Today we moved from CFGs—a solution for representing English syntax—to CFG parsing, which lets us get from sentences to parse trees. A closely related problem is CFG **recognition**, which just tells us whether a parse tree exists and hence whether the sentence is grammatical with respect to our CFG.

The first two algorithms are based on the idea of parsing as a **search** problem. **Top-down** parsing starts with the start symbol S and tries to build the sentence. Recursive descent is a classic top-down parser. **Bottom-up** parsing starts with the sentence and tries to find an S. Shift-reduce is a classic bottom-up parser. These are both conceptually simple, but they have complementary problems. Top-down parsing will hallucinate all kinds of grammatical derivations that don't match the sentence. (They also can go into infinite loops over left recursion.) Bottom-up parsing will build all kinds of structures that can never make it in the context of the entire sentence. Both fall prey to the problem of doing the same work more than once, repeatedly considering the same partial parse in slightly different contexts, since search-driven parsers follow a **backtracking** strategy whenever a path fails. We'd like to be able to reuse earlier work when we can. Another problem is that the search space is huge, so the search may need huge amounts of memory.

Dynamic programming is a familiar solution to this kind of problem. The **CKY** algorithm is the simplest dynamic programming solution for CFG parsing. It requires that the grammar be in Chomsky normal form (all rules rewrite to either two nonterminals or one terminal). We can convert any CFG into this form by introducing new nonterminals and rules, and then "fix" the trees after parsing. The CKY equations to parse a sentence $\boldsymbol{w} = \langle w_1, w_2, \ldots, w_n \rangle$ are:

$$C[i-1, i, w_i] = \text{TRUE}$$

$$C[i-1, i, V] = \begin{cases} \text{TRUE} & \text{if } V \rightarrow w_i \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$C[i, j, V] = \begin{cases} \text{TRUE} & \text{if } \exists j, Y, Z \text{ such that } V \rightarrow Y Z \text{ and } C[i, k, Y] \text{ and } C[k, j, Z] \text{ and } i < k < j \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$\text{goal} = C[0, n, S]$$

This algorithm runs in $O(n^3)$ in the length of the sentence (n) and linearly in the size of the rule set (worst case is cubic in the number of nonterminals). CKY makes use of a data structure, called a **chart**, which keeps track of all the nonterminals that can cover each subsequence in the sentence and (if the tree is desired), backpointers to the smaller constituents that can be used to build each larger one.