal Chomsky normal form implementation, the average cell has about 750 constituents. This means if the two loops over constituents go on the outside, the loop in which we consider rules is executed 750·750, e.g., about a half million, times. This will slow you down.

Now at first 750 may seem excessive. The Penn-Treebank only has about thirty phrasal constituents. Where did the other 750 come from? The answer, of course, is that they are all symbols created by binarization, e.g., DT_JJ_NN. In fact, there are about fourty binarized constituents in a call for every "regular" one.

But there is some good news here. As we noted at the end of Section 4.3, the right-most symbol in a binary rule can never be a binarization non-terminal. That means there are not 750 * 750 possible combinations, to build, but only say, 750*30, where 30 is the number of original phrasal non-terminals.

If we use an ordering with rule selection in the middle we automatically get this efficiency. Suppose we start by iterating over possible left-hand side constituents. As we just noted, there will be on average about 750 of them. We now take the first of these, say with the label *DT_NN*. Next we look for rules of the form X → *DT_NNY*. Finally we look for Y's. But we are guaranteed there can be a maximum 30 of these, since no binary constituents will every appear in this position within a rule. (This is becaused we used left-branching binarization. The opposite would be true if we had used right-branching.)

That is it. It is no harder doing rule selection in the middle, than doing it last, and it is much faster.

4.8 Exercises

Exercise 4.1: Suppose we added the following rules to the grammar of Figure 4.5:

$$\text{NP} \rightarrow \text{NP CC NP} \quad \text{VBZ} \rightarrow \text{like} \quad \text{CC} \rightarrow \text{and}$$

Show the parse tree for the sentence "Sandy and Sam like the book".

Exercise 4.2: Show a dependency tree for the sentence in the previous exercise using Figure 4.3 as a template. Assume that the head of a conjoined '*NP*' is the '*CC*', and its dependents are the noun-phrase conjuncts (the subordinate noun-phrases). Do not worry about the labels on arcs.

Exercise 4.3: Fill in the chart of Figure 4.14 with (a) two constituents which are missing, and (b) the μ values for all constituents according to the probabilities of Figure 4.7. Note: there might be more constituents than cells.

Exercise 4.4: Consider the probability of a string of N words $P(w_{0,N})$ according to a PCFG. Is this equal to $\alpha_{ROOT}(0,N)$, $\beta_{ROOT}(0,N)$, both, or neither?

Exercise 4.5: Suppose the third word of a string of length $N > 3$ is "apple" where $\rho_{X\rightarrow apple}$ is 0.001 when X is NOUN, but zero for any other part of speech. Which of the following is correct? $\alpha_{NOUN}(2,3)$ is equal to $P(w_{0,N})$, it is $1000P(w_{0,N})$, it is .001, it is 1.0? What about $\beta_{NOUN}(2,3)$?

4.9 Programming Assignment

Write a parser and evaluate its accuracy. You'll use the following files:

1. `wsj2-21.blt`, which contains binarized rules and their counts extracted from sections 2–21 of the Penn WSJ treebank. All words that appear less than 5 times have been replaced with `*UNK*`. A new root node labeled `TOP` has been inserted. Also, preterminal symbols that consist entirely of punctuation characters have had a caret '^' appended to them, to distinguish them from the identical terminal symbols. You'll need to remove this from the parse trees you finally produce (`munge-trees` can do this if you want).

2. `wsj24.tree`, which contains the Penn WSJ treebank trees for section 24. Terminals not appearing in `wsj2-21.blt` have been replaced with `*UNK*`. You should use this with the EVALB scoring program to evaluate your results.

3. `wsj24.txt`, which contains the terminal strings of `wsj24.tree`.

4. The EVALB parser scoring program and instructions.

You are to parse all sentences of length ≤ 25. When you encounter a sentence of length greater than that your program should print out "*IGNORE*" to the output file. Note that the EVALB program returns various types of results. One is just the POS tag accuracy of the parser. This should be in the mid %90 range, much like an HMM tagger. The result we car about the the labeled precision recall F-measure. This will be much lower — about %70. If the output of the parser is called "output_of_parser.txt", you would munge-trees by doing "cat output_of_parser.txt — ./munge-trees -rw ¿ output_file.txt". Then to score your output, you would run "./score_parse wsj24.gold output_file.txt".

4.10 Further Reading

Speed is a concern in syntactic parsing. Chart parsing is n^3, but it is also linear in the size of the grammar. This is important because state splitting can lead to very large grammar size. Chart parsing can be mae faster with *course to fine parsingparsing!course to fine* and *best-first parsing.* Even faster speeds can be obtained in other frameworks, such as indxshift-reduce parsing*parsing!shift-reduce*, and more generally in work on .

While parsers trained with the Penn treebank work well on newspaper text, they work much less well when used on text from other domains, such as medicine. Improving performance accross domains is known as , and some techniques used in this endever are *self training* and *bootstrapping.*

As we mentioned, the reason we want to parse is that parse trees serve as a guide to combining the meansing of the parts into the meaning of the whole. One way to operationalize this is to pair grammar rules with semantic rules that operate on the pieces the syntax rules find. Probably the most well worked out version of this in computational linguistics is *generalized phrase-structure grammar* . *Unification-based garmmar* is a general term for grammars that use the unifcation algorithm to aid in connecting syntax and semantics.