FNLP: Lab Session 4

Parsing

1 Aim

The aims of this lab session are to introduce simple parser models and Context Free Grammars (CFGs) and to explore and address the problem of syntactic ambiguity. By the end of this lab session, you should be able to:

- Access *parsed* treebank corpora provided in NLTK
- Step through the process of parsing a simple sentence using a simple CFG
- Understand why ambiguity can be a problem for parsing
- Implement simple models to resolve the problem of ambiguity
- Interpret a dependency parse and its relationship to a constituency parse

2 Running NLTK and Python Help

2.1 Running NLTK

NLTK is a Python module, and therefore must be run from within Python. To get started on DICE, type the following in a terminal window:

```
...: python
>>> import nltk
```

2.2 Python Help

Python contains an inbuilt help module that runs in an interactive mode. To run the interactive help, type:

```
>>> help()
```

To exit, press CTRL-d.

If you know the name of the module that you want to get help on, type:

```
>>> import <module_name>
>>> help(<module_name>)
```

To exit, type “q” (for “quit”).

If you know the name of the module **and** the method that you want to get help on, type:

```
>>> import <module_name>
>>> help(<module_name>.<method_name>)
```

To exit, type “q” (for “quit”).

3 Introduction

This lab will introduce parsed corpora, parsing new data, the problems of ambiguity and syntactic agreement, and using features to improve parsing.

Before continuing with this lab sheet, please download a copy of the lab template `lab4.py` for this lab *and* its accompanying helper files `cky.py` and `lab4_fix.py` from the FNLP course website. This template contains code that you should use as a starting point when attempting the exercises for this lab.

We'll also be running python interactively for the start of this lab. Please open a terminal window, run python and import nltk:

```
...: python
>>> import nltk
```

4 Parsed Corpora

NLTK contains a number of corpora with parse trees already available – that is, someone has already trained a parser and produced parse trees for each sentence in the corpus. The Penn Treebank is an example of a parsed corpus. It contains several sub-corpora including the Wall Street Journal (WSJ) corpus, and uses the following tagset: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Import the portion of the WSJ corpus provided in NLTK:

```
>>> from nltk.corpus import treebank
```

The treebank contains 199 articles:

```
>>> len(treebank.fileids())
```

Each article contains a number of sentences, each of which has been parsed. The parsed sentences are provided as a list of trees. Print out the first sentence of the first article:

```
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> print(t)
```

We can use nltk.tree to draw the trees, making them easier to read:

```
>>> from nltk import tree
>>> t.draw()
```

(Now close the window to return the prompt in the terminal window)

There are many more things that can be done with nltk.tree. The source code contains a "demo" section which outlines some of these things. If you are interested in learning more about nltk.tree please visit: http://www.nltk.org/_modules/nltk/tree.html

5 Approaches to parsing CFGs

Exercise 1

Look at the code for Exercise 1 in `lab4.py` and make sure that you understand what each step does. Ask the lab demonstrator if you are unsure.

Now uncomment the call to `test1`, load the file into python and follow the instructions overleaf:

- Look at the grammar rules that are printed to the terminal window
- Interactively step through the parsing of a sample sentence in the “app” window that pops up. Once the app window loads up:
  - Click on “Edit” in the menu bar and select “Edit Text”
  - Enter “I shot an elephant in my pyjamas” in the text box
  - Click on “OK”
  - Click the “Autostep” button and watch the parser step through
  - **Do you notice a problem?**
  - Click the “Autostep” button again to freeze the animation

Amend the grammar in Exercise 1 to replace the line: **NP** $\rightarrow$ **Det N** | **NP PP** | **’I’**
with: **NP** $\rightarrow$ **Det N** | **Det N PP** | **’I’**
Now re-run the parser using the new grammar.

Question: Can you explain why the parser is now able to parse the sentence?

Don’t reset or close the parser app just yet.

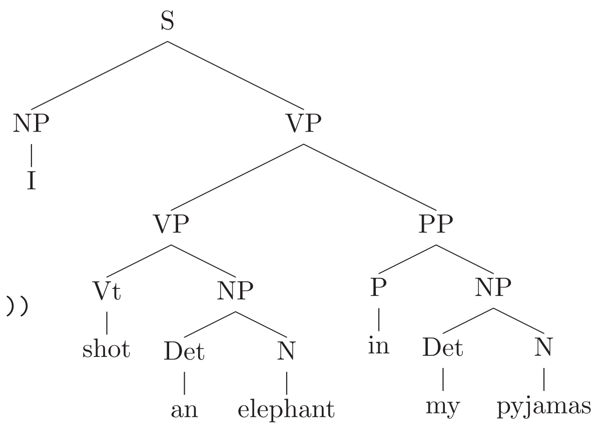
6 Ambiguity

But, getting the grammar right isn't our only problem. The sentence "I shot an elephant in my pyjamas" is also ambiguous. Consider who is wearing the pyjamas – the elephant or the person doing the shooting.

You can see this by forcing the demo parser to backtrack and look for another parse: just click autostep after it has found the first parse.

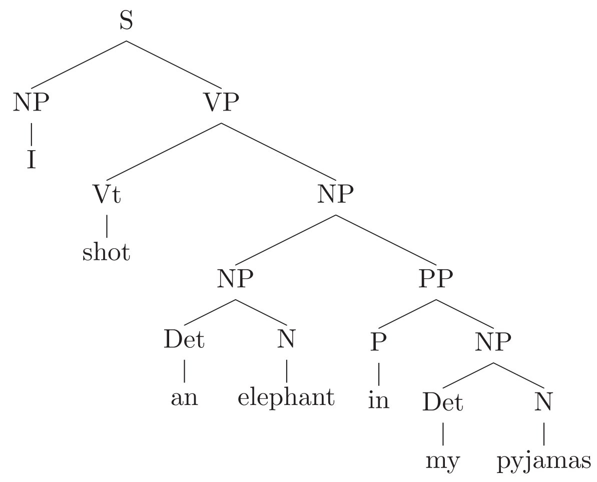
Here are the two possible parses of this sentence, depending on who is wearing the "pyjamas", and arising from structural ambiguity surrounding PP attachment:

```
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant)
        (PP (P in) (NP (Det my) (N pajamas))))))
```



and

```
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
```



Question: Which parse reflects the elephant wearing the pyjamas and which reflects the person doing the shooting wearing the pyjamas?

Now you can close the parser demo, and comment out the call to `test1`

7 CKY parsing

Exercise 2

A simple CKY parser is imported into `lab4.py` from `cky.py`. In this exercise you will explore how it implements the CKY algorithm.

Uncomment the first call to `test2_1` in `lab4.py` and reload. You'll see both the parse and the completed CKY matrix. Look at the grammar, the matrix and the result and satisfy yourself that you know where each entry in the matrix comes from, and where the backpointers must be to allow the parse to be reconstructed.

Now try the second call to `test2_1` so that the actions associated with each cell are shown. Again, inspect this and see that it confirms your earlier understanding.

Finally, try `test2_2` to go back to our ambiguous sentence. You will have to close the tree displays to get from one tree to the next, and to get back to the Python interpreter. You can see the underlying matrix by doing:

```
>>> chart.pprint()
```

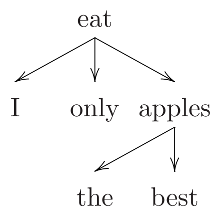
You can go on to try other inputs by doing:

```
chart.parse(tokenise("...")[,True])
```

8 Dependencies

Dependency parsing will be covered in depth in the lecture on 7 March 2017.

All the parses we've seen so far are constituency trees. An alternative way of representing syntactic structure is with a **dependency tree**, in which nodes are words in the sentence and edges represent head-modifier relationships. For instance, an *unlabeled* dependency parse of *I only eat the best apples*:



The overall head of the sentence is typically the main verb—in this case, *eat*. Heads are connected by edges to their direct modifiers or arguments. Edges can further be decorated with labels indicating grammatical relations like subject, direct object, adverbial modifier, etc.

Dependency parsers can produce such trees either by rule-based conversion from constituency trees, or by parsing directly into dependencies. The widely-used Stanford Parser is an example of the former strategy: it constituency-parses, then converts to dependencies.

Exercise 3

Use the online demo at http://nlp.stanford.edu:8080/corenlp/process to parse the sentence “I shot an elephant in my pajamas.” Look at the output displayed under “Basic Dependencies”.
Refer to http://universaldependencies.org/en/dep/ for descriptions of the edge labels.

Which way is the ambiguity resolved—i.e., which of the constituency parses from Exercise 2 does the Stanford dependency parse correspond to? How would the dependency parse differ for the other reading of the sentence?

Exercise 4

Give the productions needed for the second (‘correct’ but not funny) constituency tree from Exercise 1 (see section 6 above). On the right-hand side of each rule, underline the one nonterminal whose constituent contains the head of the rule according to the dependency parse. The first one is:

S $\rightarrow$ NP $\underline{\text{VP}}$

VP is underlined because S covers the entire sentence, and the yield of VP contains the verb, which is the overall head of the sentence. Do you see how identifying these head-containing constituents (achieved in practice with heuristics known as **head rules**) leads to a deterministic transformation from the constituency parse to the dependency parse?

The following exercises contain topics that were covered in previous years of the course: subject-verb agreement and feature unification in formal grammars. They are included only for the benefit of students who are interested in going deeper.

9 OPTIONAL: Syntactic Agreement

In this section we will address the problem of syntactic agreement. In English and other languages that have morphology words can change their form to reflect their relation with other words. For example in English the article has to agree in number with the noun: "a dog" is correct but "a dogs" is not. Similarly the verb and its subject have to agree in number and person. In languages with richer morphology such as German or Arabic the agreement constraints have to consider other aspects such as gender and case.

Let's take a look at a simple grammar:

```
g1=parse_cfg("""
    # Grammatical productions.
     S -> NP VP
     NP -> Det N | Pro
     VP -> Vi | Vt NP
    # Lexical productions.
     Pro -> "i" | "we" | "you" | "he" | "she"
     Det -> "a" | "an" | "the"
     N ->  "dog" | "banana"
     Vi -> "sleep" | "eat"
     Vt -> "eat" | "ate"
    """)
```

In this grammar we've specialized the **V** label to deal with intransitive and transitive verbs, therefore it can parse both "I sleep" and "I eat a banana". But this grammar can also parse "he sleep" and "a dog eat banana", breaking the number and person agreement. Because the lexicon doesn't have plural forms it will not parse "the dogs sleep". Call the parser app to verify this:

```
>>> app(g1, 'i sleep')
>>> app(g1, 'a dog eat banana')
>>> app(g1, 'the dogs sleep')
```

We can further enrich the label set to deal with agreement. We could modify the rule:

`S` → `NP VP`

as

`S` → `NPsg VPsg | NPpl VPpl`

Exercise 5

Create a new grammar `g1_agreement` that parses the sentence "the dogs sleep" correctly but fails to parse "a dogs sleep". To create this grammar modify the rest of the rules from grammar `g1` in a similar way as for `S` → `NP VP`, above and enrich the lexicon with the plural form of the nouns and verbs. You will for example want to add entries such as `Nsg` → `"dog" | "banana"` and `Vtsg` → `"eat" | "ate"`

Uncomment the test code for this exercise. Check if your grammar parses correctly "the dogs sleep" and fails for "a dogs sleep". How many more rules does your grammar have? The grammar should also parse correctly "a dog sleeps". What happens when parsing "a dog sleep"?

10 OPTIONAL: Feature-based grammars

We would have to add more rules to correct number and person agreement. Enriching the label set, rules and lexicon will increase the size of the grammar. To build a similar grammar for richer morphological languages where other aspects such as gender and case have to be consider could be problematic. We'll now look briefly at creating feature-based grammars that can impose agreement constrains in a more general way. For a more detailed exposition of this approach see the Building Feature Based Grammars chapter from the NLTK book.

A feature-based implementation of the *g1_agreement* grammar is provided in the file:
`/group/ltg/projects/fnlp/grammar_number_feature.fcfg`

The following rule specifies that the `NUM` (number) feature for the words "dogs" and "bananas" has the value `pl` (plural):

```
N[NUM=pl] ->  "dogs" | "bananas"
```

The following rule specifies an agreement constraint. The article and noun have to agree in number, therefore the feature value for `NUM` has to be the same. The **NP** then will have the same feature value for `NUM` as the noun or the pronoun.

```
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n] | Pro[NUM=?n]
```

Exercise 6

A function ParseWithFeatures is provided that parses a sentence with a feature-based grammar, (optionally) shows the parser trace and then the resulting parse tree for that sentence. The grammar file can be inspected with *nltk.data.show_cfg*(). You can look at the parser details by calling:

```
>>> help(nltk.load_parser)
```

Proceed in two steps as follows:

a. Uncomment the code to call `test6()`. Inspect the output of the parser when using the feature-based grammar with number constraints supplied (*grammar_number_feature.fcfg*). Try to understand how rules are combined and the feature structures unified. Inspect the grammar and try to understand how the agreement constraints and feature values are defined.

b. Enrich a local copy of the provided grammar with a `person` feature and the corresponding agreement constraints. The lexicon entry for a 1st person singular pronoun should have a feature structure like this:

   ```
   Pro[NUM=sg, PER=st] -> "i"
   ```

   You can leave a feature value underspecified like this:

   ```
   Det[NUM=pl, PER=?p] -> "the"
   ```

   Uncomment the the part b test code inside `test6`. Inspect the output of the parser. Is the parser now failing for the sentence "a dog sleep"? Are feature structures unified for the sentence "a dog sleeps"?