Task #3 part 2
Implementing the CKY algorithm

Thuong-Hai Pham

April 13, 2017

1 Source code overview

Beside **main()** function, which reads the grammar, input file and does preliminary checking, the main part of our program can be divided into two parts: **parse()** (parsing) and **back_track()** (backtracking).

1.1 parse()

As a tradition to Dynamic Programming algorithm, we need a table to memorise previous results. In this program, we use a 2D table:

```
f[i][j] = {label: [(k, prod)...]...}
```

in which:

- **i, j** denote the sentence segment that our current cell coverages (from word i to word $j-1$)
- Each cell maintains list of key-value pair **label: [(k, prod)...]** (for backtracking purpose)

- **label** is the current node label (i.e. left-hand side (lhs) of the production rule), for faster search from parent cells
- **k** denotes the separation index, that divided this current segment into two sub-segments (i, k) and (k, j)
- **prod** stores the production rule that does the splitting (with labels of the two child nodes, or one child node if k is None). This is required because a triplet (i, k, j) can be produced by different production rules (composed by various child node labels)

Having our table, we first fill it with production rules lead to a terminal. To be more specific, we fill all $f[i][i+1]$ cells ($i < l$: length of the input sentence). Then, we follow these steps:

1. Generate all triplet (i, k, j), $(i < k < j)$
2. Search for production rules that matches our condition defined by CKY algorithm
3. Add new valid separation index (k) and production rule to current cell
4. Until we reach $f[0][l]$

1.2 back_track()

In backtracking part, we implement **go()** function that traverses through the result table in Depth-First-Search approach.

1. Start with (0, l, grammar.start) (the only valid top node)
2. If the node can not be divided (k is None, production: Non-terminal ->terminal), then return a single tree: Non-terminal ->terminal.
3. else, get two lists of left and right possible sub-trees by traversing to left, right node, consequently, then, return a Cartesian product of those two lists.

2 Problems and Solutions

Grammar provided is not minimised, some rules are duplicated. As a consequence, the output has some identical trees.

```
Nom -> X15 Nom
Nom -> Nom PP        # duplicated
Nom -> Nom NP
Nom -> Adj Nom
Nom -> Nom PP        # duplicated
Nom -> ADJP Nom
```

To solve this problem, when querying production rules from grammar, we need to remove duplicated rule by using "set" as below

```
for _prod in set(grammar.productions(rhs=lhs_1)):
```

instead of

```
for _prod in grammar.productions(rhs=lhs_1):
```