**Natural Language Processing**

*Spring 2023*

Individual Programming Project #2

For this project, you will implement a parser. Your program will prompt the user to enter the name of a text file specifying a context-free grammar (CFG) in Chomsky normal form (CNF). Various assumptions about the format of the CFG below will be specified below. The CFG will represent a phrase structure grammar, a.k.a. constituency grammar, for English. After reading the grammar, your program will allow the user to enter sentences, one at a time. For each sentence, your program will parse the sentence using the Cocke-Kasami-Younger (CKY) algorithm. If there are no valid parses, your program should output "NO VALID PARSES". If there are valid parses, all valid parses should be indicated using a format that will be explained below.

Your program may assume that the CFG specifying the grammar is already in CNF. Each line of the file will contain a single valid rule in one of two forms: A --> B C, where A, B, and C are non-terminals, or A --> w, where A is a non-terminal and w is a terminal (i.e., a word). All non-terminals will start with a capital letter or an underscore ('\_'). All terminals will start with a lowercase letter or digit. All other characters in non-terminals will be alphanumeric, and all other characters in terminals will be lowercase letters or digits. There will be no OR symbols. Lexical rules will be part of the grammar, each on its own line. The start symbol will always be S (although S may not be used on the first line of the file). Every line in the file will be valid.

Of course, when creating a grammar, it would be nice if you could start with a CFG that is not in CNF. I am providing you with a program I wrote that converts a general CFG to CNF. I wrote the program using Python 3; the program does not rely on any special libraries. The program assumes that all non-terminals start with a capital letter (not an underscore) and all terminals start with a lowercase letter or digit. (The dummy non-terminals that it creates will start with underscores, guaranteeing that there are no conflicting names.) My program allows OR symbols to be used in the original CFG, but the CFG in CNF that it creates will not use them. Optional constituents (surrounded by parentheses) are not allowed in either grammar. My program ignores blank lines (containing only whitespace) and lines starting with '#' (which can be used to indicate comments). Note that *your program does NOT have to indicate parses according to the original CFG*, only according to the CFG after conversion to CNF, which is what it will load and process.

On the class website, I will post my program to convert a general CFG to CNF, a sample grammar (a general CFG), and the grammar after conversion to CNF. To create this sample grammar, I started with the L1 grammar from the textbook, and then I added a few additional grammar rules and some extra words to the lexicon. I also added some invalid rules to the general CFG to test the syntax error checking of my conversion program, but I do not guarantee that it will detect all possible types of errors. If my program detects any sort of syntax error in a line from the original CFG, it displays an error message and ignores the entire line; my program then continues to process the remaining lines of the CFG. (I also tested my program on a more complex grammar, but complex in terms of containing various funky things that would never likely occur in a real, natural language grammar.) When I test your programs, I'll use a different, slightly larger grammar with some additional grammar rules and words. If you try out my conversion program on your own grammars, please let me know if you discover any bugs.

Again, your program will only be tested using a CFG in CNF. After reading the grammar in CNF, your program should allow users to enter sentences, one at a time. You may assume that the user will only enter sentences with words complying to our programs' assumptions (i.e., all characters will be lowercase letters or digits). Words will be separated by single spaces. So, one example of a valid input sentence would be: "i book the flight from houston".

If you want to allow the user to type sentences such as "I book the flight from Houston.", your program may include a pre-processing component that strips punctuation and converts all letters to lower case, but this is not required.

Note that we are assuming that grammars will NOT recognize or handle punctuation. Since we are assuming that all terminal symbols consist of only lowercase letters and digits, something like a period would not be a valid terminal symbol.

After processing each sentence, the program should either output "NO VALID PARSES" (only if appropriate, of course), or it should display all valid parses using a format that will soon be described. For example, for the sample sentence mentioned above, "i book the flight from houston", using the provided sample grammar (after conversion to CNF), there would be three valid parses.

If the user types "quit" as a single word, all in lowercase, the program should end.

For each sentence entered by the user, the program should use the textual format known as bracketed notation to output all the valid parses. There is an example in Equation 17.1, on page 361 of the current, on-line draft of the textbook (dated January 7, 2023). (In class, we briefly looked at similar notations when we covered phrase structure grammars; e.g., the notations demonstrated in Figure 17.5 that are used by the Penn Treebank. These are not the formats I am asking you to use for the project, but the format shown on the left of that figure is similar to bracketed notation. The format shown on the right would be more common if a POS tagger was applied before the parser, but we are assuming that lexical rules are part of the grammar.)

Using bracketed notation, each constituent (i.e., type of phrase or POS) is enclosed in brackets, starting with the type of constituent followed by the sequence of sub-constituents. Examples of bracketed notation will be shown in two sample runs on the next page.

The program should also prompt the user, giving them the option to display textual parse trees, in addition to bracketed notation, for each valid parse. The format is actually very similar, but each new constituent should begin on its own line, indented the appropriate distance to make the grammar rules clear. (My program uses three spaces per level of indentation.) An examples of a textual parse tree will be shown in the second sample run on the next page.

After all the correct parses have been displayed using bracketed notation, optionally also with textual parse trees, the total number of valid parses should be displayed. (Please display this information after all the parses are displayed, not before.)

Given the requirements state above, here is an example sample run of the program:

python3 CKY\_Parse.py sampleGrammar.cnf

Loading grammar...

Do you want textual parse trees to be displayed (y/n)?: n

Enter a sentence: i book the flight through houston

VALID SENTENCE

Valid parse #1:

[S [NP i] [VP [Verb book] [NP [Det the] [Nominal [Nominal flight] [PP [Preposition through] [NP houston]]]]]]

Valid parse #2:

[S [NP i] [VP [VP [Verb book] [NP [Det the] [Nominal flight]]] [PP [Preposition through] [NP houston]]]]

Valid parse #3:

[S [NP i] [VP [\_Dummy2 [Verb book] [NP [Det the] [Nominal flight]]] [PP [Preposition through] [NP houston]]]]

Number of valid parses: 3

Enter a sentence: quit

Goodbye!

Next, we will look at a sample run **including textual parse trees**. I am choosing a sentence with just one valid parse here; if there were more than one, a textual parse tree would be displayed for each valid parse after the bracketed notation. Note that this input sentence is a question, but I am leaving out punctuation to conform to our rules for input. Again, you can optionally have your program strip punctuation and/or convert words to lowercase, but that is not required.

python3 CKY\_Parse.py sampleGrammar.cnf

Loading grammar...

Do you want textual parse trees to be displayed (y/n)?: y

Enter a sentence: does the flight fly

VALID SENTENCE

Valid parse #1:

[S [\_Dummy3 [Aux does] [NP [Det the] [Nominal flight]]] [VP fly]]

[S

[\_Dummy3

[Aux does]

[NP

[Det the]

[Nominal flight]

]

]

[VP fly]

]

Number of valid parses: 1

Enter a sentence: quit

Goodbye!

As an example of another grammar, one that you can use to test your program on cases with many valid parses, here is one that I came up with (note that it is already in CNF):  
  
S --> S S  
S --> w  
  
If you use this grammar and enter N consecutive w's (e.g., "w w w w w w w w w w w w"), the number of correct parses turns out to be the Nth Catalan number. When I tested my program on 12 w's, it correctly finds and displays 58,786 parses!

Finally, I am allowing one optional simplification, with a penalty: For a maximum possible grade of 80 (i.e., a loss of 20 points), you may implement a recognizer instead of a full parser. If you choose this option, your program should always output either "NO VALID PARSES" or "VALID SENTENCE". If you choose to take this simplification, your program will not be displaying bracketed notation or textual parse trees.

E-mail your project to CarlSable.Cooper@gmail.com (I'm using my Cooper Gmail account to avoid problems receiving Python files). The program is due the night of Tuesday, April 11, before midnight.