Mandatory assignment 2, INF2820, 2013

- **Deadline: March 7, 6:00 pm**
- **To be delivered in devilry.**
- **You should put your username in the comments of the submitted files, like `# Oblig 2, jtl`**
- **The mentioned files can be found at /projects/nlp/inf2820/cfg**

Part 1: Grammars

Exercise 1 – Basic grammar (20 points)

The starting point is the grammar you made in Exercise 4 in obligatory assignment 1. There you made a NFA. Now you are going to make a CFG for the same language fragment. We will use the NLTK grammar class for writing the grammar. You should write a grammar to a file and use `nltk.data.load()` to enter it into Python.

Because of some possible parsing problems, we will first make a grammar for a smaller fragment, the fragment without prepositional phrases (PPs). Call this grammar `basic.cfg`. You should use natural category names for the non-terminals, like `S, NP, VP, PP` as indicated by the text in obligatory assignment 1 and by the examples in the NLTK book, sec. 8.1-8.3. Extend the lexicon to contain at least 20 of each of the following categories:

- Proper nouns
- Common nouns
- Adjectives
- Intransitive verbs
- Transitive verbs

and at least 5 of each of the other subclasses of verbs.

To test your grammar, you may use the `nltk.ChartParser()` as in

```
>>> parser = nltk.ChartParser(grammar)
>>> for n in parser.nbest_parse("Mary ran".split()): print n
```

We use `nltk.Chartparser()` and not `nltk.RecursiveDescentParser()` because the former tackles all sorts of grammars, and we are not interested in the parsing procedure at this point.

Submission: `basic.cfg`

Exercise 2 – PPs (15 points)

We will now extend the grammar to also cover PPs, call the resulting grammar `pp.cfg`. It should contain at least 10 different prepositions. The grammar should reflect ambiguities in its analyses. Consider the sentences

Norwegian

a) Dyret i huset ved vannet sov.
b) Kari sov i huset ved vannet.
c) Kari likte huset ved vannet.
d) Kari likte dyret i huset ved vannet.

English

a) The animal in the house by the water slept.
b) Kari slept in the house by the water.
c) Kari loved the house by the water.
d) Kari saw the animal in the house by the water.

Each of the sentences (a-c) has two analyses. Make sure that you get that. For each of the three examples explain in one or two sentences the intuitive difference between the analyses. How many analyses does (d) get?

Also here you may use `nltk.ChartParser()` to test your grammar

Submission: The file `pp.cfg`. Program execution which shows the trees you generate for the 4 example sentences. The explanations.

Part 2 – Parsing

Exercise 3 (15 points)

You should familiarize yourself with the files `rd_recognizer.py` and `rd_parser.py`. (Cf. weekly exercise set 5)

- Run the demos that are included.
- Experiment with different levels of tracing, and
- Try to recognize/parse other sentences.
- Also experiment with changes in the grammar

You should now combine this with your own grammar `basic.cfg`. Do `basic.cfg` work together with `rd_parser.py` and do you get the same analyses as with use `nltk.Chartparser()`? If not, can you modify `basic.cfg` such that it works? Test your program on the first 4 sentences from the testset for oblig1: norsk_testset/english_testset

Submission Program execution which shows the trees you generate for the 4 testset sentences using `rd_parser.py + basic.cfg`. Possibly modified `basic.cfg`.

Exercise 4 (20 points)

In our implementation of the recursive descent parser, the files `rd_recognizer` and `rd_parser`, we implemented a purely top-down procedure. In particular, we expanded lexical rules, like

`CN → 'car',`

before we compared the word `'car'` to the input. This is unnecessary inefficient. For example, if there are 10 000 different nouns, we may have to backtrack 10 000 times before we find the right word. And even worse, we may follow a hypothesis which is wrong and the 10 000 steps are in vain. It would be much more efficient if we could process the lexical items bottom-up, e.g., look up `'car'` and see that it is a `CN` and then compared that to the top-down prediction of a `CN`.

A prerequisite for this change is that the grammar is on a special form where a rule has either of two forms

- The `rhs` contains no terminals (only zero or more non-terminals)
- The `rhs` contains exactly one terminal (and nothing more)

This is a normal way of writing a CFG for natural languages. The latter rules correspond to what we in linguistic terms call the lexicon, while the first rules are the grammar rules.

We will first check that the grammar is of the correct type. NLTK has a built-in method for checking this.

```
>>> grammar.is_nonlexical()
```

You may see the documentation by running

```
>>> help(grammar.is_nonlexical)
```

Check that this gives the correct result on the original grammar, i.e. the demo grammar in `rd_recognizer,` and then see what happens if we add any of the rules

```
NP -> 'New' 'York'
NP -> NP 'and' NP
ADJ ->
```

to the grammar one at a time.

We shall program a function `refine(grammar)` which takes a grammar as an argument and

- If the grammar is not of the correct form, prints a warning and terminates
- If the grammar is of the right form adds two attributes to the grammar.
    - The first, `grammar_rules`, should simply be a list of all productions that satisfy `is_nonlexical().`
    - The other attribute, we will call `lexicon`. It will be implemented as a Python dictionary. (In this case, the name "dictionary", which Python uses where other programming languages talks about hash tables, must be said to be appropriate ☺) The keys should be words (terminals) and the values should be lists of categories (non-terminals), e.g.

      ```
      >>> grammar.lexicon['dog']
      [N]
      ```

      The value should be a list, since a word, e.g. `'ring'` may belong to several categories.

Submission: The procedure `refine`. Print-out of the attributes of the example grammar after refinement.

Exercise 5 (20 points)

a) We may then set the refined grammar at work. We will first modify `match` from `rd_recognizer` and save the file as `rd_recognizer_refined.py`.

The match procedure considers three cases. For the first where there are no more symbols, no changes are necessary. For the last, the expansion, we may simply substitute `grammar.grammar_rules` for `grammar.productions().` For the middle case, we will have to think new. Here we shall now try to match a category (non-terminal) against a word (terminal) using `grammar.lexicon`.

There is one additional point that must be taken into consideration. That a category matches a word does not exclude the same category from being expanded into something else which matches the word. Both must be tried.

Finally, since we have used a dictionary, the program interrupts in a problematic way if we happen to feed it a string which contains a word which is not in the lexicon. To avoid this, we should when we are to test a word towards a category, check whether the word is included in the lexicon, and if not, interrupt with a meaningful message. To see whether the word is in the dictionary you may use

```
word in grammar.lexicon.keys()
```

Submission: The file `rd_recognizer_refined.py.`

b) To get an impression of how much we have gained with this refinement, we may run the original recognizer and the refined recognizer on the same input with trace, e.g. trace 1 and compare the behavior and how many numbers of lines are printed out. If you use an ungrammatical string like

"Mary saw a man with telescope"

you get an exhaustive search which best illustrates the work being done. How many lines of tracing roughly did the old recognizer yield, and how many lines did the new recognizer yield?

Submission: Answer the question.

Exercise 6 (10 points)

(A challenge) Try to refine `rd_parser.py` similarly as you did with `rd_recognizer_refined.py.` In addition to the changes you made to `rd_recognizer.py`, you now have to extend the tree when a category matches a word. This can be done on the model of how it is done when a rule is expanded in the original grammar.

Submission: `rd_parser_refined.py.` Print-out which shows that it works.