Two problems with parser theory as developed in textbook. The definition of PARSER is

\[
\text{List[Word]} \Rightarrow \text{Stream}[(\text{List[Word]}, \text{Parsetree})]
\]

(more or less). A PARSER by definition takes a list of words \( L \) and returns all the analyses it can find, an analysis being a pair \((L_{\text{rem}}, T)\), where \( T \) is a parsetree for some prefix of \( L \), and \( L_{\text{rem}} \) is a list of the words after that prefix. The idea is that if you’re looking for a sequence of items, you look for the first, then look for the rest in the remaining words.

The two problems:

1. By postponing the discussion of information propagation around the Parsetree, we’re forced to retrofit the propagation machinery to the structure they invented at the beginning of the chapter.

2. By analyzing a PARSER as a particular function, we’re drawn to recursive descent as our primary idea. The left-recursive-rule problem then gums everything up.

Two parsing algorithms are discussed in Chapter 9 of CSFP: the one that tries all possible ways of a splitting a word string, which just seems to be searching the wrong space; and the more plausible version given above. Unfortunately, if we take such a function at face value, we run into the problem of left recursion. Any rule of the form

\[
P \to P P'
\]

will be interpreted thus by a function of type PARSER: Given a list of words \( L \), one way to find a prefix of \( L \) analyzable as a \( P \) is to find a prefix \( L_1 \) of \( L \) analyzable as a \( P \), followed by another string of words \( L_2 \) analyzable as a \( P' \). Put the parsetrees for \( L_1 \) and \( L_2 \) together into a tree \( T \) and return the pair \((L_3, T)\), where \( L_3 \) is the list of remaining words \( L_3 \) (i.e., \( L = L_1 ++ L_2 ++ L_3 \)).

This is a fine plan, except for step 1, which involves calling the parser with the same goal we started with: Find a prefix of \( L \) analyzable as a \( P \). This is the problem of left recursion, so called because right-recursive rules, of the form \( P \to P' P \), cause no problems. In a rule of this form, the
recursive step is to find a \( P' \) in the word list \( L \), then find a \( P \) in the words after that. The worst that can happen is the parser runs out of words.

Unfortunately, left recursion arises very naturally in natural-language grammar. The rule for adjuncts\(^1\) in English is

\[
X' \rightarrow X' \text{adjunct} \\
X' \rightarrow X \text{-adjunct} X'
\]

An example of the former (for a \( V' \)) is \( V' \rightarrow V' \text{adverb} \) (“ran quickly”). An example of the latter is \( V' \rightarrow \text{adverb} V' \) (“quickly hid”). The former is, of course, left-recursive.

The X-bar rule for English complements is also problematic, in a different way. Complements are phrases that words “demand,” in a sense. The verb “put,” in the basic sense involving willed motion to a destination, requires an NP and a prepositional phrase (PP) involving “on” or “in.” These complements are recorded in the lexicon entry for the word. The word is said to subcategorize for the complements.

The way we would like to write this in X-bar theory goes something like this:

\[
X[\text{subcats}] \rightarrow \text{word } w \text{ s.t. lexicon entry for } w \text{ classifies it as an } X \text{ requiring subcats} \\
X[C:s] \rightarrow X[C::Cs] C \text{ (where } C \text{ is a category and } Cs \text{ is a list of categories)} \\
X' \rightarrow X[]
\]

The bracket notation here is an intrusion from a blend of Haskell and logic programming. The brackets ([...]) are short for \( \text{List(...)} \) and the stuff inside the brackets is a kind of \text{case matching}.\(^2\) What we’ve put inside are the subcategories that we’re still looking for. The last rule says that we’ve found an \( X' \) when there are no subcategories left to look for. The first rule says we’ve found an \( X \) requiring subcategories \( \text{subcats} \) when the next word \( w \) is an \( X \), according to the lexicon, and it requires subcategories \( \text{subcats} \). It’s the rule in the middle that’s problematic. It’s basically of the form \( X \rightarrow X C \), and hence left-recursive.

Putting the adjunct rules and the complement rules together, we should get nice, theoretically motivated analyses like the following, for the VP in “She put the lamp on the table last night”:

\(^1\text{At this point I’m shifting to the convention that primes are to be read as “bar.”}\)

\(^2\text{In F-structure grammars, which we look at next, it’s natural to express constraints like this.}\)
VP ---- V' ---- Adj - "last night"
   |      |
Spec    V[]
   |      |
[tns past] V[PP_{on}] --- PP --- "on the table"
   |      |
    V[NP, PP_{on}] --- NP --- "the lamp"
   | "put"
   entry in lexicon
   with subcategories [NP, PP_{on}]

The problem is getting the parser to find these analyses, assuming it’s able to keep track of subcategories at all. Suppose the bottom rule causes the parser to look for a $V'$ by looking for a $V[]$, a $V$ that’s paid its debt to the subcategories. The center rule says we can find such a $V[]$ by finding $V[C]C$ (taking $Cs$ as $\text{List}[]$). And so forth. Unfortunately, we can apply it over and over again, generating $V'[C_1::C_2::\ldots::C_n::\text{Nil}]$ for all $n$. Granted, we can cut this off by stopping when $n$ is greater than the number of words in $L$, but the whole thing is a silly way to go about parsing. What we want is to (a) find the subcategories the next word expects; (b) see if they’re there.

We need a better parsing algorithm.