One additional remark about CFGs

If we look at the pre-parsed training corpus, for the so-called Penn Treebank corpus (PTB), we can ask whether or not it “follows” the constraints on human language grammar rules talked about in the last lecture. And the answer has two parts:

1. It does not follow the condition that a phrase’s name depends on some “base” word underneath (i.e., a Verb Phrase expands somewhere as a verb). However, it does so implicitly, so we will see how when we cover parsing using the PTB.

2. It does not follow the restriction to binary-branching rules, as the small excerpt in Figure 1 shows. In particular, Noun Phrases have a “flat” structure that is not quite correct.

The reason for this is that it can be very hard to figure out the Noun-Noun modification relationships in many cases. Think about our natural language processing book example. Without some semantics to guide us, it can be very hard to determine the syntactic structure. We will look at an approach that tried to “rebracket” the PTB training examples, a bit later on in the course.

---

1 Context-free parsing: the CKY algorithm

We will now take a look at the CKY algorithm for parsing CFGs in Chomsky normal form bottom up, that is, working from the words upwards, building from small subtrees to larger ones. Later, we shall look at a top-down method, called Earley’s algorithm, that can work just as well, and, for some purposes better: for one thing, it does not need a binary branching grammar (it constructs one in real-time).

In what follows, we will use the ‘toy’ grammar of English and in Table 1 below.

Suppose we have a CFG $G$, and a sentence $s$, how can we quickly determine whether $s$ is generated by $G$, and, if so, what the parse tree for $s$ with respect to $G$ is? In this section, we will describe a bottom-up polynomial time parser for CFGs in Chomsky Normal Form that has a running time of $O(|G| \cdot n^3)$, where $n$ is the length of a sentence, and $|G|$ is the size of a grammar, as measured by the number of symbols it takes to write the grammar down. For example, in our sample grammar in Table 1, there are 41 symbols (we ignore the arrow symbol, as in Assignment 2). The algorithm presented below was first polynomial time
algorithm discovered for CFG sentence recognition and parsing, found independently by John Cocke (one day over lunch at IBM research in NY), Kasami, and Younger.

After we show how the algorithm works to recognize sentences, we briefly discuss how to convert it into a parser (an algorithm that associates one or more derivation sequences or parse trees with a sentence). In the section after that, we turn to a probabilistic version of the algorithm, which will lead us to modern day statistically-based language processing.

Preliminary notation and definitions.

\( T_G \) is the set of all possible left-most derivations (parse trees) under the grammar \( G \). When the grammar \( G \) is clear from context we will just drop the subscript and write \( T \).

For any derivation \( t \in T_G \), we write \( yield(t) \) to denote the string of words \( s \in V^* \) that is the yield of \( t \) (i.e., \( yield(t) \) is just the sequence of words in \( t \)).

For a given sentence \( s \in V^* \), we write \( T_G(s) \) to refer to the set \( \{ t : t \in T_G \text{ and } yield(t) = s \} \). That is, \( T_G \) is the set of possible parse trees for \( s \). We say that a sentence \( s \) is ambiguous if it has more than one parse tree, i.e., \( |T_G(s)| > 1 \). We say that a sentence \( s \) is grammatical if it has at least one parse tree with respect to the grammar, i.e., \( |T_G(s)| > 0 \).

We will index the word sequence for a sentence \( s \) that is \( n \) words long as follows: \( w_{0,1}, w_{1,2}, \ldots, w_{n-1,n} \). A subsequence of \( s \), e.g., the words \( i \) through \( j \) inclusively, will be notated as follows: \( w_{i,j} \). In what follows we will use the example sentence \( \text{the dog saw the man} \). For this example, the sentence would be notated as: \( 0 \text{ the} 1 \text{ dog} 2 \text{ saw} 3 \text{ the} 4 \text{ man} 5 \).

We say that the nonterminal \( X \) spans or derives the word sequence \( w_{i,j} \) if \( X \) derives \( w_{i,j} \) in one or more steps, i.e., one or more applications of a sequence of grammar rules. (This is the reflexive, transitive closure of the derives relation, \( \rightarrow \).)

Informally, \( X \) sits at the head of a subtree that terminates in the word sequence \( w_{i,j} \). In our example grammar and sentence, DT derives the first word, \( \text{dog} \), so we can say that DT spans \( w_{0,1} \). Similarly, the first NP spans the two words \( \text{the dog} \) because NP can be rewritten as DT NN, and DT and NN together span the first two words of the sentence (NP derives \( \text{the dog} \) via the application of 3 grammar rules.)

From this definition and the way that CFGs work it is easy to see that the following fact holds, which is the basic, recursive property of CFGs in Chomsky normal form that enable the CKY memorization scheme to work:

**Fact.** Given a CFG \( G \) in Chomsky normal form, if a nonterminal \( B \) spans \( w_{i,k} \), and a nonterminal \( C \) spans \( w_{k,j} \), and if there exists a rule \( A \rightarrow BC \) in \( G \), then \( A \) spans \( w_{i,j} \). (This is easily proved by induction.)

All this says is that we can ‘paste’ together a new, larger tree \( A \) out of two smaller subtrees \( B \) and \( C \) if and only if \( B \) spans the words from some point \( i \) up to the point \( k \), a left-hand subtree, and then \( C \) spans the words from \( k \) to \( j \). words, a right-hand subtree. Together, \( B \) and \( C \) put together make up \( A \), as shown

<table>
<thead>
<tr>
<th>Production</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>( V_i ) → sleeps</td>
</tr>
<tr>
<td>S → NP VB</td>
<td>( V_t ) → saw</td>
</tr>
<tr>
<td>VP → Vi</td>
<td>NN → dog</td>
</tr>
<tr>
<td>VP → Vi NP</td>
<td>NN → man</td>
</tr>
<tr>
<td>VP → VP PP</td>
<td>NN → telescope</td>
</tr>
<tr>
<td>NP → DT NN</td>
<td>DT → the</td>
</tr>
<tr>
<td>NP → NP PP</td>
<td>IN → with</td>
</tr>
<tr>
<td>PP → P NP</td>
<td>IN → in</td>
</tr>
</tbody>
</table>

Table 1: A toy grammar that is binary branching.
just below. Note that this works because of the ‘binary branching’ nature of Chomsky normal form: every terminal word has a preterminal that derives it, and aside from this, every nonterminal has exactly two daughters.

![Diagram of A, B, C with w_{ik} and w_{kj}]

For example, we can set $A = NP$, and $B = DT$ and $C = NN$, and then given the sentence *the dog saw the man*, we have following for $w_{i,k} = w_{0,1} = \text{the}$, and $w_{k,j} = w_{1,2} = \text{dog}$ and $w_{i,j} = w_{0,3} = \text{the dog}$.

The **key property** that allows us to use dynamic programming for parsing in the CKY algorithm follows from this last fact: it says that once we have computed that a nonterminal $A$ spans the positions between $i$ and $j$ in the input, then we **never** have to recompute this fact. Instead, what we do is store the result in a table, and simply look this fact up if we have to re-use it – it might serve in several parses. So for example, in our *natural language processing book* example, even though there are two possible parse trees that include the subtree (NP (NP natural) (NP language)), we need only write down that fact once: we have and NP spanning positions 0 through 2, an NP spanning 0,1; and an NP spanning 1, 2. Similarly, *processing book* is an NP that spans positions 2 through 4, but it can serve the *same role* in two different parses. This method of storing results that have been previously recomputed is a general approach to saving computing time that is known as memoization. We shall see that this is the real key to efficient CFG parsing algorithm – not whether we work bottom-up (as in CKY) or top-down, or in some other order.

In particular, this suggests what our memorization table should look like! If we have a sentence that is $n$ words long, we will use the **upper triangular portion of an array** that is $n$ rows by $n$ columns, that we will call *table*. The contents of each cell of *table* at a particular position $i, j$, so table[$i, j$], will hold the set of all nonterminals that span words $i$ through $j$. In particular, in our example, *the dog saw the man*, table[0, 2] will wind up holding the non-terminal NP, since a noun phrase spans the words *the dog*, while table[1, 2] will hold DT and table[2, 3] will hold NN. If the sentence can be generated by the grammar, then table[0, $n$] should hold S – the Start symbol, since this means that the Start symbol can generate the entire sentence. Importantly, note that a cell can (and sometimes must) hold more than one nonterminal symbol, so that in general there can be a set of nonterminal symbols in any table cell.

The CKY algorithm is a particular way to fill in the cells of *table* with the required information about nonterminals, all with respect to a particular grammar and sentence. It fills the cells of *table* as it encounters words in a bottom-up way, scanning the sentence from left to right, one word at a time. In this sense, the method is on-line – that is, it need not read all the words into the computer first and then start processing. In this respect, it is partly psychologically faithful to the way that people process sentences. (Though there are other aspects in which is not – for instance, it implicitly constructs all possible parses for, e.g., *natural language processing book*, but people do not seem to do this.)

Further, we will assume in what follows that no part of speech disambiguation has been done, though in practice, one would first run a method to reduce words back to their part of speech tags, and use those instead. In the literature, *table* is also called a well-formed substring table. For our example sentence, which is 5 words long, CKY builds a table that is 5 by 5 in size, and uses its upper triangular portion, as shown in Figure 2 on the next page. As it reads the input, CKY will place the actual words in the input sentence at the diagonal positions – Figure 2 shows the word *the* at position (1, 1). Figure 2 also shows all the table indices as used by the CKY algorithm, in the left-hand corner of each cell.

OK, with these notions in place, we can now state the CKY algorithm. We will then go through a description and a step-by-step example of this algorithm in action using our example sentence. First we shall see how to state this as a simple recognition algorithm – that is, an algorithm that will just tell us whether or not a sentence $s$ is derivable in $G$. Then we will suggest how to add backpointers to the method so that it can retrieve parse trees, similar to the Viterbi algorithm.

Here’s the algorithm. It consists of three nested loops. We assume the input to the algorithm includes a list of words, in an array $\text{words}(i)$, $i = 1 \ldots n$, and the grammar $G = (N, V, S, R)$ in Chomsky normal form (binary branching).
The outermost loop, using index $j$, moves from left-to-right, as shown by the arrow in Figure 2, working from each word column to the next. Thus $j$ runs from 1 to $n$ (in our example figure, $n = 5$).

The next loop in, using index $i$, starts from some fixed $j$ value, a particular column, and then works up the fixed column, row by row, counting down to 0. This is indicated in Figure 2 by the upward sweeping dotted arrow. In any particular column, the algorithm tries to fill the cells starting first from the cell immediately above where the part of speech tag will go, that is, row $j - 2$, and then work up the particular column down to 0. Thus, the index $i$ will range from $j - 2$ down to 0.

The innermost loop uses index $k$ and then, from a single particular cell $i, j$ tries to search simultaneously both to the left and down from that cell to find all the ways that two smaller, already constructed subtrees can be put together using the rules in $G$ to make a non-terminal name to put in $table[i, j]$. This is the operation that constructs a larger tree out of smaller parts. This is shown by the two light (red) arrows in Figure 2. Note that once the algorithm is working in a single cell like this, it must try all possible ways of splitting the span from $i$ to $j$ into 2 portions, according the binary branching format of $G$. Thus, $k$ has to range over all possible positions between $i$ and $j$, exclusively: that is, from $i + 1$ to $j - 1$. So, we call $k$ a split point index.

We fill the entries for $table[i,j]$ in a left-to-right, bottom-up fashion. Here’s the algorithm.

Given: a CFG $G = N, V, S, R$ in Chomsky normal form, and input sentence $s$.

```plaintext
function CKY-Parse(words, G) returns table
  1 for j ← 1 to length(words)
  2    do table[j−1, j] ← {A | A → w_{j−1, j} ∈ R}
  3      for i ← j−2 downto 0
  4        do for k ← i + 1 to j − 1

  5          table[i, j] ← table[i, j] ∪ {A | A → B C ∈ R &
  6            B ∈ table[i, k] &
  7            C ∈ table[k, j] }
```

Figure 2: The CKY algorithm works with the upper-triangular portion of a matrix table. Here we have indicated the indices associated with each cell in the table as used by the algorithm, and arrows to indicate the general flow of operations. There is one column in the table for each word in the sentence. The upper right corner of the table holds an entry that is meant to derive the entire sentence.
Let’s explain how this works, step by step with our example, and then figure out its running time.  

\[ j = 1, \text{word the – Initial step and filling in parts of speech: code lines 1 and 2.} \]

The first line of code is the loop that runs from one word to the next, \( j \) moving along the columns one at a time from 1 to \( j \) (here, 5). As it sets \( j \) and moves to a particular column, code line 2, the algorithm first fills in the possible parts of speech given the particular input word corresponding to that column. For example, at the very start, \( j = 1 \), and CKY looks at the first word, \textit{the}.

The code at line 2 then fills in the table entry for the ‘off-diagonal’, cell \( \text{table}[j−1,j] \), with the set of nonterminals \( A \) that could possibly derive the specific word, as given by the grammar. For example, in our sentence, since we have the rule \( DT \rightarrow \text{the} \), CKY places the left-hand side of this rule, \( DT \), in the cell \( \text{table}[0,1] \), so \( DT \) is placed in \( \text{table}[0,1] \). The two loops starting at line 3 do not execute (since \( j = 1, j−2 < 0 \)). Therefore, CKY increments \( j \) and moves on to the next column, \( j = 2 \). We show what the table looks like at this point in Figure 3.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\text{column} & 0 & 1 & 2 & 3 \\
\hline
row 0 & DT & & & \\
row 1 & the & 1,2 & 1,3 & 1,4 & 1,5 \\
row 2 & & 2,3 & 2,4 & 2,5 \\
row 3 & & & 3,4 & 3,5 \\
row 4 & & & & 4,5 \\
\hline
\end{tabular}
\caption{The CKY algorithm after processing the first word, \textit{the}, in the example sentence, with column \( j = 1 \). Table entry \( \text{table}[0,1] \) has been filled in with \( DT \) because there is a rule that expands \( DT \) as \textit{the}. There is nothing more to do, so CKY will move on to the next column, with \( j = 2 \), scanning the next word, \textit{dog}.}
\end{figure}

\[ j = 2, \text{word dog – Filling in a column and building a new tree out of two subtrees:} \]

Next, with \( j = 2 \), CKY scans up column 2 to work on each of the cells in the column in turn, starting with the word \textit{dog}. It first fills in the cell for \( \text{table}[j−1,j] \) as before, so \( \text{table}[1,2] \leftarrow NN \) since \textit{dog} is derived by the rule \( NN \rightarrow dog \). If there was more than one part of speech for \textit{dog} it would place that also in this cell; for example, perhaps \textit{dog} is a transitive verb (\( V_t \)). Then \( V_t \) would go in this cell as well. Note that this part of speech ambiguity would be quickly resolved, since there would be no rule to combine this verb with a \( DT \). However, in practice, one generally removes this part of speech ambiguity before sentence processing, as mentioned earlier.

The algorithm then scans up the second column by decrementing the counter \( i \). Note that since the parts of speech are always filled at row \( j−1 \) (here, \( j = 2 \) so \( j−1 = 1 \)), the beginning of this scan upwards always starts with the row above this, so initially, \( i = j−2 \) for this loop running up the columns, which will then decrement down to 0 (the topmost row in a column). In this case, since \( j−2 = 0 \), CKY moves up to row 0.

At this cell, \( \text{table}[0,2] \), CKY begins the executions of the last, innermost third loop of the algorithm at line 4, which attempts to find all possible split points \( k \) such that there are 2 subtrees, one corresponding to a subtree on the left starting at \( i \) and running through \( k \), and the second a subtree on the right, starting at \( k \), and running through to \( j \). The crucial point is that there must be some rule in the grammar that says
that these 2 subtrees combine to form some larger tree that spans the entire sequence of words \(w_{i,k}\) and \(w_{k,j}\) (as per our earlier fact and picture about how trees are ‘glued together’ in normal form CFGs).

Thus, the innermost loop of CKY must range over \(k\) running from \(i+1\) (the first possible split point after the ‘edge’ of any possible subtree on the left, all the way to \(j-1\) (the last possible split point in any subtree on the right). In other words, for each possible split point \(k\), CKY tries to see if it can fill in \(table[i,j]\), here \(table[0,2]\), with a nonterminal \(A\) such that \(A \rightarrow BC\), and the subtrees \(B\) and \(C\) can be found in the cells in the table already filled in, that span the range from \(I\) to \(j\). This is done in lines 5–7 of the algorithm.

In this example, since \(j−2 = 0, i=0\), and we have only a single possible value for \(k\) between \(i\) and \(j\) exclusively, that is, \(k = 1\). So, CKY looks at the entries in \(table[0,1]\) (one cell to the left of the current cell being analyzed) and \(table[1,2]\) (one cell down from the current cell). We see that \(DT\) is found in \(table[1,1]\) and \(NN\) in \(table[1,2]\), and there is in fact a rule \(NP \rightarrow DT \ NN\) in the grammar, so \(table[0,2] \leftarrow NP\). Figure 4 displays this process graphically. At this point, CKY is done with column 2, because there are no more rows above this one, with \(i = 0\).

Note that in line 5 CKY takes the union of any previous values in \(table[i,j]\) with what was there before. That is, CKY can potentially accumulate a set of nonterminals entered into a single cell, just as with ambiguous parts of speech. In this case, multiple nonterminals in a single cell always imply some parsing ambiguity.

---

![Figure 4: The CKY algorithm after processing dog in the example sentence, with column \(j = 2\). Table entry \(table[1,2]\) gets filled in with \(NN\). The algorithm then considers the row above, \(table[0,2]\), and looks to the left and below, as indicated by light (red) arrows, to see if the nonterminals in cells to the left can be put together with nonterminals in cells below. In this case, CKY places \(NP\) in \(table[0,2]\) because there is a grammar rule \(NP \rightarrow DT \ NN\), as well as \(DT\) in \(table[0,1]\) and \(NN\) in \(table[1,2]\). There is no further work for CKY to do, since there are no further split points \(k\), and no rows above row 0, so CKY will move on to the next column, \(j = 3\).

\(j = 3\), word saw – Iterating over the next column.

CKY moves on to the next column, \(j = 3\), and starts filling this column’s entries. First as usual, CKY fills in the table cell immediately above the word, \(table[j−1,j]\) for \(j = 3\), so it fills \(table[2,3]\) with any nonterminal that derives \(saw\). Therefore, \(table[2,3] \leftarrow V_i\). Next, CKY scans up column 3, starting with row \(j−2\), so \(i = 1, j = 3\), which means starting at the entry for \(table[1,3]\), eventually reaching \(table[0,3]\).

Working now on filling \(table[1,3]\), CKY enters its innermost loop to try all split points \(k\) to see if there are any cells to the left of \(table[1,3]\) along with cells below \(table[1,3]\) that it can put together to fill the current cell. Here, \(k\) runs from \(i+1\), so, from 2, up to \(j−1\), or 2, so again there is just one split point to try, \(k = 2\). In this case, CKY is looking for rules in \(G\) that put together whatever is in \(table[1,2]\), here, just
NN, with whatever is in table[2, 3], here just $V_t$. But there is no rule in $G$ that expands a nonterminal into $NN V_t$, so CKY puts nothing into table[1, 3]. (This failure is shown by light (red) dotted arrows in Figure 5 below.) This finishes the execution of the innermost $k$ loop for $i = 1$.

Therefore, still in column $j = 3$, CKY decrements the counter $i$ by 1, $i = 0$, to move up one row in this column, to the cell table[0, 3]. CKY then enters its innermost loop, to try all split points $k$, running from $k = i + 1 = 1$ to $k = j - 1 = 2$, that is, CKY considers two possible split points, to see whether some nonterminal can be placed in table[0, 3]. The first split is where the left subtree spans the input words from $i = 0$ to 1, and where the right subtree spans the words from $k = 1$ to $j = 3$. The second split is where the left subtree spans the input words from $i = 0$ to 2, and where the right subtree spans the words from $k = 2$ to $j = 3$.

Considering the first split point, CKY looks in the entries table[0, 1] and table[1, 3] to see if there is any grammar rule that can derive the nonterminals in these cells. In this case however, there is only $DT$ in table[0, 1] and there is nothing at all in table[1, 3] (the empty set), so this split fails. This failure is indicated by light (blue) dotted lines in Figure 5 below.

Turning to the second split, CKY looks at the entries in table[0, 2] and table[2, 3]. The first contains $NP$, and second contains $V_t$. But there is no rule in our grammar that combines them, and nothing can be put in table[0, 3].

In any event, with this last split, CKY is done with column 3, row 0, because there are no more rows in this column, since $i = 0$. It has finished with the loop at line 3, and so increments $j$ to 4, and scans the next word, *the*.

![Figure 5](image)

Figure 5: The CKY algorithm after reading *saw* in the example sentence, with $j = 3$. Table entry table[2, 3] gets filled in with $V_t$. The algorithm then considers the row above, row 1, and table[1, 3], and looks to the left and below, as indicated by the light (red) dotted arrows, to see if the nonterminals in cells to the left can be put together with nonterminals in cells below. However, this fails and table[1, 3] remains empty. CKY then moves up one row in the same column, to row 0, and tries two possible split points to see if it can fill table[0, 3]. The first attempt, with split $k = 1$, is shown by light (blue) dotted lines; this fails because there is nothing in cell table[1, 3]. The second attempt also fails. At this point, there are no more rows above to process, and CKY moves on to consider the next column, with word *the*.

$j = 4$, *the* – Iterating over the next column.

CKY moves on to the next column, $j = 4$, and then fills this column’s entries, starting by looking at the next word, word 4, or *the*. As before, it fills in the table cell immediately above the *the*, table[$j - 1$, $j$] for $j = 4$, so it fills table[3, 4] with any nonterminal that derives *the*. Therefore, table[3, 4] $\leftarrow DT$. Next, CKY
scans up column 4 starting with row \( j - 2 \), so \( i = 2 \), \( j = 4 \), which means starting at the entry for table\[2,4\], eventually reaching row 0 at table\[0,4\].

Working first on filling the entry at row \( i = 2 \), table\[2,4\], CKY enters its innermost loop to consider all splits \( k \) ranging from \( i + 1 = 3 \) to \( j - 1 = 3 \), so just one possible split. CKY looks at the entries in table\[2,3\], \( V_t \), and table\[3,4\], DT to see if there is a rule in \( G \) that derives DT \( V_t \). However, there is no such rule, and this cell remains empty.

Moving up to the entry at row \( i = 1 \), decrementing \( i \) as usual, CKY reaches table\[1,4\]. It enters the innermost loop to consider all splits \( k \) from \( k = i + 1 = 2 \) to \( k = j - 1 = 3 \). There are thus two possible choices for split points, as indicated by the light (blue) dashed arrows in Figure 6 below. For the first, CKY looks at the entries in table\[1,2\] and table\[2,4\], to see if there is a rule that can derive the nonterminals in both of them. However, while table\[1,2\] contains NP, the set in table\[2,4\] is empty, so this split with \( k = 2 \) fails.

The second split attempt sets \( k = 3 \), so CKY looks at the entries in table\[1,3\] and table\[3,4\]. This fails too, because table\[3,4\] is empty. This is shown by the second set of light (blue) dashed arrows in Figure 6 below.

Since this completes all the possible splits for row 1, 1, CKY decrements \( i \) by 1, setting \( i = 0 \), and considers the cell in column 4 for this row, for table\[0,4\]. Here, \( k \) ranges from \( k = i + 1 = 1 \) through \( k = j - 1 = 3 \), so there are now three possible splits to consider. However, as the blue dashed arrows indicate, all of these fail, because one or another of the possible table entries on the left or below the current cell are empty. Since \( i = 0 \), CKY is now done with row \( j = 4 \), and can move to the next, and last column, \( j = 5 \), word man.

![Figure 6: The CKY algorithm after reading the in the example sentence, with j = 4. Table entry table[3,4] gets filled in with DT. The algorithm then considers the row above, row 2, and table[2,4], and looks to the left and below, as indicated by red dotted arrows, to see if the nonterminals in cells to the left can be put together with nonterminals in cells below. However, this fails and table[2,4] remains empty. CKY then moves up one row in the same column, to row 1, and tries two possible split points to see if it can fill table[1,4]. The first attempt, with split \( k = 2 \), is shown by blue dotted lines; this fails because there is nothing in cell table[1,3]. The second attempt, with split \( k = 2 \), also fails. CKY then moves up to the last row in this column, and must try three possible splits to see if it can fill table[0,4]. But all 3 splits fail, as shown by blue dotted lines, because one or another cell is empty. CKY moves on to the next column, \( j = 5 \), and the word man.](image)

\( j = 5 \), word man – Iterating over the next, and final column.

With \( j = 5 \), CKY now processes the final word, man, filling in table\[4,5\] with the nonterminal NN. CKY
then scans up column 5, starting with row \( i = j - 2 = 3 \), so \( \text{table}[3,5] \). Entering CKY’s inner loop, we have \( k = i + 1 = 4 \), iterating to \( k = j - 1 = 4 \), so just one split, with \( k = 4 \). Looking to the left and below \( \text{table}[3,5] \), we have \( DT \) in \( \text{table}[3,4] \) and \( NN \) in \( \text{table}[4,5] \), and there is a rule \( NP \to DT \ NN \), so \( \text{table}[3,5] \to NP \). (The man forms an \( NP \).) This completes row 3 in column 5.

Moving up to row 2, \( i = 2 \), CKY considers \( \text{table}[2,5] \). Entering its inner loop, there are two split points \( k \) to try, from \( k = i + 1 = 3 \) to \( k = j - 1 = 4 \). For the first split point, \( k = 3 \), CKY looks at \( \text{table}[2,3] \) and \( \text{table}[3,5] \). The first cell holds \( V_t \) (the verb), and the second holds an \( NP \), and there is a rule \( VP \to V_t NP \), so \( \text{table}[2,5] \to VP \), as shown by the solid (red) arrows in Figure 7 below. For the other split, \( k = 4 \), \( \text{table}[2,4] \) is empty, so CKY cannot fill in anything more to \( \text{table}[2,5] \), as indicated by the dotted blue lines.

Next, moving up to row 1, CKY looks at \( \text{table}[1,5] \) to see if this can be filled. Now there are three possible split points. However, all these attempts fail, because either to the left or below, there are table cells that are empty. Green dotted lines in Figure 7 show the failed attempts. Figure 7 below shows the action up to this point, just before CKY has to move up to the final row, row 0, and consider the entries for \( \text{table}[0,5] \). (We will do the last step using a separate figure since the lines are so cluttered.)

Figure 7: The CKY algorithm after reading man in the example sentence, with \( j = 5 \). Table entry \( \text{table}[4,5] \) gets filled in with \( NN \). The algorithm then considers the row above, row 3, and \( \text{table}[3,5] \), and looks to the left and below, as indicated by red arrows, to see if the nonterminals in cells to the left can be put together with nonterminals in cell below. This succeeds, and an \( NP \) is placed in \( \text{table}[3,5] \). Moving up to the next row, CKY fills in \( VP \) into \( \text{table}[2,5] \) because \( V - t \) is in \( \text{table}[2,3] \) and \( NP \) is in \( \text{table}[3,5] \). Moving up one row, with \( i = 1 \), CKY tries to fill in \( \text{table}[1,5] \), but all attempts here fail, and CKY moves on to row 0 (as shown in the next Figure).

Turning to the last row in column \( j = 5 \), row \( i = 0 \), CKY attempts to fill \( \text{table}[0,5] \), so must try all possible split points \( k \), and now there are 4 of these, as \( k \) runs from \( i + 1 = 1 \) to \( k = j - 1 = 4 \). The first split, looking at \( \text{table}[0,1] \) and \( \text{table}[1,5] \), fails, because \( \text{table}[1,5] \) is empty. But the next split, \( k = 2 \), succeeds, because \( \text{table}[0,2] \) holds \( NP \), \( \text{table}[2,5] \) holds \( VP \), and we have the rule \( S \to NP VP \). Consequently, CKY can put \( S \) into \( \text{table}[0,5] \), indicated by the solid red lines in Figure 8 on the next page. (This marks the successful completion of the sentence analysis, but CKY does not stop here since there could be more than one analysis, that is, more than one parse.) Continuing for the last 2 possible splits, shown as dotted blue lines in the Figure below. These both fail, because either there are no grammar rules that put the two pieces together (eg, \( S \) and \( NP \)), or else a table cell is empty. CKY is then done. Whew.
Figure 8: The CKY algorithm after reading man in the example sentence, with \( j = 5 \), and working on the final, topmost row, 0, so the entries for table\([0, 5]\). To compute how to fill this cell, CKY considers all the split points between 0 and 5, exclusively, that is, \( k \) runs from 1 to 4. Looking to the left and below this cell, the dotted lines indicate failed possible combinations, while there is one combination where the split point \( k = 2 \) succeeds: here, an NP is found in table\([0, 2]\), corresponding the the dog, and a VP is in table\([2, 5]\), corresponding to saw the man. So, CKY can put S in table\([0, 5]\), which indicates a successful parse. (In general, CKY must try to find all possible successful parses, so it cannot stop after finding the first one, but must run to completion, trying other \( k \) values past 2.)

### Running time of the algorithm and handling ambiguity

How much time does CKY take in the worse case, and how does it handle the possibility of ambiguity? Looking at the algorithm, it is easy to see that there are three nested loops, each of which can run at worst from 0 to \( n \), for a sentence \( n \) words long (\( j, i \), and \( k \)). Together then, these steps take \( O(n^3) \) time.

In addition, steps 5-7 require more careful analysis. We have to assume that the union operation in step 5 can be carried out in constant time, not dependent on \( n \). If one uses bit vectors and hash tables, this can usually be done. Further, we must consider the largest number of nonterminals that might ever be found in the two cells CKY looks at when testing whether a split point works. In the worst case, there could be \(|G|\) distinct nonterminals in either of the two table entries (this is a very generous over-estimate, if we set as \(|G|\) the total number of symbols in a CFG grammar \( G \)). Given all these assumptions, we can say that steps 5-7 will take at worst \( O(|G|) \) time. Thus, in all we can say that CKY takes \( O(n^3\cdot|G|) \) time. Note that the grammar size is an important component here – this often dominates running time, so the smaller the grammar the better. Note that since the table itself is of size \( O(|G|^2) \), then our algorithm must use at least this much space.

What about ambiguity? What we have provided is only a recognition algorithm, i.e., it just says whether or not a sentence can be generated by a grammar, but it does not say what the parses are. In order to recover parses, we must encode additional information in the cells: backpointers, as in Viterbi, that point back to the two nonterminals in a split that prompt the entry of some new nonterminal. That is, whenever CKY adds a new nonterminal to table\([i, j]\), the algorithm must place in this table a pointer to the table entries from which it was derived, table\([i, k]\) and table\([k, j]\). Further, in order to accommodate ambiguity, it can be the case that the same nonterminal in a particular cell would have to be listed more than once. To see this, note that if we have a sentence that has multiple parses, then the final entry for the sentence nonterminal in table\([0, n]\] must contain more than one instance for the start symbol (call it S). Each of these
entries must have different backpointers from each other, because they tell us that the $S$ was put together in different ways. This alters the running time of the algorithm, since that means we have to keep distinct what would otherwise be the same nonterminal name in a particular table cell. (You might want to figure out how much the running time is increased by this.)

To recover a particular parse, one could then trace through any sequence of backpointers, starting with all $S$ entries in $table[0, n]$. Note that if one had to recover all possible parses, then this would entail an exponential amount of time again, since we have shown that there can be an exponential number of parses for some sentences and grammars, and writing down an exponential number of parses must take at least exponential time.

To see an example of what such multiple ambiguity would look like, consider our simple 2 rule grammar to parse natural language processing. With 3 words, the filled-out CKY table for this will be 3 x 3, as shown in Figure 9. Note that the upper-right corner, $table[0,3]$, is filled with $NP$, as we could expect. Further, $table[0,1]$, $table[1,2]$, and $table[2,3]$ are all filled with $NPs$, corresponding to the rule that derives each of the single words. Red lines indicate how each word pair natural language and then language processing, form $NPs$ that are entered in $table[0,2]$ and $table[1,3]$. Finally, the $NP$ in $table[0,3]$ shows two possible ways in which it can cover two $NPs$ below it, corresponding to the two different ways for parsing natural language processing. In this case then, CKY would have to record two entries for $NP$ in this last table cell: one that has backpointers to cells 0,1 and 1,3 (the first set of blue lines), and the other that has backpointers to cells 0,2 and 2,3 (the second set of blue lines).

If we then added another word, natural language processing book, the resulting 4 by 4 table would have an additional “layer”. Now there are five possible parses. Figure 10 indicates four of them; can you find the fifth, missing parse? Try to draw in the backpointer lines this “missing” parse corresponds to.

![Figure 9: How the CKY method can use backpointers to encode possible parses, for the example, natural language processing. There are 2 parses, so the cell for $table[0, 2]$ must have 2 entries. Retrieval of all parses would take more than polynomial time as $n$ increases. Usually, we are just interested in the “best” parse, however.](image)
Figure 10: The parses for *natural language processing book*; in all there should be five distinct parses. We have left out one parse for you to find.

2. Probabilistic Context Free Grammars (PCFGs)

In part to deal with this problem of ambiguity and too many parses, one can turn to making the rules in a CFG probabilistic. In this section, we will defined PCFGs, and see how to estimate their probabilities from training data. We will then see how to alter CKY so it can use such rules. Finally, we shall see why PCFGs themselves are not enough to deal with all the problems of ambiguity in natural language.

2.1 Definitions

The key idea in probabilistic context-free grammars is to extend our definition to give a probability distribution over possible derivations. That is, we will find a way to define a distribution over parse trees, \( p(t) \), such that for any \( t \in T_G \),

\[
\forall t \in T_G, \quad p(t) \geq 0
\]

and as usual to get a valid probability distribution,

\[
\sum_{t \in T_G} p(t) = 1
\]

This seems tricky because each parse tree \( t \) is a complex structure, and the set \( T_G \) will most likely be infinite. However, we will see that there is a very simple extension to context-free grammars that allows us to define a function \( p(t) \). Once we have this distribution, then we have a ranking over possible parses for any sentence in order of probability. In particular, given a sentence \( s \), we can return as the output from our parser the most likely parse tree for \( s \) given the grammar and the probability model:

\[
\arg\max_{t \in T_G(s)} p(t)
\]

If our probability distribution is a good one for the probability of different parse trees, this will deal with ambiguity. So, we now have the questions:
1. How to define $p(t)$
2. How to estimate the parameters of the model of $p(t)$ from training data
3. For a given sentence $s$, who to find the most likely tree (a per above)

2.2 Definition of PCFGs

Definition (PCFGs) A PCFG consists of:

1. A context-free grammar $G = (N, V, S, R)$
2. A parameter $q(\alpha \rightarrow \beta)$ for each rule $\alpha \rightarrow \beta \in R$. The parameter can be interpreted as the conditional probability of choosing a rule $\alpha \rightarrow \beta$ in a left-most derivation, given that the non-terminal being expanded is $\alpha$. For any $X \in N$, we have the constraint that
   \[ \sum_{\alpha \rightarrow \beta \in R: \alpha = X} q(\alpha \rightarrow \beta) = 1 \]

   Also, $q(\alpha \rightarrow \beta) \geq 0$ for any $\alpha \rightarrow \beta \in R$

   Given a parse tree $t \in \mathcal{T}_G$ containing rules $\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n$, the probability of $t$ under the PCFG is:
   \[ p(t) = \prod_{i=1}^{n} q(\alpha_i \rightarrow \beta_i) \]

   To give an example of a PCFG, we can add probabilities (the $q$ parameters) to each rule. Each of these is non-negative, and we must have the constraint that for any nonterminal $X \in N$,
   \[ \sum_{\alpha \rightarrow \beta \in R: \alpha = X} q(\alpha \rightarrow \beta) = 1 \]

   So for any nonterminal $X$, the parameter values for all rules with that nonterminal on the left-hand side of a rule must sum to 1. See the PCFG example below, Figure 11.

   \[
   \begin{array}{c|c|c}
   S & \rightarrow & NP \ VP \quad 1.0 \\
   VP & \rightarrow & V_i \quad 0.4 \\
   VP & \rightarrow & V_i \ NP \quad 0.4 \\
   VP & \rightarrow & VP \ PP \quad 0.2 \\
   NP & \rightarrow & DT \ NN \quad 0.3 \\
   NP & \rightarrow & NP \ PP \quad 0.7 \\
   PP & \rightarrow & IN \ NP \quad 1.0 \\
   \end{array}
   \]

   \[
   \begin{array}{c|c}
   V_i & \rightarrow & sleeps \quad 1.0 \\
   V_t & \rightarrow & saw \quad 1.0 \\
   NN & \rightarrow & dog \quad 0.3 \\
   NN & \rightarrow & man \quad 0.6 \\
   NN & \rightarrow & telescope \quad 0.1 \\
   DT & \rightarrow & the \quad 1.0 \\
   IN & \rightarrow & with \quad 0.5 \\
   IN & \rightarrow & in \quad 0.5 \\
   \end{array}
   \]

   Figure 11: An example PCFG.

   Note for instance that this constraint holds for $X = VP$, since the sum of the probabilities for all the $q$ values associated for expanding $VP$ work out to $0.4 + 0.4 + 0.2 = 1$. To calculate the probability of any parse tree $t$, we multiply together the $q$ values for the context-free rules that it has. For example, if the parse tree $t$ looks like the following:

```
S
  └── NP
    └── VP
        └── V
            └── sleeps
```

13
then we just multiply the individual rules out:

\[
p(t) = q(S \rightarrow NP \ VP) \times q(NP \rightarrow DT \ NN) \times q(DT \rightarrow \text{the}) \times q(NN \rightarrow \text{dog}) \times \\
q(VP \rightarrow V_i) \times q(V_i \rightarrow \text{sleeps})
\]

In short, we have added probabilities to left-most derivations. The probability of a whole tree is the product of the probabilities for the individual selections of rules to generate a terminal string.

2.3 Deriving a PCFG from a training corpus

Having defined PCFGs, the next question is the following: how do we derive a PCFG from a corpus? We will assume a set of training data, which is simply a set of parse trees \( t_1, t_2, \ldots, t_m \). We write \( \text{yield}(t_i) \) to be the yield for the \( i \)th parse tree in the sentence, i.e., \( \text{yield}(t_i) \) is the \( i \)th sentence in the corpus. Each parse tree \( t_i \) is a sequence of context-free rules: we assume that every parse tree in our corpus has the same symbol, \( \text{Start} \), at its root. We can then define a PCFG \((N, V, \text{Start}, R, q)\) as follows:

- \( N \) is the set of all nonterminals seen in the trees \( t_1 \ldots t_m \).
- \( V^* \) is the set of all words seen in the trees \( t_1 \ldots t_m \).
- The start symbol is taken to be \( S \).
- The set of rules \( R \) is taken to be the set of all rules \( \alpha \rightarrow \beta \) seen in the trees \( t_1 \ldots t_m \).
- The maximum-likelihood parameter estimates are:
  \[
  q_{ML}(\alpha \rightarrow \beta) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}
  \]
  where \( \text{Count}(\alpha \rightarrow \beta) \) is the number of times that the rule \( \alpha \rightarrow \beta \) has been seen in the trees \( t_1 \ldots t_m \), and \( \text{Count}(\alpha) \) is the number of times the nonterminal \( \alpha \) has been seen in the training corpus \( t_1 \ldots t_m \).

So, if the rule \( VP \rightarrow V_i \ NP \) is encountered 100 times in the corpus, and the nonterminal \( VP \) 1000 times, then the parameter estimate for \( q(\text{NP} \rightarrow V_i \ NP) \) would be 100/1000.

2.4 Parsing with PCFGs

To solve for the highest scoring (maximum probability) parse tree for a sentence \( s \) with respect to a PCFG, we have to solve the problem:

\[
\arg\max_{t} p(t) \quad t \in T_G(s)
\]

We can modify the CKY algorithm to do this, using the same dynamic programming idea. Instead of keeping track of nonterminals in the CKY table, we will simply record the maximum probability for a particular derivation seen so far. To see the modification, we first repeat the listing for the ordinary CKY algorithm, and then follow it with the probabilistic version. You will see that we have to just add 3 new statements that compute the probability of a nonterminal \( A \) once we find that we can construct \( A \) from the two subtrees \( B \) and \( C \) as in the ordinary CKY algorithm. In addition, we will have to add an extra element to \text{table} \( i, j \) that keeps track of which nonterminal we are talking about, since probabilities can be different for different nonterminals. (Eg, even if both NP and PP derive the same words, it may be that the NP derivation is more likely.) So, now \text{table}[i,j] \ will become \text{table}[i,j,A] \, where \( A \) ranges over nonterminals. For keeping track of maximum probabilities, all we have to recognize is that the probability of the probability of a (subtree) \( A \) is simply the product of three terms: (1) the probability of the rule \( A \rightarrow B \ C \); (2) the probability of subtree \( B \); and (3) the probability of subtree \( C \). If this product is larger than the probability that is already stored in the entry for \text{table}[i,j,A], then we replace the old entry with the new one. In
addition, the last line of the code adds a backpointer (now nearly exactly like Viterbi), that stores the path through the derivation for just one derivation, the maximum probability one. (In practice, we will want to keep more than just the highest scoring parse, since others may be worthwhile to look at.)

In short, the modification to the original CKY algorithm means that when we go to fill in a particular table cell, we not only have to consider possible split points, we also have a choice of nonterminals to store in the cell – we only want to keep the one that has the largest possibility probability given that we have already found the maximum probability for the two subtrees that make it up. So for example, where before we might have found a VP that derives the words saw the man, what we now find is the highest scoring (maximum probability) score for any tree with root VP that covers those same words.

Original:

```
function CKY-Parse(words, G) returns table
1  for j ← 1 to length(words)
2     do table[j − 1, j] ← \{A|A → w_{j−1,j} ∈ R\}
3         for i ← j − 2 downto 0
4             do for k ← i + 1 to j - 1
5                 table[i, j] ← table[i, j] ∪ \{A|A → B ∈ R & B ∈ table[i, k] & C ∈ table[k, j]\}

Probabilistic version:

function CKY-Parse(words, G) returns most probable parse and its probability
1  for j ← 1 to length(words)
2     do table[j − 1, j, A] ← \{A|A → w_{j−1,j} ∈ R\}
3         for i ← j − 2 downto 0
4             do for k ← i + 1 to j - 1
5                 do forall \{A|A → BC ∈ R, and table[i, k, B] > 0 and table[k, j, C] > 0\}
6                     if (table[i, j, A] < P(A → BC) × table[i, k, B] × table[k, j, C])
7                         then table[i, j, A] ← P(A → BC) × table[i, k, B] × table[k, j, C]
8                             back[i, j, A] ← \{k, B, C\}
9                             return Build-Tree(back[1, length(words), S]), table[1, length(words), S]
```

As an example of how this can resolve ambiguities, suppose we have a longer sentence with a Prepositional Phrase (PP) at the end, e.g., the dog saw the man with the telescope. You may recall that this sentence has (at least) 2 possible parses, corresponding to different choices for where the PP goes: does it modify the verb phrase (VP), in which case, the derivation must have uses the rules VP→VP PP and NP→DT NN, or does the PP modify the noun phrase (NP), the man, in which case the derivation must have used the rules VP→VtNP and NP→NP PP instead?

Notice first that in either case, both derivations use the rules NP→DTNN and NP→DT N, so the choice comes down to which rule has the higher probability: VP→VP PP (with probability 0.2 out of the VP rules) or NP→NP PP (with probability 0.7 out of the NP rules). Therefore, the maximum scoring probability will assign the most likely parse to the second analysis, where the PP modifies the NP. (This is sometimes called “low attachment” or “close attachment”.)

However, this does not solve all our problems, because we can only force the parse to prefer low attachment. Yet, it is not clear that low attachment is always to be preferred. The main insight is that if we use the product of the probabilities of individual rules, as in this model, then we are assuming that the rules are
independent of one another. But is this true? Doesn’t the previous history of the rules applied affect the probabilities of rules later on? Indeed. Further, PCFGs don’t make use of the properties of individual words, either (because the decision to expand one word vs. another is made entirely locally, when the preterminal is expanded).

As an example of where this fails us, consider a (partial) sentence such as, *workers dumped sacks into a bin*. This is another case where the PP attaches ambiguously to either a VP (modifying where the sacks were dumped), or an NP (modifying the sacks). Without writing down a PCFG, we can just exhibit two parse trees for these two analyses:

(a) S
   NP
     NNS
     workers
   VP
     VBD
     dumped
     NNS
     sacks
   PP
     IN
     into
     DT
     a
     NN
     bin

(b) S
   NP
     NNS
     workers
   VP
     VBD
     dumped
     NNS
     sacks
   PP
     IN
     into
     DT
     a
     NN
     bin

Let’s now compare the two sets of rules implied by these derivations. We see that if \( q(\text{NP} \rightarrow \text{NP PP}) \) \( > q(\text{NP} \rightarrow \text{VP PP}) \) then (b) is more probable, else (a) is more probable:

(a) Rules

<table>
<thead>
<tr>
<th>S (\rightarrow) NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP (\rightarrow) NNS</td>
</tr>
<tr>
<td>VP (\rightarrow) VBD NP</td>
</tr>
<tr>
<td>NP (\rightarrow) NNS</td>
</tr>
<tr>
<td>PP (\rightarrow) IN NP</td>
</tr>
<tr>
<td>NP (\rightarrow) DT NN</td>
</tr>
<tr>
<td>NNS (\rightarrow) workers</td>
</tr>
<tr>
<td>VBD (\rightarrow) dumped</td>
</tr>
<tr>
<td>NNS (\rightarrow) sacks</td>
</tr>
<tr>
<td>INS (\rightarrow) into</td>
</tr>
<tr>
<td>DT (\rightarrow) a</td>
</tr>
<tr>
<td>NN (\rightarrow) bin</td>
</tr>
</tbody>
</table>

(b) Rules

<table>
<thead>
<tr>
<th>S (\rightarrow) NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP (\rightarrow) NNS</td>
</tr>
<tr>
<td>NP (\rightarrow) NP PP</td>
</tr>
<tr>
<td>VP (\rightarrow) VBD NP</td>
</tr>
<tr>
<td>NP (\rightarrow) NNS</td>
</tr>
<tr>
<td>PP (\rightarrow) IN NP</td>
</tr>
<tr>
<td>NP (\rightarrow) DT NN</td>
</tr>
<tr>
<td>NNS (\rightarrow) workers</td>
</tr>
<tr>
<td>VBD (\rightarrow) dumped</td>
</tr>
<tr>
<td>NNS (\rightarrow) sacks</td>
</tr>
<tr>
<td>INS (\rightarrow) into</td>
</tr>
<tr>
<td>DT (\rightarrow) a</td>
</tr>
<tr>
<td>NN (\rightarrow) bin</td>
</tr>
</tbody>
</table>

But this cannot be correct! What is wrong with this idea? Answer: the decision is made completely independently of the words – whether it is more likely that dumping occurs at a location, as opposed to being a property of sacks. That is wrong. How can we fix this? The answer will be to apply a little bit of linguistic theory (the bit that was first thought of in 1965, but not used until 1995 in NLP.)

And there are other cases of ambiguity that PCFGs won’t fix. For example, consider the sentence, *President of a company in Cambridge*. The two obvious parses both have the same rules that are used, and so therefore must receive the same probability under any PCFG analysis. However, in the PTB, the form (a) is twice as common. How can we capture this kind of information?
We will see that problems of this sort led to two developments in statistical parsing: one, the use of particular word (lexical) information – actually the view from linguistic theory around 1970, finally imported into parsing models by 1995; and second, the use of other kinds of parsing context, rather than just independent probabilities attached to single rules.