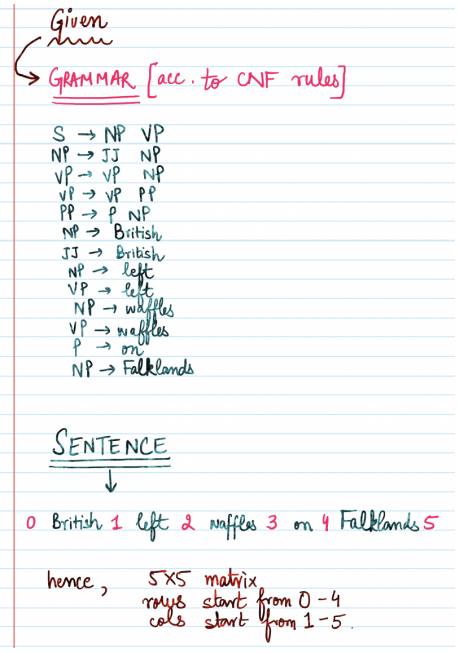
KUSHAGRA SETH
2005986
kuseth@ucsc.edu

Homework Assignment 2
NLP-202

Prof: Jeff Flanigan

1 Filling Out The Chart By Hand



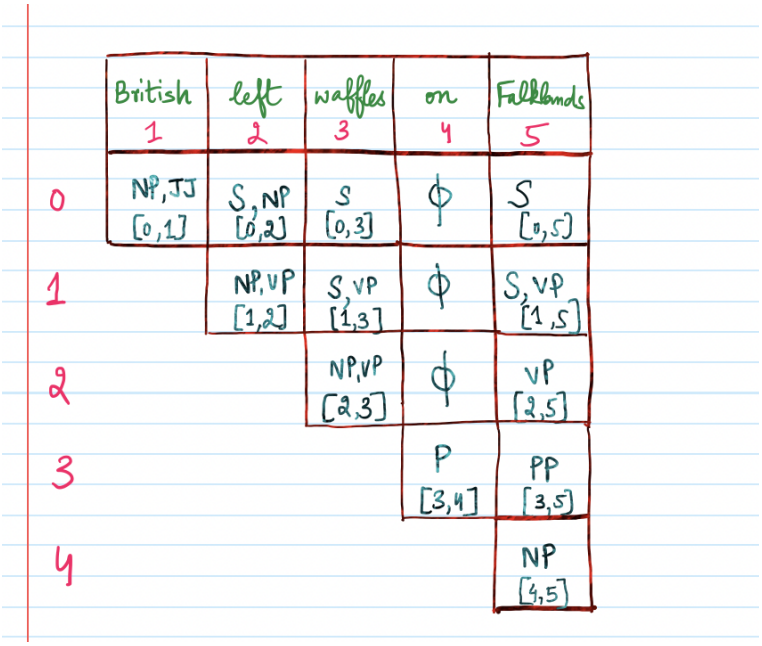


Figure 1: CKY Algorithm Chart filled by hand

2 Programming: CKY Algorithm

The **Cocke–Younger–Kasami (CKY) Algorithm** is a dynamic programming algorithm used to parse a string in a Context-Free Grammar (CFG) and check if the grammar can generate it.
The function $cky(words, grammar)$ implements the CKY algorithm in Python. The input to the function is a list of words and a context-free grammar(CFG). The function returns a table of all possible parses for the sentence, represented as a list of lists of non-terminals.

CKY Table:

	British	left	waffles	on	Falklands
0	{'NP', 'JJ'}	{'NP', 'S'}	{'S'}	set()	{'S'}
1		{'NP', 'VP'}	{'S', 'VP'}	set()	{'S', 'VP'}
2			{'NP', 'VP'}	set()	{'VP'}
3				{'P'}	{'PP'}
4					{'NP'}

Figure 2: CKY Algorithm Table

The function initializes the table with empty sets of non-terminals for each cell. Then, it fills the table's diagonal with the non-terminals that can generate each terminal in the input sentence using the $grammar.bottom\_up\_map$ dictionary to look up the non-terminals for each terminal.
Next, the function iterates over the off-diagonal cells of the table, considering all possible splits of the substring into two parts (i and j) and combining the non-terminals that can generate each part. The function uses the Cartesian product of the non-terminals in the two parts to generate all possible combinations of non-terminals and checks if each combination is in the grammar. If a combination is in the grammar, its corresponding non-terminal is added to the set of non-terminals for the cell.

Finally, the function returns the filled table, representing all possible input sentence parses. The code prints the CKY table using the $tabulate$ library in a user-friendly format. The headers of the table show the individual words of the input sentence, and each cell shows the syntactic categories that can produce the corresponding substring of words.

In summary, the CKY algorithm is a robust parsing algorithm that can efficiently parse context-free grammar. The cky function implements the CKY algorithm and returns a table of all possible parses for a given input sentence and grammar.

3 Programming: Weighted CKY Algorithm

The **Weighted CKY Algorithm** is also a parsing algorithm that utilizes dynamic programming to build parse trees for a given sentence based on a probabilistic context-free grammar. It is an extension of the traditional CKY algorithm. Still, instead of simply determining if a grammar generates a string, it computes the most likely parse trees for a sentence given a probabilistic grammar.

Weighted CKY Table:

	astronomers	saw	stars	with	ears
0	{('NP', 0.4)}	set()	{('S', 0.05040000000000001)}	set()	{('S', 0.0027216000000000015), ('S', 0.0036288000000000015)}
1		{('NP', 0.04000000000000001), ('V', 1.0)}	{('VP', 0.12600000000000003)}	set()	{('VP', 0.009072000000000004), ('VP', 0.006804000000000004)}
2			{('NP', 0.18000000000000002)}	set()	{('NP', 0.012960000000000006)}
3				{('P', 1.0)}	{('PP', 0.18000000000000002)}
4					{('NP', 0.18000000000000002)}

Figure 3: Weighted CKY Algorithm Table

The algorithm begins by creating an empty table of size N x N, where N is the length of the input sentence. Each cell of the table represents a substring of the input sentence. The cell stores a set of tuples, where each tuple represents a constituent or sub-constituent of the input sentence. Each tuple consists of a constituent label (e.g., NP, VP, S) and a parse tree node associated with that label.

The $weighted\_cky(words, grammar)$ function implements the Weighted CKY algorithm. The input to the process is a list of words representing the input sentence and a Grammar object representing the probabilistic context-free grammar used for parsing. The function returns a tuple consisting of a table of parse constituents and a list of parse trees.
The function initializes the N x N table by creating an empty list of lists of cells. Each cell is initially an empty set. It then initializes an empty list of parse trees.
The function then proceeds to fill the diagonal cells of the table with the non-terminal constituents

that generate each corresponding word in the input sentence. For each non-terminal that produces a word, the function creates a parse tree node associated with that non-terminal and adds it to the corresponding cell in the table.
The function then proceeds to fill in the remaining cells in a bottom-up manner. For each cell, the function considers all possible combinations of constituents to generate the substring represented by that cell. It then computes the probability of each possible sub-constituent using the probabilities associated with the rules in the grammar. The function stores the most likely sub-constituent for each cell in the table.
Finally, the function returns the table of parse constituents and a list of parse trees. The table of parse constituents provides the most likely constituents that generate the input sentence, whereas the list of parse trees provides the most likely parse trees for the input sentence. Only parse trees with the root node labeled "S" are added to the list of parse trees.

The tabulate function from the tabulate library to pretty-print the parse table generated by the $weighted\_cky$ function.

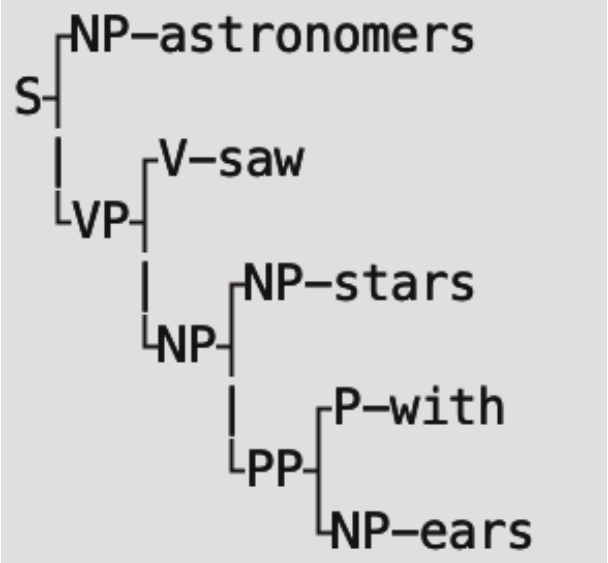


Figure 4: Most Probable Parse Tree

4 Programming: Sum Over All Parse Trees

$marginalize\_prob(trees)$ prints out the probability of the input sentence by marginalizing over all possible parse trees generated by the $weighted\_cky$ function. We use the stored probabilities of the non-terminals in each cell of the chart and then sum over all possible parse trees to compute the probability of the sentence.

```
Probability of Sentence: 0.006350400000000003
```

Figure 5: Probability of Input Sentence