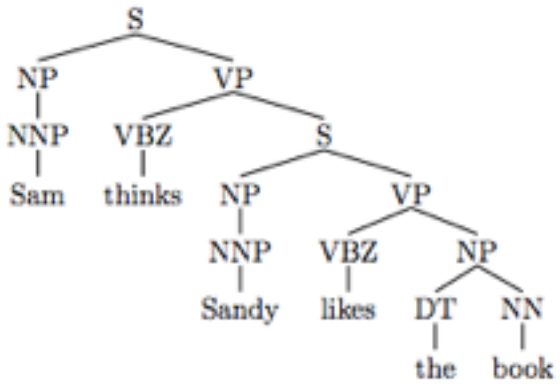
Sentence “Parsing” Application

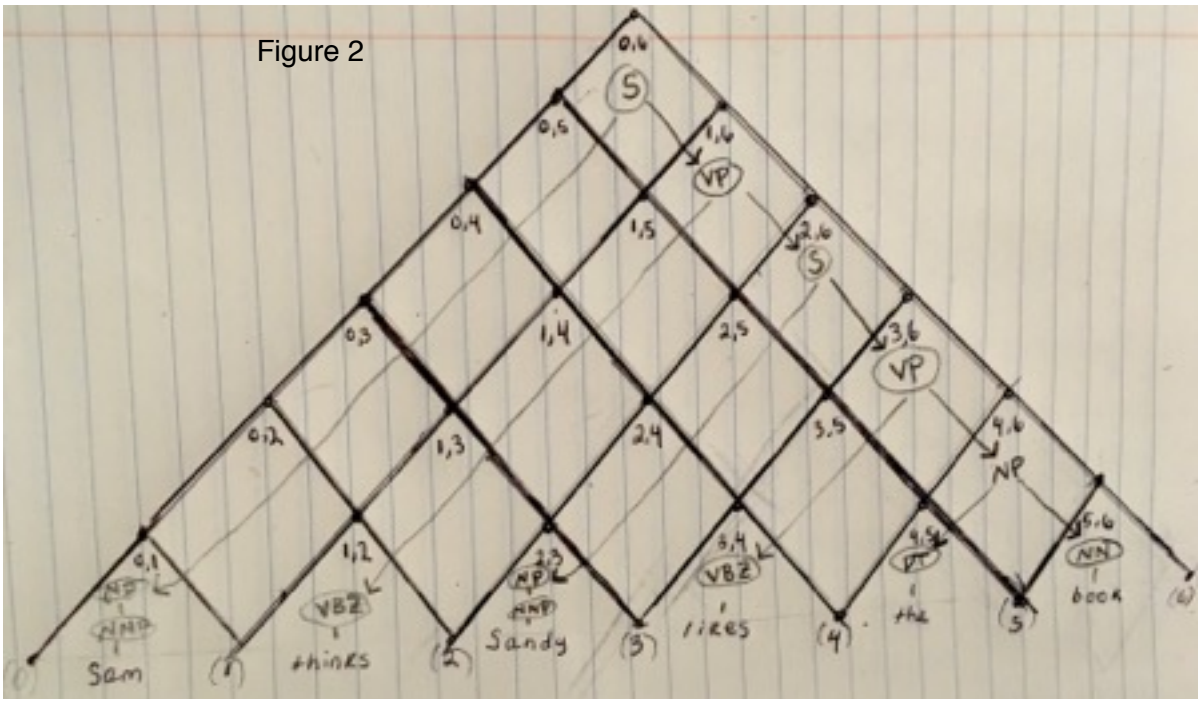
1. Overview

Figure 1



Sentences can be represented by *phrase structure trees* which show how words and phrases combine to form other phrases. Parsing, the processes of identifying this structure from a sentence, is used in machine translation to translate between languages with radically different word orders. Figure 1 represents the phrase structure tree for the utterance “Sam thinks Sandy likes the book.” As seen in the figure, the “leaf” or terminal nodes of the tree are the english words, and the non-terminal nodes are *constituents* spanning one or more terminal nodes. Every non-terminal node can be rewritten with a rule. For example, the top ’S' node is can be rewritten with the rule ‘S' —> ‘NP,’ ‘VP.’ The constituent directly under ‘S’, ‘NP,’ can be rewritten with the rule ‘NP’ —> ‘NNP,’ and ‘NNP’ can be rewritten with the rule ‘NNP’ —> ‘Sam.’ ‘Sam,’ a terminal, cannot be rewritten. I wrote an application that takes in the frequency of the use of each rule used in a training corpus of phrase structure trees, and returns the most likely phrase structure tree for every sentence in a testing corpus of thousands of sentences.

2. Data Structures Used

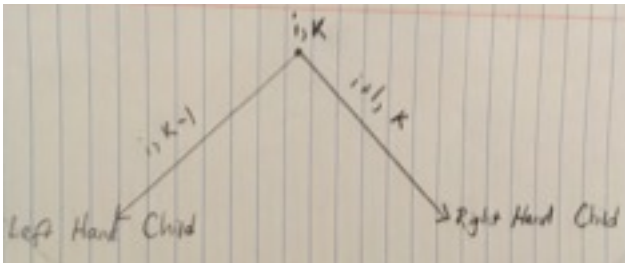


Sentence “Parsing” Application

It actually simplifies parsing to imagine the problem with “computer science” zero-based indexing for strings. Imagine the indices are in between the words, as in Figure 2. For a string of n words, indices can span anywhere from one to n words. The pyramid data structure in Figure 2 represents all possible combinations of spans for the sentence “Sam thinks Sandy likes the book.” Each pair of indices, stored as a “cell” in the data structure, represents a span. For example, the cell at (0,6) represents the span ‘Sandy likes the book.’ Each cell can hold multiple constituents, and each constituent has pointers to it’s children. For example, the top node ‘S’ in figures 1 and 2 has a pointer to its right child ‘NP’ and its left child ‘VP’.

3. Calculations

Figure 3



The parser algorithm fills in the pyramid from the bottom up. To add a constituent in cell i, k the algorithm looks at the constituents in left child ‘B’ stored at i, k-1, and right child ‘C’ stored at i + 1, k (see Figure 3). Then the algorithm looks for rules of the type ‘A’ —> ‘B’ + ‘C’, and if such a rule exists, it can add new constituent ‘A’ with ‘B’ for a left hand child and ‘C’ for a right hand child. The probability that ‘A’ will be rewritten as ‘B’ + ‘C’ equals the number of times rule ‘A’ —> ‘B’ + ‘C’ occurs in our training corpus divided by the number of times A is ever rewritten (number of times ‘A’—> anything) in our training corpus. To calculate the probability of a new constituent, the algorithm multiples the probability of ‘A’—> ‘B’ + ‘C’ by the probability of constituent ‘B’ and the probability of constituent ‘C’. Each constituent knows it’s probability and each cell knows the combination of left and right children that maximizes the probability for that cell. When we finish filing out the pyramid, we can ask the top cell for it’s best constituent, and preform a breath first search, following the child pointers to reproduce the best path. This path represents the most likely phrase structure diagram of the sentence.

4. Notes on Uniary/ Binary rules

To create the tree digram in Figure 1, we need two types of rules: *uniary* rules that produce one result, like ‘NP’ —> ‘NNP’ and *binary* rules that produce two results like ’S'—> ‘NP’ + ‘VP.’ Uniary rules can really slow the algorithm down. Let’s say we have a constituent in a cell labeled ‘NNP.’ We have to go through all the uniary rules of type ‘anything’ —> NNP to see if we can add new constituents. The uniary rule NP —> NNP allows us to add NP as a new constituent, with the old constituent ‘NNP’ as its child. My initial algorithm would start the process of looking through uniary rules all over again at this point, because uniary rules of the type ‘anything’ —> ‘NP’ would now generate valid new constituents, and we haven’t looked for those yet. However, this takes too long. My final algorithm relies on the knowledge that if a uniary rule doesn’t produce a new constituent for a given old constituent on the first iteration, no uniary rule will ever produce anything from that constituent. This means, I only have to re-check uniary rules for newly added constituents. This is easily accomplished with a “newly added” list that gets rewritten with each iteration, and saves a lot of run time.