Name: Goh Xin Yee
Student ID: 20093715

Comparing RE approach and NLTK approach

Generally, NLTK is more compact and efficient for Lexical Analysis than RE. In terms of runtime analysis, my RE approach is $O(n^3)$ while my NLTK approach is $O(n^2)$, which is faster.

For defining the pattern of each token type, RE is more flexible than NLTK as NLTK's *RegexpTokenizer()* can't contain parentheses while RE's *compile()* can, hence we can easily group regular expressions.

For splitting target strings into lexemes, NLTK is easier to implement than RE, as NLTK has a built-in *tokenize()* function which extracts and returns a list of elements that match the patterns. Contrarily, RE's *split()* uses the matched elements as the separators and the resulting list won't contain the matched elements. Also, unlike *tokenize*(), *split()* doesn't automatically ignore space or empty values, causing the lexeme list to have unnecessary elements and generally not "clean". Therefore, in my first approach, I have applied more of a manual splitting method, which looks more complex.

However, my first approach with manual splitting can identify invalid elements like "**78sgsg**" while NLTK can't because *tokenize()* will identify them as "**78**", "**sgsg**" (shown in ***testfile3.py*)**. But both approaches can handle invalid symbols and operators (shown in ***testfile2.py***)

For identifying the token type of each lexeme, NLTK's *RegexpTagger()* is much more efficient than RE *match(), search(),* and *findall()*. *RegexpTagger()* allows us to group all token patterns together and identify the token types of all lexemes easily in one go. There is no need to apply the "for" loop to check the lexemes one by one, like when using RE's *match()*. Additionally, *match()* has its weak points where it will return something (as TRUE) as long as the target string matches the pattern initially, even not as a whole. As such, the defined pattern needs to be really specific when using *match()*.

Both approaches are sensitive to tiny mistakes, like misspelling or missing a metacharacter. The RE patterns might look more complex and confusing to readers, as there are too many different symbols representing different meanings. Meanwhile, NLTK will be slightly easier to understand especially when using its built-in functions like *word_tokenize()* and *sent_tokenize()*. Nevertheless, once the conventions are understood, both methods are straightforward to apply for lexical analysis. RE rules are universal and can be applied anywhere while NLTK is only available in Python as it is a Python package.

Reference

Code faster with line-of-code completions, cloudless processing. Kite. (n.d.). Retrieved October 18, 2021, from https://www.kite.com/python/docs/nltk.RegexpTagger

Enthusiast, byV. G. O. S. (2013, March 14). *Groovy : Tokenize() vs Split()*. TO THE NEW BLOG. Retrieved October 14, 2021, from https://www.tothenew.com/blog/groovy-tokenize-vs-split/

Jablonski, J. (2021, May 5). *Natural language processing with python's NLTK package*. Real Python. Retrieved October 14, 2021, from https://realpython.com/nltk-nlp-python/

RegexpTokenizer. Kite. (n.d.). Retrieved October 15, 2021, from https://www.kite.com/python/docs/nltk.RegexpTokenizer