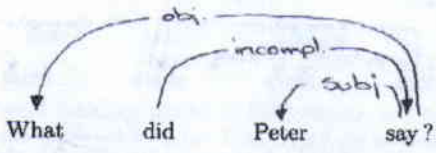
Question 1. Dependency trees

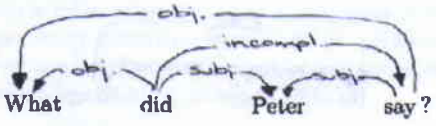
Sentence 1 - What did Peter say ?

If we parse this sentence using Hudson's dependency grammar but without structure sharing, we get the following parse.



This parse tree is actually not admissible, because the arrow going from *say* to *what* violates the adjacency principle. More specifically, the head of *say*, namely *did*, is not allowed to appear between *say* and its subordinate *what*; the "preliminary" adjacency principle (without structure sharing) states that every word between D (in this case *what*) and H (in this case *say*) must be a subordinate of H. *did* violates this constraint here.

Introducing structure sharing rules yields the following parse :



did is now the shared head of *what* and *say*, and *Peter* has also been made the subject of *did*, so that in effect the grammatical relationships around the verb are now shared by the verb's two words. These additions are dictated by the two following structure-sharing rules :

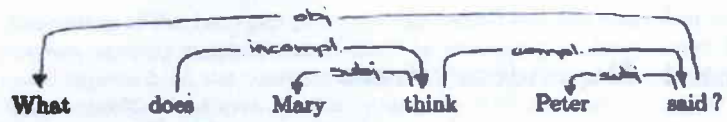
1. subject of incomplement of *word* = subject of it
2. object of incomplement of *word* = object of it

The first rule is directly taken from Hudson's English Word Grammar, the second one is simply its symmetrical complement. They attach the subject and object of the incomplement (in this case *say*) to its head (*did*).

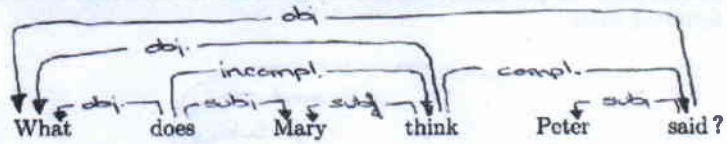
We can now use the updated definition of adjacency, stipulating that two words D and H such that H is the head of D are adjacent if every word between them is a subordinate of H or of a mutual head of D and H. This new definition, along with the new parse tree updated with structure sharing, allows us to form a valid and admissible parsing for the sentence.

Sentence 2 - What does Mary think Peter said ?

Once again, if we simply parse this sentence without using structure sharing, we get something like the following, which violates the adjacency principle (because *said* is the head of *what* but they are not adjacent, since all three words *does Mary think* are not subordinates of *said*).



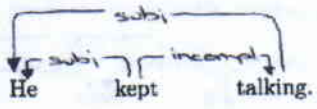
Applying the same two structure sharing rules as previously, we get the following parse tree, which satisfies the adjacency principle through structure sharing.

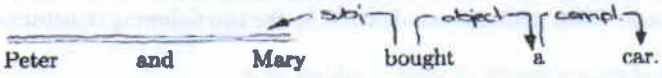


The analysis here is the same as for the first sentence - applying the structure sharing rules has made *does* a shared head of which all other words now are subordinates, thus satisfying the adjacency principle.

Question 2. Mapping from DG to PSG

Let us consider the following DG analyses :



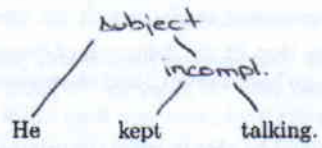


The first one is a pretty straightfoward parse tree. The second use uses the same technique Hudson uses in English Word Grammar to deal with conjunctions, namely grouping the coordinated nouns together to form one "word" that functions as a unit.

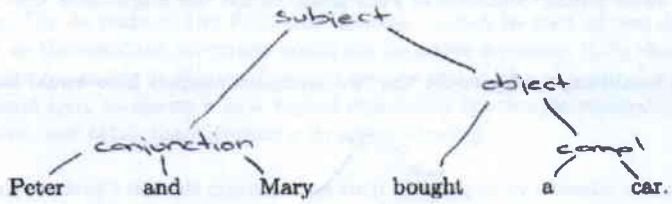
We would now like to devise an algorithm that would convert these DG tree relations to phrases structure rules. [1]

[1] For the scope of this exercise we will focus on the two sentences being examined here. There is little doubt that numerous new examples could be brought forward that would contradict facts assumed here to be true, or that could steer the discussion in a completely different direction. Considering the relative ignorance of this student, however, a thorough analysis of the matter is beyond the scope of this work.

Simply converting from one notation to another can be done rather simply if one merely wishes to use the formalism's basic structuring and does not care to follow the conventions that usually come with it. For instance, the first sentence could be expressed by the following PSG tree :



and the second sentence by this one.



These trees do use the basic building blocks of PSG theory, namely the grouping of words into blocks that are further grouped together following a set of predefined production rules. For instance one could write a grammar containing a rule such as

```
Subj --> <he> Incomplement
```

Such a rule could be fed to a PSG grammar parser and be used to build a PSG tree. The parser program should have no problem executing it. However it is obvious that in terms of grammatical analysis, we are not building very logical blocks here. A parser program will happily execute any grammar, regardless of the actual meaning the names given to nonterminal symbols might have, but to us humans it doesn't make much sense to state that as a general rule subjects are composed of "he" (or any personnal pronoun, if we care to make the rule a little more general than as it is written here) and an incomplement. Incomplements are a dependency grammar concept and do not fit within the usual approach PSGs take on things. They are particularily absurd as a constituent name. But still, such a grammar definition fits the basic building blocks of PSG.

Writing an algorithm that converts from DGs to PSGs is then pretty straightforward if one does not care for the plausibility of the PSG grammar thereby generated. Such an algorithm, following the ideas presented by Collins in his 1996 paper (which we will discuss in the next section) would start at the head of the DG tree and, working its way down, build constituent blocks by following the dependency relationships. A rather informal description of the algorithm follows.

0. Before starting, give each node on the DG tree a label that is simply the word that node represents
1. Take a root of the dependency tree, i.e. a node that is not the subordinate of any other node; call that node H;
2. Select one of that node's dependents; call that node D, and call the label of the dependency relathionship between them L;
3. Write a PSG rule of the form

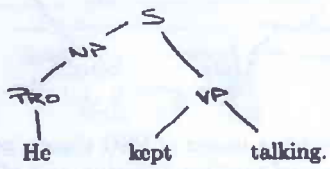
   ```
   L --> HL DL
   ```

   Where HL and DL are the labels or H and D, respectively.
   If your favourite flavour of PSG cares for heads, then HL would be the head of the L constituent.

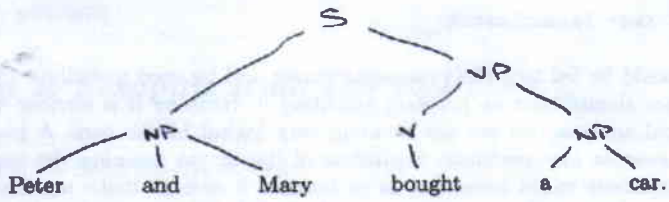
4. Set "L" as the label of both H and D
5. Remove dependency L from the DG tree
6. For every dependency L' going from D to some other node D', if there also exists a dependency going from H to D' then delete L' from the graph
7. If there are no more dependencies, stop
8. If there are no more nodes that fit the definition of "root" given at step 1, then delete the rule written at step 3, go back to step 2 and chose another of the node's dependents.

The resulting PSG grammar should be able to parse the original sentence and produce a PDg tree. As we have said, however, the resulting tree, which are represented above, although following the basic formal definition of PSG trees, do not use a grammar that makes much sense.

The trees we would expect to see for the two sentences studied here would look more like this :



for the first sentence and this :



for the second one. However reflexion has unfortunately led this student to believe that no simple algorithm would be able to convert from the above DG trees to these PSG trees. This reflexion is detailed in the following section.

Michael Collins's Dependency Model

Comparison with other DG models

One important difference between Collins' and Hudson's dependency grammars is that whereas Hudson defines a full set of rules based on basic DG principles and that define the whole grammar, Collins simply ports the rules of the Penn Treebank PSG over to a DG notation. Judging from the examples he gives, Collins' method seems to work well. However it suffers from the same problem as the algorithm outlined above in our discussion of question 2 (actually from the symetrical pendant of that problem) : the resulting dependency relationships that link words together have nothing to do with the type of relationships consistently used by Hudson and others, rather they are simply the direct mappings of the PS constituents. Just as "incomplement" and "subject" are not usually considered proper constituents in a PSG, "NP" and "PP" are not very interesting dependencies, actually they are not dependencies at all. The resulting structure still fits within the very basic formal definition of DGs in terms of how words are linked and how the structure is drawn, but uses that definition in ways that would not make much sense to its authors.

This direct mapping is hence simply a different notation for PSG, and serves quite well the purposes of Collins' work, that is better illustrating how word pairs and their linkings are

counted in his statistical analysis. The resulting structures do not constitute however what is usually considered a DG tree.

Mapping from PS trees to dependencies

As has been mentionned earlier, this student believes that these is no way to fully automate the translation from DG to PSGs (or vice versa, for that matter). There a several problems that arise when one tries to devise such a system ; here are some of them.

- PSG has no way of assigning two heads to a single constituent. If we look at the DG tree for the first sentence in question 2, we see that *he* has two heads, namely *kept* and *talking*. The *he* node in the PSG tree however cannot be part of two distinct constituents, as the resulting structure would not be a tree anymore. Such shared structures would hence, as far as this student can see, be lost in the process of conversion, and one would have to devise a new way of expressing the double relationship within the PSG tree, one other than Hudson's structure sharing.

  It is this student's humble opinion that such a discrepancy actually underlines a weakness in Hudson's theory. A programmer's intuition suggests here that the simple elegance of the DG grammar is being tarnished by the addition of such irregular features as structure sharing. In the programming world, simplicity based on a few simple base principles is usually the best way to build a robust program. Human language syntax is obviously very different from computer programming, but intuitively an experienced programmer would still tend to favour the simple balance of a few simple primitive principles that govern the whole system rather than the addition of more exceptional rules.

- Another, much more general and more complex problem that arises in translating from DG to PSG is simply that PSG and DG do not express the same things - a typical DG tree does not indicate how words are to be grouped into constituents, and a typical PS tree does not indicate the grammatical relationships between words. Such information might very well have been implied by the grammar engineer - for instance when the engineer writes a rule such as

  ```
  S --> NP VP
  ```

  it is understood that the NP stands as the subject of the sentence, and hence that we could see a "subject" dependency between it and the VP, or between the NP's head and the VP's head. This relationship however is not explicitly stated in a typical PSG.

  A similar situation exists when going from DGs to PSGs : when a DG grammar engineer writes a rule that makes a noun the head of preceding adjectives, it is understood that the two words unite to form what PSG grammarians call an NP. However this is not explicitly written in the grammar.

  How, then, could an automated conversion program go from one form to another without being given full knowledge of both grammars, the intentions motivating their rules, and how the information conveyed by one grammar relates to the information conveyed by the other. Clearly both grammars are based on the same input (in our case the English language and its structure), and there certainly are patterns that will emerge when comparing the output of a DG parser to that of a PSG parser. But, as far as this student can see, in order for a program to actually systematically convert from one form to another (and produce a result that DG grammarians would recognize as plausible, unlike Collins' DG trees), one would need to write a third "grammar", one that would map from the PS grammar to DG. The bulk of the work would then not be to write the conversion program itself but rather this "conversion grammar".