1.Overview

This is the project package for 2nd assignment of EMNLP, which generates results of converting phrase structures (syntax trees) in 'ctb.bracketed' file to dependency trees represented in CoNLL format.

Run the Transform.py to get the answers printed on the terminal, and redirect the output to get a result file.

The result get score of ***93.10 (malt)*** at the online test.

2.General Methods

There are three main steps to get the results:

2.1
Extract all the production rules from the syntax trees whose corresponding dependency tree are shown in the example answers file. Then determine the head word among child words of each production rule according to the dependency trees. These head rules are saved for transforming tasks in following steps

2.2
Transform all the syntax trees to dependency trees according to the head rules we know from 1st step. But there are some strange production rules that we don't know. When the program can't decide with child word to chose to be the head word, it will print the current production rule and wait for me to input the answer. The input is a number indicating the index of head child in children list. When a new rule is known from human(me) input, it will be saved for further use.

2.3
Redo the 2nd step again, with the aggregated set of head rules. This time the program will not stopped by any unknown circumstances, and the results will be automatically produced.

3.Algorithms

3.1 Extracting head rules from samples

There we have a syntax tree ST and its corresponding dependency tree DT.
When using head rules and ST to generate DT, we shift the head child node to the father position, from leave nodes. Head nodes are at higher position to their peers, in another word, shorter depth in the tree.
In a reversed view, when inferring the head rules from ST and DT, the head child is the one have the shortest depth in ST. For a node whose children are all leaves, this statement is literally true. For any node that is not a leaf, its 'depth' in ST is the value of the shortest depth among it children. Because when we do conversion from ST to DT, the node represent a non-leaf node is the head word of head words (of head words …). Suppose the children's depth are the depth of leaves that is the overall heads of the node, then current father node's depth should be defined as that shortest child depth, which is the depth of father's overall head leaf node.
This is a recursive definition, which can be represent in the following pseudo code:

```
define depth (node):
    if node is leaf
    then return depth_in_ST(node)
    else return min (map (depth, children_of(node)))
```

3.2 Saving and Quarrying Head Rules

The head rules are saved in a list, a *head rules list.* Each component of head rules list is also a list, which saves head rules of for a specific father node. The first element of a *sublist* is the father node's tag, the following components are lists of children of each production rule, together with the head child's index in list.

head rule list:

[[father1, [x1, [child1, child2, child3 …]], [x2, [child1, child2, child3 …]] …],
 [father2, [x1, [child1, child2, child3 …]], [x2, [child1, child2, child3 …]] …],
 [father3, [x1, [child1, child2, child3 …]], [x2, [child1, child2, child3 …]] …],
……]

When quarrying head rules, first the program search a sublist with father's tag exactly matches the target father's tag. Then find a matched children list. There are always lots of variations in the children list so if we only look for the exactly same list, it will frequently fail. When matching child tags, the program only match their prefix, which indicate the major function of a word token. But this is also not viable due to the number of tasks and the variability of production rules.
If program fails to find a matching production rule, it will then look for a child list whose head child is included in the target child list because there usually isn't lots of choices of head child for a father node.
If program fails to find a matched father tag, it will search the head rule list again for a father tag match the target father tag in terms of prefix, then do the similar thing.
After all above, if a matched production rule is still not found, the program will print out the current target production rule and let user(me) to input the head child index.

The list is saved in a file and loaded from file using *pickle* methods provided for python. There are two versions of the saved list file, 'Head,list' the head rule list simply extracted from samples, 'FinalHead,list' aggregates the user's(my) input which covers all cases in the test file.