Chapter 5

Syntax

Syntax, in linguistics, is the study of the structure of natural language sentences. It is sometimes approached as an “autonomous” part of language, describing what sentences are or aren’t grammatical, and sometimes approached as working together with semantics, describing the structure which is used to compute the meaning of a sentence.

5.1 Why Syntax?

5.1.1 Examples of ambiguity

Syntax proposes that there is something more to a sentence than just the sequence of words. To get an idea of what that something is, consider the following example sentences, taken from newspaper articles.¹ What exactly is going wrong in these sentences, and how can we hope to make computers understand them?

(5.1) Two cars were reported stolen by the Groveton police yesterday.

(5.2) Mrs. Consigny was living alone in her home in Nakoma after her husband died in 1954 when the phone rang.

(5.3) Black Panther leader Huey Newton, terming a 1974 murder charge “strictly a fabrication,” said yesterday he will testify at his trial on charges of killing a prostitute against his lawyer’s advice.

(5.4) Yoko Ono will talk about her husband John Lennon who was killed in an interview with Barbara Walters.

5.1.2 A simpler example

Let’s focus on one example: Imagine that we want a computer to turn statements into questions. How would you write a program to do this?

(5.5)  a. The Pope is Catholic.
       b. Is the Pope Catholic?

¹http://www.ling.upenn.edu/~beatrice/humor/newspaper-screwups.html
5.6  a. Soylent Green is people.
    b. Is Soylent Green people?

5.7  a. Good fences make good neighbors.
    b. Make good fences good neighbors?
    c. Do good fences make good neighbors?

5.8  a. A man who is his own lawyer has a fool for a client.
    b. Is a man who his own lawyer has a fool for a client?
    c. Does a man who is his own lawyer have a fool for a client?

5.1.3 Tests for constituency

To formulate a rule to do the above, we impose a tree structure on sentences, in which the leaves of the tree are words. (There’s an alternative kind of structure, called dependency trees, in which the internal nodes are also words.) The substring spanned by a node of the tree is called a constituent, and since we can’t see constituents, linguists have developed various tests for constituency:

Can you move it around?

5.9  Fences make, good good neighbors.
5.10 Make good, good fences neighbors.
5.11 Good neighbors, good fences make.
5.12 Fences make good, good neighbors.
5.13 Make good neighbors, good fences.

Can you replace it with something like a pronoun?

5.14 They make good neighbors.
5.15 Good they good neighbors.
5.16 Good fences it/they/... neighbors.
5.17 Good fences make them.
5.18 They good neighbors.
5.19 Good it/they/... neighbors.
5.20 Good fences do.

Can it participate in a cleft?

5.21 It’s good fences that make good neighbors.
5.22 It’s fences make that good good neighbors.
5.23 It’s make good that good fences neighbors.
5.2 Context Free Grammars

5.2.1 Why, what’s wrong with finite automata?

We have spent a long time talking about all the things that finite automata can do, but there are important things that they can’t do. For example, they cannot generate the language

\[ L = \{ a^n b^n \mid n \geq 0 \} \]  

(5.33)

Linguistically, the analogous property that finite automata are missing is the ability to do center-embedding. English allows sentences like:

(5.34) The motorcycle rusted.

(5.35) The motorcycle that the guy rode rusted.

(5.36) The motorcycle that the guy that my sister married rode rusted.

At minimum, we need the number of noun phrases to equal the number of verbs, which we have already seen finite automata aren’t able to do. Actually, we want to be able to create a structure like
which can tell us which noun corresponds to which verb. Later, we will see how this structure is also useful for translating into another language like Japanese or Hindi.

**Question 1.** The above argument holds only if you believe that *unbounded* center embedding is possible. In fact, center-embedding examples degrade rather quickly as more levels are added:

\[
(5.38) \quad \text{The motorcycle that the guy that the sister that my mom spoiled married rode rusted.}
\]

If center embedding is bounded, then how would you write a finite automaton to model it? What would still be unsatisfactory about such an account?

### 5.2.2 Context free grammars

Our solution is to use *context free grammars (CFGs)*. CFGs are also widely used in compilers, where they are known as *Backus-Naur Form*. We begin with two examples of CFGs. The first one generates the non-finite-state language \{a^n b^n \mid n \geq 0\}:

\[
S \rightarrow aSb \\
S \rightarrow \epsilon
\]

Here, \(S\) is called a *nonterminal symbol* and can be rewritten using one of the above rules, whereas \(a\) and \(b\) are called *terminal symbols* and cannot be rewritten. This is how the grammar works: start with a single \(S\), then repeatedly choose the
leftmost nonterminal and rewrite it using one of the rules until there are no more nonterminals. For example:

\[
\begin{align*}
S & \Rightarrow aSb \\
& \Rightarrow aaSbb \\
& \Rightarrow aaaSbbb \\
& \Rightarrow aaabbb \\
\end{align*}
\]

Our next example generates sentences (6.2), (6.3), and (6.4).

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow DT \ NN \\
NP & \rightarrow PRP\$ \ NN \\
NP & \rightarrow NP \ SBAR \\
VP & \rightarrow VBD \\
SBAR & \rightarrow IN \ S \\
DT & \rightarrow \text{the} \\
PRP\$ & \rightarrow \text{my} \\
NN & \rightarrow \text{motorcycle} | \text{guy} | \text{sister} \\
VBD & \rightarrow \text{married} | \text{rode} | \text{rusted} \\
IN & \rightarrow \text{that} \\
\end{align*}
\]

Here, the uppercase symbols are nonterminal symbols, and the English words are terminal symbols. Also, we have used some shorthand: \( A \rightarrow \beta_1 \ | \ \beta_2 \) stands for two rules, \( A \rightarrow \beta_1 \) and \( A \rightarrow \beta_2 \).

**Question 2.** How would you use the above grammar to derive sentence (6.3)?

Here's a more formal definition of CFGs.

**Definition 1.** A context-free grammar is a tuple \( G = (N, \Sigma, R, S) \), where
• $N$ is a set of nonterminal symbols
• $\Sigma$ is a set of terminal symbols
• $R$ is a set of rules or productions of the form $A \rightarrow \beta$, where $A \in N$ and $\beta \in (N \cup \Sigma)^*$
• $S \in N$ is a distinguished start symbol

If $A \in N$ and $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, we write $\alpha Ay \Rightarrow_G \alpha \beta \gamma$ iff $(A \rightarrow \beta) \in R$, and we write $\Rightarrow_G^*$ for the reflexive, transitive closure of $\Rightarrow_G$. Then the language generated by $G$ is $L(G) = \{ w \in \Sigma^* | S \Rightarrow_G^* w \}$.

### 5.2.3 Structure and ambiguity

As has already been alluded to, CFGs are interesting not only because they can generate more string languages than finite automata can, but because they build trees, known as syntactic analyses, phrase-structure trees, or parse trees. Whenever we use a rule $A \rightarrow \beta$ to rewrite a nonterminal $A$, we don’t erase $A$ and replace it with $\beta$; instead, we make the symbols of $\beta$ the children of $A$.

**Question 3.** What would the tree for sentence (6.3) be?

One of the main purposes of these trees is that every subtree of the parse tree is supposed to have a semantics or meaning, so that the tree shows how to interpret the sentence. As a result, it is possible that a single string can have more than one structure, and therefore more than one meaning. This is called ambiguity. To illustrate it, we need a new example.

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow DT \ NN \\
NP & \rightarrow NN \\
NP & \rightarrow NN \ NNS \\
VP & \rightarrow VBP \ NP \\
VP & \rightarrow VBP \\
VP & \rightarrow VP \ PP \\
PP & \rightarrow IN \ NP \\
DT & \rightarrow a \ | \ an \\
NN & \rightarrow time \ | \ fruit \ | \ arrow \ | \ banana \\
NNS & \rightarrow flies \\
VBP & \rightarrow flies \ | \ like \\
IN & \rightarrow like \\
\end{align*}
\]

This grammar generates (among others) the following two strings:

(5.43) Time flies like an arrow.

(5.44) Fruit flies like a banana.
Their “natural” structures are:

\[(5.45)\]
\[
\begin{array}{c}
S \\
NP \\
NN \\
\text{time} \\
\text{flies} \\
an \\
\end{array}
\begin{array}{c}
VP \\
\text{flies} \\
\text{like} \\
\text{DT} \\
an \\
\end{array}
\begin{array}{c}
PP \\
\text{arrow} \\
\end{array}
\begin{array}{c}
NP \\
\end{array}
\]

\[(5.46)\]
\[
\begin{array}{c}
S \\
NP \\
NN \\
\text{fruit} \\
\text{flies} \\
an \\
\end{array}
\begin{array}{c}
VP \\
\text{like} \\
\text{DT} \\
\text{NN} \\
\end{array}
\begin{array}{c}
PP \\
\text{a} \\
\text{banana} \\
\end{array}
\begin{array}{c}
NP \\
\end{array}
\]

But the grammar also allows other structures, which would lead to other meanings:

\[(5.47)\]
\[
\begin{array}{c}
S \\
NP \\
NN \\
\text{time} \\
\text{flies} \\
an \\
\end{array}
\begin{array}{c}
VP \\
\text{like} \\
\text{DT} \\
\text{NN} \\
\end{array}
\begin{array}{c}
PP \\
\text{arrow} \\
\end{array}
\begin{array}{c}
NP \\
\end{array}
\]

\[(5.48)\]
\[
\begin{array}{c}
S \\
NP \\
NN \\
\text{fruit} \\
\text{flies} \\
an \\
\end{array}
\begin{array}{c}
VP \\
\text{like} \\
\text{DT} \\
\text{NN} \\
\end{array}
\begin{array}{c}
PP \\
\text{a} \\
\text{banana} \\
\end{array}
\begin{array}{c}
NP \\
\end{array}
\]

Interpretation (6.15) says that a certain kind of fly, the time fly, is fond of arrows. Interpretation (6.16) says that fruits generally fly in the same way that bananas fly.

**Question 4.** The English word *buffalo* has two meanings: it can be a noun (the name of several species of oxen) or a verb (to overpower, overawe, or constrain by superior force or influence; to outwit, perplex). Also, the plural of the noun *buffalo* is *buffalo*. Therefore, the following strings are all grammatical:

\[(5.49)\] Buffalo! (Overpower!)

\[(5.50)\] Buffalo buffalo. (Oxen overpower.)
In fact, the entire set \( \{ \text{buffalo}^n \mid n \geq 1 \} \) is a subset of English. Can you write a CFG that generates it according to English grammar? Hint: here is a tree for the fourth example:

(5.53) Buffalo buffalo buffalo buffalo buffalo.

How many structures can you find for the following sentence:

(5.54) Buffalo buffalo buffalo buffalo buffalo.

### 5.2.4 Weighted context free grammars

Weighted CFGs are a straightforward extension of CFGs. Recall that FSTs map an input string to a set of possible output strings, whereas weighted FSTs give us a distribution over possible output strings. In the same way, weighted CFGs help us deal with ambiguity (a single string having multiple structures) by giving us a distribution over possible structures.

In a weighted CFG, every production has a weight attached to it, which we write as

\[ A \xrightarrow{p} \beta \]

The weight of a derivation is the product of the weights of the rules used in the derivation (if a rule is used \( k \) times, we multiply its weight in \( k \) times).
Thus we can take our grammar from last time and add weights:

$$\begin{align*}
S & \xrightarrow{\frac{1}{2}} \text{NP VP} \\
\text{NP} & \xrightarrow{0.5} \text{DT NN} \\
\text{NP} & \xrightarrow{0.4} \text{NN} \\
\text{NP} & \xrightarrow{0.1} \text{NN NNS} \\
\text{VP} & \xrightarrow{0.6} \text{VBP NP} \\
\text{VP} & \xrightarrow{0.3} \text{VBP} \\
\text{VP} & \xrightarrow{0.1} \text{VP PP} \\
\text{PP} & \xrightarrow{1} \text{IN NP}
\end{align*}$$

$$\begin{align*}
\text{DT} & \xrightarrow{0.5} \text{a} \\
\text{DT} & \xrightarrow{0.5} \text{an} \\
\text{NN} & \xrightarrow{0.25} \text{time} \\
\text{NN} & \xrightarrow{0.25} \text{fruit} \\
\text{NN} & \xrightarrow{0.25} \text{arrow} \\
\text{NN} & \xrightarrow{0.25} \text{banana} \\
\text{NNS} & \xrightarrow{1} \text{flies} \\
\text{VBP} & \xrightarrow{0.5} \text{flies} \\
\text{IN} & \xrightarrow{1} \text{like}
\end{align*}$$

**Question.** What would the weight of these two derivations be?

![Diagram of two derivations]

*A probabilistic CFG or PCFG* is one in which the probabilities of all rules with a given left-hand side sum to one (Booth and Thompson, 1973). A PCFG is called *consistent* if the probabilities of all derivations sum to one.

Aren’t all PCFGs consistent? Actually, no:

$$\begin{align*}
S & \xrightarrow{0.9} \text{SS} \\
S & \xrightarrow{0.1} \text{a}
\end{align*}$$

Let $P_n$ be the total weight of trees of height $\leq n$. Thus

$$\begin{align*}
P_1 &= 0.1 \\
P_{n+1} &= 0.9P_n^2 + 0.1
\end{align*}$$
The second equation is because a tree of height \( \leq n + 1 \) can either be a tree of height 1, formed using rule (6.25), or a tree formed using rule (6.24) and two trees of height \( \leq n \). The limit \( P = \lim_{n \to \infty} P_n \) must be a fixed point of the second equation. There are two fixed points: \( P = 0.9P^2 + 0.1 \Rightarrow P = \frac{1}{9} \) or 1. But note that \( P_1 < \frac{1}{9} \), and \( P_i < \frac{1}{9} \Rightarrow P_{i+1} < \frac{1}{9} \). Since the sequence is always less than \( \frac{1}{9} \), it cannot converge to 1. Therefore, \( P = \frac{1}{9} \).

**Question.** What happened to the other \( \frac{8}{9} \)?

### 5.3 Parsing Algorithms

Next, we explore the parsing problem, which encompasses several questions, including:

- Does \( L(G) \) contain \( w \)?
- What is the highest-weight derivation of \( w \)?
- What is the set of all derivations of \( w \)?

#### 5.3.1 Chomsky normal form

Let’s assume that \( G \) has a particularly simple form. We say that a CFG is in Chomsky normal form if each of its productions has one of the following forms:

\[
X \rightarrow YZ
\]

\[
X \rightarrow a
\]

It can be shown (see below) that any context-free grammar not generating a language containing \( \epsilon \) can be converted into Chomsky normal form, and still generate the same language.

Our grammar from above can be massaged to be in Chomsky normal form:

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow DT \ NN \\
NP & \rightarrow time \mid fruit \\
NP & \rightarrow NN \ NNS \\
VP & \rightarrow VBP \ NP \\
VP & \rightarrow flies \\
VP & \rightarrow VP \ PP \\
PP & \rightarrow IN \ NP \\
DT & \rightarrow a \mid an \\
NN & \rightarrow time \mid fruit \mid arrow \mid banana \\
NNS & \rightarrow flies \\
VBP & \rightarrow like \\
IN & \rightarrow like
\end{align*}
\]
5.3.2 The CKY algorithm

The CKY algorithm is named after three people who independently invented it: Cocke, Kasami, and Younger, although it has been rediscovered more times than that.

In its most basic form, the algorithm just decides whether \( w \in L(G) \). It builds a data structure known as a chart; it is an \( n \times n \) array. The element \( chart[i, j] \) is a set of nonterminal symbols. If \( X \in chart[i, j] \), then that means we have discovered that \( X \Rightarrow^* w_{i+1} \cdots w_j \).

**Require:** string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)

**Ensure:** \( w \in L(G) \) iff \( S \in chart[0, n] \)

\begin{algorithmic}[1]
\Require string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)
\Ensure \( w \in L(G) \) iff \( S \in chart[0, n] \)
\State initialize \( chart[i, j] \leftarrow \emptyset \) for all \( 0 \leq i < j \leq n \)
\ForAll { \( i \leftarrow 1, \ldots, n \) and \( (X \rightarrow w_i) \in R \) }
\State \( chart[i - 1, i] \leftarrow chart[i - 1, i] \cup \{X\} \)
\EndFor
\For { \( \ell \leftarrow 2, \ldots, n \) }
\For { \( i \leftarrow 0, \ldots, n - \ell \) }
\State \( j \leftarrow i + \ell \)
\For { \( k \leftarrow i + 1, \ldots, j - 1 \) }
\ForAll { \( (X \rightarrow YZ) \in R \) }
\State if \( Y \in chart[i, k] \) and \( Z \in chart[k, j] \) then
\State \( chart[i, j] \leftarrow chart[i, j] \cup \{X\} \)
\EndIf
\EndFor
\EndFor
\EndFor
\EndFor
\EndFor

**Question 5.** What is the time and space complexity of this algorithm?

**Question 6.** Using the grammar (7.1), run the CKY algorithm on the string:

\( \text{0 time 1 flies 2 like 3 an 4 arrow 5} \)
5.3.3 Viterbi CKY

But it is much more useful to find the highest-weight parse. Suppose that our grammar has the following probabilities:

\[
\begin{align*}
S & \xrightarrow{1} NP \ VP \\
NP & \xrightarrow{0.5} DT \ NN \\
NP & \xrightarrow{0.2} time \\
NP & \xrightarrow{0.2} fruit \\
NP & \xrightarrow{0.1} NN \ NNS \\
VP & \xrightarrow{0.6} VBP \ NP \\
VP & \xrightarrow{0.3} flies \\
NP & \xrightarrow{0.1} VP \ PP \\
PP & \xrightarrow{1} IN \ NP
\end{align*}
\]

Then we use a modification of CKY that is analogous to the Viterbi algorithm. First, we modify the algorithm to find the maximum weight:
Require: string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)

Ensure: \( \text{best}[0,n][S] \) is the maximum weight of a parse of \( w \)

1. initialize \( \text{best}[i, j][X] \leftarrow 0 \) for all \( 0 \leq i < j \leq n, X \in N \)

2. for all \( i \leftarrow 1, \ldots, n \) and \( (X \xrightarrow{p} w_i) \in R \) do
   3. \( \text{best}[i - 1, i][X] \leftarrow \max\{ \text{best}[i - 1, i][X], p \} \)

end for

4. for \( \ell \leftarrow 2, \ldots, n \) do
   5. for \( i \leftarrow 0, \ldots, n - \ell \) do
      6. \( j \leftarrow i + \ell \)
      7. for \( k \leftarrow i + 1, \ldots, j - 1 \) do
         8. for all \( (X \xrightarrow{p} YZ) \in R \) do
            9. \( p' \leftarrow p \times \text{best}[i, k][Y] \times \text{best}[k, j][Z] \)
            10. \( \text{best}[i, j][X] \leftarrow \max\{ \text{best}[i, j][X], p' \} \)
         11. end for
      12. end for
   13. end for
5. for \( \ell \leftarrow 2, \ldots, n \) do
   6. for \( i \leftarrow 0, \ldots, n - \ell \) do
      7. \( j \leftarrow i + \ell \)
      8. for \( k \leftarrow i + 1, \ldots, j - 1 \) do
         9. for all \( (X \xrightarrow{p} YZ) \in R \) do
            10. \( p' \leftarrow p \times \text{best}[i, k][Y] \times \text{best}[k, j][Z] \)
            11. if \( p' > \text{best}[i, j][X] \) then
               12. \( \text{best}[i, j][X] \leftarrow p' \)
               13. \( \text{back}[i, j][X] \leftarrow (X_{i-1,i} \rightarrow w_i) \)
            14. end if
         15. end for
      16. end for
   17. end for

Question 7. Do you see how to modify the algorithm to compute the total weight of all parses of \( w \)?

A slight further modification lets us find the maximum-weight parse itself. Just as in the Viterbi algorithm for FSAs, whenever we update \( \text{best}[i, j][X] \) to a new best weight, we also need to store a back-pointer that records how we obtained that weight. We will represent back-pointers like this: \( X_{i,j} \rightarrow Y_{i,k}Z_{k,j} \) means that we built an \( X \) spanning \( i, j \) from a \( Y \) spanning \( i, k \) and a \( Z \) spanning \( k, j \).

Require: string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)

Ensure: \( G' \) generates the best parse of \( w \)

Ensure: \( \text{best}[0,n][S] \) is its weight

1. for all \( 0 \leq i < j \leq n, X \in N \) do
   2. initialize \( \text{best}[i, j][X] \leftarrow 0 \)
   3. initialize \( \text{back}[i, j][X] \leftarrow \text{nil} \)
2. end for

4. for all \( i \leftarrow 1, \ldots, n \) and \( (X \xrightarrow{p} w_i) \in R \) do
   5. if \( p > \text{best}[i - 1, i][X] \) then
      6. \( \text{best}[i - 1, i][X] \leftarrow p \)
      7. \( \text{back}[i - 1, i][X] \leftarrow (X_{i-1,i} \rightarrow w_i) \)
   8. end if
   9. end if
10. end for

11. for \( \ell \leftarrow 2, \ldots, n \) do
   12. for \( i \leftarrow 0, \ldots, n - \ell \) do
      13. \( j \leftarrow i + \ell \)
      14. for \( k \leftarrow i + 1, \ldots, j - 1 \) do
         15. for all \( (X \xrightarrow{p} YZ) \in R \) do
            16. \( p' \leftarrow p \times \text{best}[i, k][Y] \times \text{best}[k, j][Z] \)
            17. if \( p' > \text{best}[i, j][X] \) then
               18. \( \text{best}[i, j][X] \leftarrow p' \)
               19. \( \text{back}[i, j][X] \leftarrow (X_{i-1,i} \rightarrow w_i) \)
            14. end if
         15. end for
      16. end for
   17. end for
Chapter 5. Syntax

18: \[ \text{best}[i,j][X] \leftarrow p' \]
19: \[ \text{back}[i,j][X] \leftarrow (X_{i,j} \rightarrow Y_{i,k}Z_{k,j}) \]
20: \textbf{end if}
21: \textbf{end for}
22: \textbf{end for}
23: \textbf{end for}
24: \textbf{end for}
25: \( G' = \{ \text{back}[i,j][X] \mid 0 \leq i < j \leq n, X \in N \} \)

\( G' \) is then a grammar that generates at most one tree, the best tree for \( w \).

**Question 8.** Using the grammar (7.2), run the Viterbi CKY algorithm on the same string:

0 time 1 flies 2 like 3 an 4 arrow 5

5.3.4 Parsing general CFGs

Previously, we learned about PCFGs, and how to find the best PCFG derivation of a string using the Viterbi algorithm. Now we will extend those algorithms to the general CFG case.
**Binarization**

It turns out that any CFG (whose language does not contain $\epsilon$) can be converted into an equivalent grammar in Chomsky normal form.

To guarantee that $k \leq 2$, we must eliminate all rules with right-hand side longer than 2. We will see below that the grammars we extract from training data may already have this property. But if not, we need to binarize the grammar. For example, suppose we have the production

$$NP \rightarrow DT \ JJS \ NN \ NN \ PP$$  \hspace{1cm} (5.62)

which is too long to be in Chomsky normal form. There are many ways to break this down into smaller rules, but here is one way. We create a bunch of new nonterminal symbols $NP(\beta)$ where $\beta$ is a string of nonterminal symbols; this stands for a partial NP whose sisters to the left are $\beta$. Then we replace rule (7.3) with:

\begin{align*}
NP & \rightarrow DT \ NP(DT) \hspace{1cm} (5.63) \\
NP(DT) & \rightarrow JJS \ NP(DT,JJS) \hspace{1cm} (5.64) \\
NP(DT,JJS) & \rightarrow NN \ NP(DT,JJS,NN) \hspace{1cm} (5.65) \\
NP(DT,JJS,NN) & \rightarrow NN \ NP(DT,JJS,NN,NN) \hspace{1cm} (5.66) \\
NP(DT,JJS,NN,NN) & \rightarrow PP \hspace{1cm} (5.67)
\end{align*}

Note that the annotations contain enough information to reverse the binarization. So the binarized grammar is equivalent to the unbinarized grammar, but has $k \leq 2$.

**Parsing with unary rules**

But we are not done yet. CKY does not just require $k \leq 2$, but also forbids rules of any of the following forms:

\begin{align*}
A & \rightarrow ab \hspace{1cm} (5.68) \\
A & \rightarrow aB \hspace{1cm} (5.69) \\
A & \rightarrow Ab \hspace{1cm} (5.70) \\
A & \rightarrow \epsilon \hspace{1cm} (5.71) \\
A & \rightarrow B \hspace{1cm} (5.72)
\end{align*}

The first three cases are very easy to eliminate, but we never see them in grammars induced from the Penn Treebank. **Nullary rules** (7.12) are not hard to eliminate (Hopcroft and Ullman, 1979), but the weighted case can be nasty (Stolcke, 1995). Fortunately, nullary rules aren’t very common in practice, so we won’t bother with them here.

**Unary rules** (7.13) are quite common and annoying. Like nullary rules, they are not hard to eliminate from a CFG (Hopcroft and Ullman, 1979), but in practice, most people don’t try to; instead, they extend the CKY algorithm to handle them directly. The extension shown below is not the most efficient, but fits most naturally with the way we have implemented CKY.
Chapter 5. Syntax

Require: string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)
Ensure: \( w \in L(G) \) iff \( S \in \text{chart}[0, n] \)

1: initialize \( \text{chart}[i, j] \leftarrow \emptyset \) for all \( 0 \leq i < j \leq n \)
2: for all \( i \leftarrow 1, \ldots, n \) and \( (X \to w_i) \in R \) do
3: \( \text{chart}[i - 1, i] \leftarrow \text{chart}[i - 1, i] \cup \{X\} \)
4: end for
5: for \( \ell \leftarrow 2, \ldots, n \) do
6: for \( i \leftarrow 0, \ldots, n - \ell \) do
7: \( j \leftarrow i + \ell \)
8: for all \( k \leftarrow i + 1, \ldots, j - 1 \) do
9: \( \text{chart}[i, j] \leftarrow \text{chart}[i, j] \cup \{X\} \)
10: end for
11: end for
12: end for
13: end while
14: end for
15: again \leftarrow true
16: while again do
17: again \leftarrow false
18: for all \( (X \to Y) \in R \) do
19: \( \text{chart}[i, j] \leftarrow \text{chart}[i, j] \cup \{X\} \)
20: end if
21: end for
22: end while
23: end for
24: end for
25: end for
26: end for

The new part is lines 15–24 and is analogous to the binary rule case.

**Question.** Why is the while loop on line 16 necessary? What is its maximum number of iterations?

Finally, let’s put together weights (Viterbi CKY) and unary rules:

Require: string \( w = w_1 \cdots w_n \) and grammar \( G = (N, \Sigma, R, S) \)
Ensure: \( G' \) generates the best parse of \( w \)
Ensure: \( \text{best}[0, n][S] \) is its weight

1: for all \( 0 \leq i < j \leq n, X \in N \) do
2: initialize \( \text{best}[i, j][X] \leftarrow 0 \)
3: initialize \( \text{back}[i, j][X] \leftarrow \text{nil} \)
4: end for
5: for all \( i \leftarrow 1, \ldots, n \) and \( (X \to p \to w_i) \in R \) do
6: \( \text{if } p > \text{best}[i - 1, i][X] \text{ then} \)
7: \( \text{best}[i - 1, i][X] \leftarrow p \)
8: \( \text{back}[i - 1, i][X] \leftarrow (X_i \to w_i) \)
9: end if
10: end for
11: for \( \ell \leftarrow 2, \ldots, n \) do
for \( i \leftarrow 0, \ldots, n - \ell \) do
\[
\begin{align*}
j & \leftarrow i + \ell \\
\text{for } k \leftarrow i + 1, \ldots, j - 1 \text{ do} \\
& \text{ for all } (X \xrightarrow{p} YZ) \in R \text{ do} \\
& \quad p' \leftarrow p \times \text{best}[i, k][Y] \times \text{best}[k, j][Z] \\
& \quad \text{if } p' > \text{best}[i, j][X] \text{ then} \\
& \quad \quad \text{best}[i, j][X] \leftarrow p' \\
& \quad \quad \text{back}[i, j][X] \leftarrow (X_{i,j} \to Y_{i,k} Z_{k,j}) \\
& \quad \text{end if} \\
& \quad \text{end for} \\
& \text{end for} \\
& \text{again} \leftarrow \text{true} \\
& \text{while again do} \\
& \quad again \leftarrow \text{false} \\
& \quad \text{for all } (X \xrightarrow{p} Y) \in R \text{ do} \\
& \quad \quad p' \leftarrow p \times \text{best}[i, j][Y] \\
& \quad \quad \text{if } p' > \text{best}[i, j][X] \text{ then} \\
& \quad \quad \quad \text{best}[i, j][X] = p' \\
& \quad \quad \quad \text{back}[i, j][X] \leftarrow (X_{i,j} \to Y_{i,j}) \\
& \quad \quad \quad again \leftarrow \text{true} \\
& \quad \text{end if} \\
& \quad \text{end for} \\
& \text{end while} \\
& \text{end for} \\
& G' \leftarrow \text{EXTRACT}(S, 0, n)
\end{align*}
\]

If the grammar has unary cycles in it, that is, it is possible to derive \( X \Rightarrow \ldots \Rightarrow^* X \), then certain complications can arise from the fact that a string may have an infinite number of derivations. In particular, if the weight of the cycle is greater than 1, then the Viterbi CKY algorithm will break. Even if all rule weights are less than 1, some algorithms require modification; for example, if we want to find the total weight of all the derivations of a string, we have to perform an infinite summation (Stolcke, 1995). Therefore, it is fairly common to implement hacks of various kinds to break the cycles. For example, we could modify the grammar so that it goes round the cycle at most five times.

**Question 9.** Why doesn’t the Viterbi CKY algorithm break on unary cycles if we assume that all rule weights are less than 1?
Bibliography

