Chapter 6

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.

6.1 Why Syntax?

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, let's look at some examples.

6.1.1 A quiz (not for credit)

For each pair of sentences, choose which one is better.

1. (a) I don't like being made fun of.
   (b) I don't like being taken pictures of.
2. Who'd like some ice cream?
   (a) I'd!
   (b) I would!
3. (a) I saw a brown big spider.
   (b) I saw a big brown spider.
4. (a) That movie was fan-freaking-tastic.
   (b) That movie was fantas-freaking-tic.
5. (a) That's the teacher that I couldn't understand what he was talking about.
   (b) That's the teacher that I couldn't understand what was talking about.

6.1.2 Newspaper ambiguities

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?

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

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

(6.3) 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.

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

6.1.3 Implementing subject-aux inversion

There are situations where we want computers to perform transformations on text that seem to require knowledge of concepts like "subject" or "object" – for example, machine translation. To keep things monolingual, imagine that we want a computer to turn statements into questions. How would you write a program to do this?

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

(6.6) a. Soylent Green is people.
b. Is Soylent Green people?

(6.7) a. Good fences make good neighbors.
b. Make good fences good neighbors?
c. Do good fences make good neighbors?

(6.8) 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?

6.1.4 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?

[1] `http://www.ling.upenn.edu/~beatrice/humor/newspaper-screwups.html`

(6.9) Fences make, good good neighbors.

(6.10) Make good, good fences neighbors.

(6.11) Good neighbors, good fences make.

(6.12) Fences make good, good neighbors.

(6.13) Make good neighbors, good fences.

Can you replace it with something like a pronoun?

(6.14) They make good neighbors.

(6.15) Good they good neighbors.

(6.16) Good fences it/they/. . . neighbors.

(6.17) Good fences make them.

(6.18) They good neighbors.

(6.19) Good it/they/. . . neighbors.

(6.20) Good fences do.

Can it participate in a *cleft*?

(6.21) It's good fences that make good neighbors.

(6.22) It's fences make that good good neighbors.

(6.23) It's make good that good fences neighbors.

(6.24) It's good neighbors that good fences make.

(6.25) It's good fences make that [do] good neighbors.

(6.26) It's make good neighbors that good fences do.

Can it be the answer to a question?

(6.27) What makes good neighbors? Good fences.

(6.28) Good what good neighbors? Fences make.

(6.29) Good fences what neighbors? Make good.

(6.30) Good fences make what? Good neighbors.

(6.31) What [do] good neighbors? Good fences make.

(6.32) Good fences do what? Make good neighbors.

6.2 Context Free Grammars

6.2.1 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

$$L = \{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\} \tag{6.33}$$

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

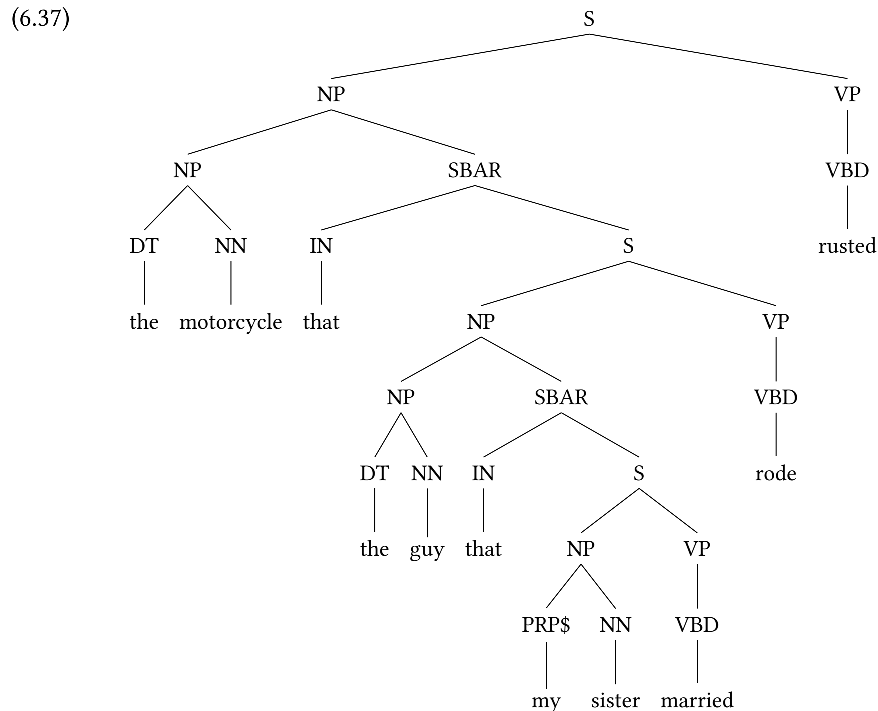
(6.34) The motorcyle rusted.

(6.35) The motorcycle that the guy rode rusted.

(6.36) 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

(6.37)



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:

(6.38) The cheese lay in the house that Jack built.

(6.39) The cheese that the rat ate lay in the house that Jack built.

(6.40) The cheese that the rat that the cat killed ate lay in the house that Jack built.

(6.41) The cheese that the rat that the cat that the dog worried killed ate lay in the house that Jack built.

⋮

(6.42) The cheese that the rat that the cat that the dog that the cow with the crumpled horn that the maiden all forlorn that the man all tattered and torn that the priest all shaven and shorn that the rooster that crowed in the morn that the farmer sowing his corn that the horse and the hound and the horn belonged to kept woke married kissed milked tossed worried killed ate lay in the house that Jack built.

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

6.2.2 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 $\{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\}$:

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{aSb} \\ \mathrm{S} &\rightarrow \epsilon \end{aligned} \tag{6.43}$$

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:

$$\begin{aligned} \mathrm{S} &\Rightarrow \mathrm{aSb} \\ &\Rightarrow \mathrm{aaSbb} \\ &\Rightarrow \mathrm{aaaSbbb} \\ &\Rightarrow \mathrm{aaabbb} \end{aligned} \tag{6.44}$$

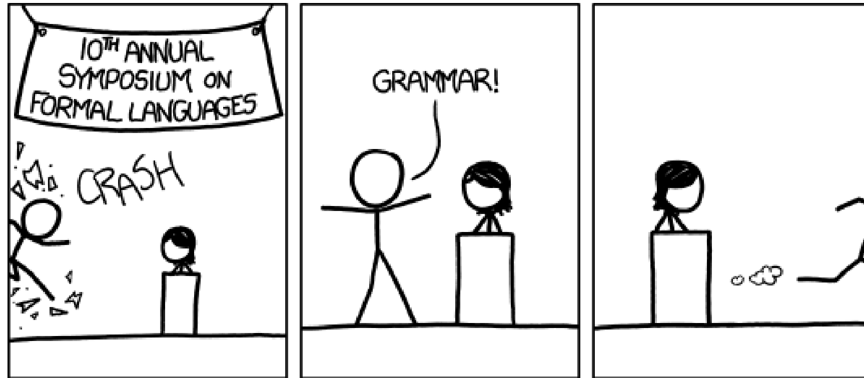


Figure 6.1: Formal Languages. [audience looks around] 'What just happened?' 'There must be some context we're missing.'

Our next example generates sentences (6.34), (6.35), and (6.36).

$$\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \text{DT NN} \\
\text{NP} &\rightarrow \text{PRP\$ NN} \\
\text{NP} &\rightarrow \text{NP SBAR} \\
\text{VP} &\rightarrow \text{VBD} \\
\text{SBAR} &\rightarrow \text{IN S} \\
\text{DT} &\rightarrow \text{the} \\
\text{PRP\$} &\rightarrow \text{my} \\
\text{NN} &\rightarrow \text{motorcycle} \mid \text{guy} \mid \text{sister} \\
\text{VBD} &\rightarrow \text{married} \mid \text{rode} \mid \text{rusted} \\
\text{IN} &\rightarrow \text{that}
\end{aligned} \tag{6.45}$$

Here, the uppercase symbols are nonterminal symbols, and the English words are terminal symbols. Also, we have used some shorthand: $A \rightarrow \beta_1 \mid \beta_2$ 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 (6.35)?

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*
- Σ 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 $\alpha A \gamma \Rightarrow_G \alpha\beta\gamma$ 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 $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

6.2.3 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 β; instead, we make the symbols of β the children of A.

Question 3. What would the tree for sentence (6.35) 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.

$$
\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \text{DT NN} \\
\text{NP} &\rightarrow \text{NN} \\
\text{NP} &\rightarrow \text{NN NNS} \\
\text{VP} &\rightarrow \text{VBP NP} \\
\text{VP} &\rightarrow \text{VBP} \\
\text{VP} &\rightarrow \text{VP PP} \\
\text{PP} &\rightarrow \text{IN NP} \\
\text{DT} &\rightarrow \text{a} \mid \text{an} \\
\text{NN} &\rightarrow \text{time} \mid \text{fruit} \mid \text{arrow} \mid \text{banana} \\
\text{NNS} &\rightarrow \text{flies} \\
\text{VBP} &\rightarrow \text{flies} \mid \text{like} \\
\text{IN} &\rightarrow \text{like}
\end{aligned}
\tag{6.46}
$$

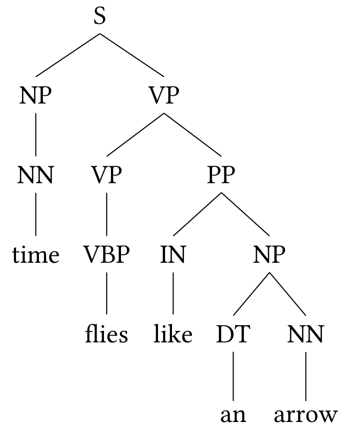
This grammar generates (among others) the following two strings:

(6.47) Time flies like an arrow.

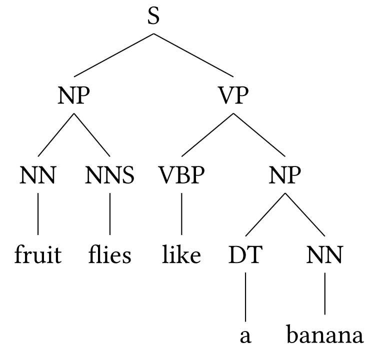
(6.48) Fruit flies like a banana.

Their "natural" structures are:

(6.49)

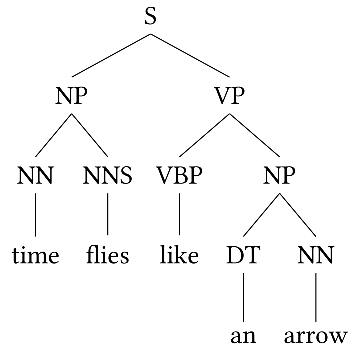


(6.50)

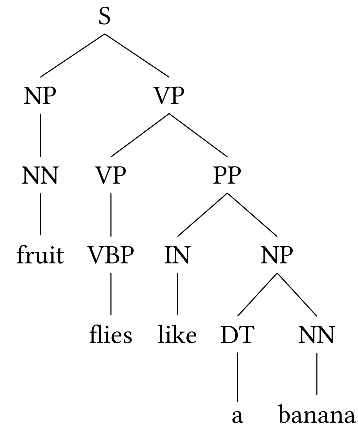


But the grammar also allows other structures, which would lead to other meanings:

(6.51)



(6.52)



Interpretation (6.51) says that a certain kind of fly, the time fly, is fond of arrows. Interpretation (6.52) 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:

(6.53) Buffalo! (Overpower!)

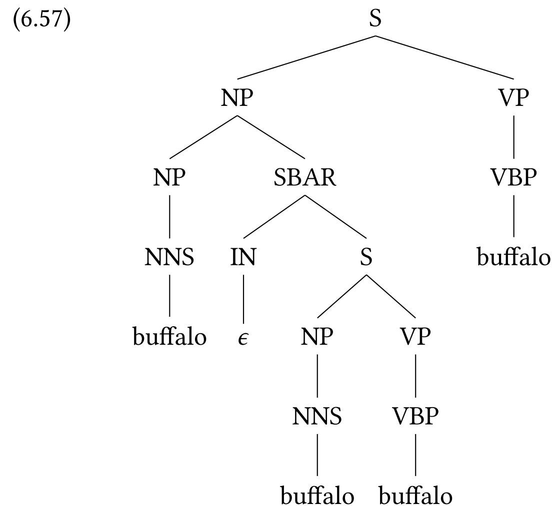
(6.54) Buffalo buffalo. (Oxen overpower.)

(6.55) Buffalo buffalo buffalo. (Oxen overpower oxen.)

(6.56) Buffalo buffalo buffalo buffalo. (Oxen that overpower 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:

(6.57)



How many structures can you find for the following sentence:

(6.58) Buffalo buffalo buffalo buffalo buffalo.

6.2.4 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

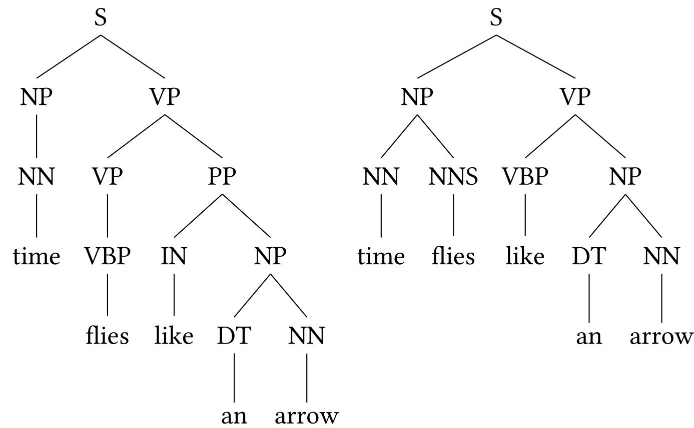
$$A \xrightarrow{p} \beta$$

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:

$$\begin{array}{rlrl} & & \text{DT} & \xrightarrow{0.5} \text{a} \\ \text{S} & \xrightarrow{1} \text{NP VP} & \text{DT} & \xrightarrow{0.5} \text{an} \\ \text{NP} & \xrightarrow{0.5} \text{DT NN} & \text{NN} & \xrightarrow{0.25} \text{time} \\ \text{NP} & \xrightarrow{0.4} \text{NN} & \text{NN} & \xrightarrow{0.25} \text{fruit} \\ \text{NP} & \xrightarrow{0.1} \text{NN NNS} & \text{NN} & \xrightarrow{0.25} \text{arrow} \\ \text{VP} & \xrightarrow{0.6} \text{VBP NP} & \text{NN} & \xrightarrow{0.25} \text{banana} \\ \text{VP} & \xrightarrow{0.3} \text{VBP} & \text{NNS} & \xrightarrow{1} \text{flies} \\ \text{VP} & \xrightarrow{0.1} \text{VP PP} & \text{VBP} & \xrightarrow{0.5} \text{flies} \\ \text{PP} & \xrightarrow{1} \text{IN NP} & \text{VBP} & \xrightarrow{0.5} \text{like} \\ & & \text{IN} & \xrightarrow{1} \text{like} \end{array} \tag{6.59}$$

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



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:

$$\text{S} \xrightarrow{0.9} \text{SS} \tag{6.60}$$

$$\text{S} \xrightarrow{0.1} \text{a} \tag{6.61}$$

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

$$P_1 = 0.1 \tag{6.62}$$

$$P_{n+1} = 0.9P_n^2 + 0.1 \tag{6.63}$$

The second equation is because a tree of height $\leq n + 1$ can either be a tree of height 1, formed using rule (6.61), or a tree formed using rule (6.60) 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 $\frac{8}{9}$?

6.3 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?

6.3.1 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:

$$\begin{aligned} X &\to YZ \\ X &\to a \end{aligned}$$

It can be shown (see below) that any context-free grammar not generating a language containing ϵ 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:

$$\begin{aligned}
\text{S} &\to \text{NP VP} \\
\text{NP} &\to \text{DT NN} \\
\text{NP} &\to \text{time} \mid \text{fruit} \\
\text{NP} &\to \text{NN NNS} \\
\text{VP} &\to \text{VBP NP} \\
\text{VP} &\to \text{flies} \\
\text{VP} &\to \text{VP PP} \\
\text{PP} &\to \text{IN NP} \\
\text{DT} &\to \text{a} \mid \text{an} \\
\text{NN} &\to \text{time} \mid \text{fruit} \mid \text{arrow} \mid \text{banana} \\
\text{NNS} &\to \text{flies} \\
\text{VBP} &\to \text{like} \\
\text{IN} &\to \text{like}
\end{aligned} \tag{6.64}$$

6.3.2 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 $S \in chart[0, n]$

```
 1: initialize chart[i, j] ← ∅ for all 0 ≤ i < j ≤ n
 2: for all i ← 1, ..., n and (X → w_i) ∈ R do
 3:     chart[i − 1, i] ← chart[i − 1, i] ∪ {X}
 4: end for
 5: for ℓ ← 2, ..., n do
 6:     for i ← 0, ..., n − ℓ do
 7:         j ← i + ℓ
 8:         for k ← i + 1, ..., j − 1 do
 9:             for all (X → YZ) ∈ R do
10:                 if Y ∈ chart[i, k] and Z ∈ chart[k, j] then
11:                     chart[i, j] ← chart[i, j] ∪ {X}
12:                 end if
13:             end for
14:         end for
15:     end for
16: end for
```

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

Question 6. Using the grammar (6.64), run the CKY algorithm on the string:

$$_0 \text{ time } _1 \text{ flies } _2 \text{ like } _3 \text{ an } _4 \text{ arrow } _5$$

0,1	0,2	0,3	0,4	0,5
	1,2	1,3	1,4	1,5
		2,3	2,4	2,5
			3,4	3,5
				4,5

6.3.3 Viterbi CKY

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

$$\begin{aligned} \text{S} &\xrightarrow{1} \text{NP VP} & \text{DT} &\xrightarrow{0.5} \text{a} \\ \text{NP} &\xrightarrow{0.5} \text{DT NN} & \text{DT} &\xrightarrow{0.5} \text{an} \\ \text{NP} &\xrightarrow{0.2} \text{time} & \text{NN} &\xrightarrow{0.25} \text{time} \\ \text{NP} &\xrightarrow{0.2} \text{fruit} & \text{NN} &\xrightarrow{0.25} \text{fruit} \\ \text{NP} &\xrightarrow{0.1} \text{NN NNS} & \text{NN} &\xrightarrow{0.25} \text{arrow} \\ \text{VP} &\xrightarrow{0.6} \text{VBP NP} & \text{NN} &\xrightarrow{0.25} \text{banana} \\ \text{VP} &\xrightarrow{0.3} \text{flies} & \text{NNS} &\xrightarrow{1} \text{flies} \\ \text{VP} &\xrightarrow{0.1} \text{VP PP} & \text{VBP} &\xrightarrow{1} \text{like} \\ \text{PP} &\xrightarrow{1} \text{IN NP} & \text{IN} &\xrightarrow{1} \text{like} \end{aligned} \tag{6.65}$$

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: $best[0, n][S]$ is the maximum weight of a parse of w

```
1:  initialize best[i, j][X] ← 0 for all 0 ≤ i < j ≤ n, X ∈ N
2:  for all i ← 1, ..., n and (X →^p w_i) ∈ R do
3:      best[i − 1, i][X] ← max{best[i − 1, i][X], p}
4:  end for
5:  for ℓ ← 2, ..., n do
6:      for i ← 0, ..., n − ℓ do
7:          j ← i + ℓ
8:          for k ← i + 1, ..., j − 1 do
9:              for all (X →^p YZ) ∈ R do
10:                 p′ ← p × best[i, k][Y] × best[k, j][Z]
11:                 best[i, j][X] ← max{best[i, j][X], p′}
12:             end for
13:         end for
14:     end for
15: 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 $best[i, j][X]$ 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: $best[0, n][S]$ is its weight

```
1:  for all 0 ≤ i < j ≤ n, X ∈ N do
2:      initialize best[i, j][X] ← 0
3:      initialize back[i, j][X] ← nil
4:  end for
5:  for all i ← 1, ..., n and (X →^p w_i) ∈ R do
6:      if p > best[i − 1, i][X] then
7:          best[i − 1, i][X] ← p
8:          back[i − 1, i][X] ← (X_{i−1,i} → w_i)
9:      end if
10: end for
11: for ℓ ← 2, ..., n do
12:     for i ← 0, ..., n − ℓ do
13:         j ← i + ℓ
14:         for k ← i + 1, ..., j − 1 do
15:             for all (X →^p YZ) ∈ R do
16:                 p′ ← p × best[i, k][Y] × best[k, j][Z]
17:                 if p′ > best[i, j][X] then
```

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

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

Question 8. Using the grammar (6.65), run the Viterbi CKY algorithm on the same string:

$$_0 \text{ time } _1 \text{ flies } _2 \text{ like } _3 \text{ an } _4 \text{ arrow } _5$$

0,1	0,2	0,3	0,4	0,5
	1,2	1,3	1,4	1,5
		2,3	2,4	2,5
			3,4	3,5
				4,5

Partial solution. Cell $(0, 5)$ is missing.

0,1 NP/0.2/NP → time NN/0.25/NN → time	0,2 S/0.06/S → NP $_1$ VP NP/0.025/NP → NN $_1$ NNS	0,3 $\emptyset$	0,4 $\emptyset$	0,5
	1,2 VP/0.3/VP → flies NNS/1/NNS → flies	1,3 $\emptyset$	1,4 $\emptyset$	1,5 VP/0.001875/VP → VP $_2$ PP
		2,3 VBP/1/VBP → like IN/1/IN → like	2,4 $\emptyset$	2,5 VP/0.0375/VP → VBP $_3$ NP PP/0.0625/PP → IN $_3$ NP
			3,4 DT/0.5/IN → an	3,5 NP/0.0625/NP → DT $_4$ NN
				4,5 NN/0.25/NN → arrow

6.3.4 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 ϵ) 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

$$\text{NP} \rightarrow \text{DT JJS NN NN PP} \tag{6.66}$$

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(β) where β is a string of nonterminal symbols; this stands for a partial NP whose sisters to the *left* are β. Then we replace rule (6.66) with:

$$\text{NP} \rightarrow \text{DT NP(DT)} \tag{6.67}$$

$$\text{NP(DT)} \rightarrow \text{JJS NP(DT,JJS)} \tag{6.68}$$

$$\text{NP(DT,JJS)} \rightarrow \text{NN NP(DT,JJS,NN)} \tag{6.69}$$

$$\text{NP(DT,JJS,NN)} \rightarrow \text{NN NP(DT,JJS,NN,NN)} \tag{6.70}$$

$$\text{NP(DT,JJS,NN,NN)} \rightarrow \text{PP} \tag{6.71}$$

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:

$$A \rightarrow ab \tag{6.72}$$

$$A \rightarrow aB \tag{6.73}$$

$$A \rightarrow Ab \tag{6.74}$$

$$A \rightarrow \epsilon \tag{6.75}$$

$$A \rightarrow B \tag{6.76}$$

The first three cases are very easy to eliminate, but we never see them in grammars induced from the Penn Treebank. *Nullary rules* (6.75) 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 (6.76) 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.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: $w \in L(G)$ iff $S \in \mathit{chart}[0, n]$
1: initialize $\mathit{chart}[i, j] \leftarrow \emptyset$ **for all** $0 \leq i < j \leq n$
2: **for all** $i \leftarrow 1, \ldots, n$ **and** $(X \rightarrow w_i) \in R$ **do**
3: $\quad \mathit{chart}[i-1, i] \leftarrow \mathit{chart}[i-1, i] \cup \{X\}$
4: **end for**
5: **for** $\ell \leftarrow 2, \ldots, n$ **do**
6: $\quad$ **for** $i \leftarrow 0, \ldots, n - \ell$ **do**
7: $\quad\quad j \leftarrow i + \ell$
8: $\quad\quad$ **for** $k \leftarrow i + 1, \ldots, j - 1$ **do**
9: $\quad\quad\quad$ **for all** $(X \rightarrow YZ) \in R$ **do**
10: $\quad\quad\quad\quad$ **if** $Y \in \mathit{chart}[i, k]$ **and** $Z \in \mathit{chart}[k, j]$ **then**

11: $chart[i, j] \leftarrow chart[i, j] \cup \{X\}$
12: **end if**
13: **end for**
14: **end for**
15: $again \leftarrow$ **true**
16: **while** $again$ **do**
17: $again \leftarrow$ **false**
18: **for all** $(X \rightarrow Y) \in R$ **do**
19: **if** $X \notin chart[i, j]$ **and** $Y \in chart[i, j]$ **then**
20: $chart[i, j] \leftarrow chart[i, j] \cup \{X\}$
21: $again \leftarrow$ **true**
22: **end if**
23: **end for**
24: **end while**
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: $best[0, n][S]$ is its weight
1: **for all** $0 \leq i < j \leq n, X \in N$ **do**
2: initialize $best[i, j][X] \leftarrow 0$
3: initialize $back[i, j][X] \leftarrow$ nil
4: **end for**
5: **for all** $i \leftarrow 1, \ldots, n$ **and** $(X \xrightarrow{p} w_i) \in R$ **do**
6: **if** $p > best[i-1, i][X]$ **then**
7: $best[i-1, i][X] \leftarrow p$
8: $back[i-1, i][X] \leftarrow (X_{i-1,i} \rightarrow w_i)$
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: $p' \leftarrow p \times best[i, k][Y] \times best[k, j][Z]$
17: **if** $p' > best[i, j][X]$ **then**
18: $best[i, j][X] \leftarrow p'$
19: $back[i, j][X] \leftarrow (X_{i,j} \rightarrow Y_{i,k} Z_{k,j})$
20: **end if**
21: **end for**
22: **end for**

```
23:         again ← true
24:         while again do
25:             again ← false
26:             for all (X -p-> Y) ∈ R do
27:                 p' ← p × best[i, j][Y]
28:                 if p' > best[i, j][X] then
29:                     best[i, j][X] = p'
30:                     back[i, j][X] ← (X_{i,j} → Y_{i,j})
31:                     again ← true
32:                 end if
33:             end for
34:         end while
35:     end for
36: end for
37: G' ← EXTRACT(S, 0, n)
```

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.

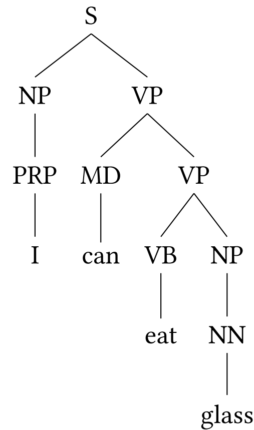
Question 9. Why doesn't the Viterbi CKY algorithm break on unary cycles if we assume that all rule weights are less than 1?

6.4 Neural Parsing

The literature on parsing can be divided into *dependency parsing* and *constituency parsing*. The former is probably more widespread, but we're focusing on the latter. Neural constituency parsing models can be further divided into models that are (loosely) based on pushdown automata (PDAs) and models that are based on context-free grammars.

6.4.1 Stack-based approaches

These approaches treat trees as sequences of symbols and use some kind of stack to ensure that the sequence is well-formed. I give just one example, which is to train a neural language model on trees represented as bracketed strings (Vinyals et al., 2015; Choe and Charniak, 2016). For example,



would become something like

```
[S [NP [PRP I PRP] NP] [VP [MD can MD] [VP [VB eat VB] [NP [NN glass NN] NP] VP] VP] S]
```

To parse, we search among flattened trees for the one which has the highest probability according to the language model. This is an intractable search, but we can make it practical using beam search, in which at each time step we keep only the top k possibilities.

The neural language model is a RNN. I'm not aware of approaches that use Transformer language models, but that would certainly be possible.

6.4.2 Grammar-based approaches

Another approach is to use a weighted CFG, but to compute the weights of the CFG using a neural network. Suppose that the probability of a tree is still

$$P(\text{tree}) = \prod_{(A \rightarrow \beta) \in \text{tree}} P(A \rightarrow \beta)$$

but imagine that $P(A \rightarrow \beta)$ is computed by a neural network, in the hopes that it can learn better probabilities than relative frequency estimation ($P(A \rightarrow \beta) = c(A \rightarrow \beta)/c(A)$). However, relative-frequency estimation is already guaranteed to maximize the likelihood of the training data, so it's not possible to do better! The only way to improve the model is to let the rule scores depend on information outside the rule itself.

For example, maybe a rule probability could depend on the rest of the words in the sentence. This doesn't make sense mathematically, because words depend on the rules, so it's circular for the rules to depend on the words.[2]

The way out is to change from a *generative* model, that is, a model of $P(\text{tree}, w)$ to a *discriminative* model, that is, a model of $P(\text{tree} \mid w)$. Since the tree now depends on the words, instead of the other way around, any rule can depend on any of the words in the sentence.

[2]Although, admittedly, it worked pretty well when I tried it. Something similar happens in the hybrid HMM-DNN model that is still the state of the art in speech recognition.

The model described here is the one provided in HW3. It's a highly simplified version of several neural models that were state-of-the-art at the time (Durrett and Klein, 2015; Stern, Andreas, and Klein, 2017). We start off with an encoder as in a translation system:

$$\begin{aligned}
\mathbf{V} &\in \mathbb{R}^{n\times d} \\
\mathbf{V}_i &= \text{Embedding}^{\boxed{1}}(w_i) && i = 1, \ldots, n && (6.77) \\
\mathbf{H} &\in \mathbb{R}^{n\times d} \\
\mathbf{H} &= \text{RNN}^{\boxed{2}}(\mathbf{V}). && && (6.78)
\end{aligned}$$

The encoder could just as easily be a Transformer encoder (Kitaev and Klein, 2018).

If this were a language model, you'd expect something like a softmax layer to predict the next word. But we want to predict a whole tree, so we have something like a softmax over trees:

$$P(\text{tree} \mid w) = \frac{\exp s(\text{tree})}{\sum_{\text{tree}'} \exp s(\text{tree}')} \tag{6.79}$$

where the summation over tree′ sums over all trees with w on their leaves, and $s(\cdot)$ is the score of a tree, which is the sum of the scores of the rules in it:

$$s(\text{tree}) = \sum_{(A\to\beta)\in\text{tree}} s(A \to {}_i\beta_j). \tag{6.80}$$

This $s(A \to {}_i\beta_j)$ is the score of rule $A \to \beta$, where A spans the substring $w_{i+1} \cdots w_j$. It is computed by a feedforward neural network. Define a new kind of module,

$$\text{LinearLayer}^{\boxed{\ell}}(\mathbf{x}) = \mathbf{W}^{\boxed{\ell}}\mathbf{x} + \mathbf{b}^{\boxed{\ell}}. \tag{6.81}$$

Then for each span (i, j), compute a *span encoding*

$$\mathbf{s}^{(i,j)} = \begin{bmatrix} \mathbf{H}_{i+1,:} \\ \mathbf{H}_{j,:} \end{bmatrix} \tag{6.82}$$

and then feed it through two layers:

$$\begin{aligned}
\mathbf{h}^{(i,j)} &\in \mathbb{R}^{d} \\
\mathbf{h}^{(i,j)} &= \tanh(\text{LinearLayer}^{\boxed{3}}(\mathbf{s}^{(i,j)})) && (6.83) \\
\mathbf{o}^{(i,j)} &\in \mathbb{R}^{|G|} \\
\mathbf{o}^{(i,j)} &= \text{LinearLayer}^{\boxed{4}}(\mathbf{h}^{(i,j)}). && (6.84)
\end{aligned}$$

If we number the rules of the grammar and rule $A \to \beta$ has index r, then the rule score $s(A \to {}_i\beta_j)$ is $\mathbf{o}_r^{(i,j)}$.

Two challenges remain. First, how do we use this model to search for the best

tree? This turns out to be a simple modification to the CKY algorithm.

$$\arg\max_{\text{tree}} P(\text{tree} \mid w) = \arg\max_{\text{tree}} \frac{\exp s(\text{tree})}{\sum_{\text{tree}'} \exp s(\text{tree}')} \tag{6.85}$$

$$= \arg\max_{\text{tree}} \exp s(\text{tree}) \tag{6.86}$$

$$= \arg\max_{\text{tree}} \prod_{(A\rightarrow\beta)\in\text{tree}} \exp s(A \rightarrow {}_i\beta_j). \tag{6.87}$$

This is the same problem that the Viterbi CKY algorithm solves, except we just change the weight of a rule from $P(A \rightarrow \beta)$ to $\exp s(A \rightarrow {}_i\beta_j)$. The weights don't sum to one like probabilities do, but nothing about CKY depends on whether the weights sum to one, so it will work fine.

The second challenge is, how do we compute $\sum_{\text{tree}'} \exp s(\text{tree}')$ in the denominator of (6.79)? This is needed to form the loss function during training. It is the sum of the weights of all trees with w on their leaves. This, too, is a simple modification to the CKY algorithm. It builds a table $\mathit{total}[i, j]$, and whenever two items X are added to the same cell $\mathit{total}[i, j]$, instead of keeping the best one, we add them. The pseudocode is as follows:

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: $\mathit{total}[0, n][S]$ is the log of the total weight of all parses of w

1: initialize $\mathit{total}[i, j][X] \leftarrow -\infty$ **for all** $0 \le i < j \le n, X \in N$
2: **for all** $i \leftarrow 1, \ldots, n$ **do**
3: $\quad a \leftarrow w_i$
4: $\quad$ **if** $(X \rightarrow a) \in R$ **then**
5: $\quad\quad \mathit{total}[i-1, i][X] \leftarrow \text{logaddexp}(\mathit{total}[i-1, i][X], s(X \rightarrow {}_{i-1}\, a\, {}_i))$
6: $\quad$ **end if**
7: **end for**
8: **for** $\ell \leftarrow 2, \ldots, n$ **do**
9: $\quad$ **for** $i \leftarrow 0, \ldots, n-\ell$ **do**
10: $\quad\quad j \leftarrow i + \ell$
11: $\quad\quad$ **for** $k \leftarrow i+1, \ldots, j-1$ **do**
12: $\quad\quad\quad$ **if** $(X \rightarrow YZ) \in R$ **then**
13: $\quad\quad\quad\quad p' \leftarrow s(X \rightarrow {}_i\, YZ\, {}_j) + \mathit{total}[i, k][Y] + \mathit{total}[k, j][Z]$
14: $\quad\quad\quad\quad \mathit{total}[i, j][X] \leftarrow \text{logaddexp}(\mathit{total}[i, j][X], p')$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: **end for**

The function logaddexp is defined as

$$\text{logaddexp}(x, y) = \log(\exp x + \exp y) \tag{6.88}$$

but is implemented in a way that avoids over/underflow.

Bibliography

Booth, Taylor L. and Richard A. Thompson (1973). "Applying Probability Measures to Abstract Languages". In: *IEEE Trans. Computers* C-22.5, pp. 442–450.

Choe, Do Kook and Eugene Charniak (Nov. 2016). "Parsing as Language Modeling". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, pp. 2331–2336. DOI: 10.18653/v1/D16-1257. URL: https://www.aclweb.org/anthology/D16-1257.

Durrett, Greg and Dan Klein (July 2015). "Neural CRF Parsing". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, pp. 302–312. DOI: 10.3115/v1/P15-1030. URL: https://www.aclweb.org/anthology/P15-1030.

Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley.

Kitaev, Nikita and Dan Klein (July 2018). "Constituency Parsing with a Self-Attentive Encoder". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, pp. 2676–2686. DOI: 10.18653/v1/P18-1249. URL: https://www.aclweb.org/anthology/P18-1249.

Stern, Mitchell, Jacob Andreas, and Dan Klein (July 2017). "A Minimal Span-Based Neural Constituency Parser". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, pp. 818–827. DOI: 10.18653/v1/P17-1076. URL: https://www.aclweb.org/anthology/P17-1076.

Stolcke, Andreas (1995). "An Efficient Probabilistic Context-Free Parsing Algorithm that Computes Prefix Probabilities". In: *Computational Linguistics* 21, pp. 165–201.

Vinyals, Oriol et al. (2015). "Grammar as a Foreign Language". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2015/file/277281aada22045c03945dcb2ca6f2ec-Paper.pdf.