1. Context free grammar

The following are data concerning transitive sentences in a verb-final language with argument markers (or case markers) *a* and *o*. The markers are separate words. The subject and object can come in either order:

Gloss:	Jack admires Amy
(i)	Jack a Amy o admires
(ii)	Amy o Jack a admires
*	Jack a Amy a admires
*	Jack o Amy o admires
*	Amy o Jack o admires
*	Amy a Jack a admires

The verbs require two arguments:

*	Jack a respects
*	Amy o admires

The conjunction *an* is used to combine a phrase consisting of a verb and one argument with one (or more) similar phrases. In (iii), 'Jack' is a subject and 'Amy' and 'Sandy' are objects. In (iv), 'Amy' is an object, while 'Jack' and 'Sandy' are subjects.

Gloss:	Jack admires Amy and respects Sandy
(iii)	Jack a Amy o admires an Sandy o respects

Gloss:	Jack admires Amy and Sandy respects Amy
(iv)	Amy o Jack a admires an Sandy a respects

There can be any number of conjuncts:

Gloss:	Jack admires Amy and respects Sandy and respects Amy
(v)	Jack a Amy o admires an Sandy o respects an Amy o respects

Markers on the phrases combined by *an* must match, these are not possible:

*	.... Jack a admires an Sandy o respects
*	... Jack o admires an Sandy a respects

Write a context free grammar in NLTK notation for the above sentences and similar ones.

[9]:
```
import nltk
```

The productions introducing terminals are as follows, do not add to them.

[10]:
```
Gt = """
... N -> 'Jack' | 'Amy' | 'Sandy'
... V -> 'respects' | 'admires'
... A -> 'a'
... O -> 'o'
... CONJ -> 'an'
"""
```

Part A

Define a string Gn for the remaining productions. We have provided some of the left hand sides. You need two productions for each variety of VP.

[11]:
```
Gn= """
    S ->


    S ->






    VPa ->


    VPo ->
```

```
    VPa ->

    VPo ->

"""
```

[12]:
```
Gnt = Gn + Gt
G = nltk.CFG.fromstring(Gnt)
parser = nltk.ChartParser(G)
```

Part B

Draw the tree for sentence (i) that is licensed by your grammar. In our solution we do this using the parser, you should just draw the tree.

[13]:
```
# Gloss: Jack admires Amy
s1 = 'Jack a Amy o admires'.split()
gen = parser.parse(s1)
```

[14]:
```
print('\n\n\n\n\n')
next(gen)
```

[14]:

Draw the tree for sentence (iv) that is licensed by your grammar. In our solution we do this using the parser, you should just draw the tree.

[15]:
```
# Gloss Jack admires Amy and Sandy respects Amy
#
#
s4 = 'Amy o Jack a admires an Sandy a respects'.split()
gen = parser.parse(s4)
print('\n\n\n')
next(gen)
```

[15]:

[ ]:
```
```

2. Feature constraint grammar and semantics

This is a semantically interpreted feature grammar in NLTK notation.

[10]:
```
import nltk
from nltk import grammar, parse
from nltk.parse.util import load_parser
```

[11]:
```
nltk.data.show_cfg('/local/teach/cl23/T/semantics-midterm.fcfg')
```

```
% start S
# Grammar Rules
S[SEM=<?subj(?vp)>] -> DP[SEM=?subj] VP[SEM=?vp]
S[SEM=<?p & ?q>] -> S[SEM=?p] 'and' S[SEM=?q]
VP[SEM=?Q] -> 'is' NP[SEM=?Q]
DP[ SEM=<?X(?P)>] -> Det[SEM=?X] N[SEM=?P]
NP[SEM=?Q] -> 'a' N[SEM=?Q]
# Lexical Rules
Det[SEM=<\P Q.all n.(P(n) -> Q(n))>] -> 'every'
Det[SEM=<\P Q.exists n.(P(n) & Q(n))>] -> 'some'
N[SEM=<\x.cat(x)>] -> 'cat'
N[SEM=<\x.dog(x)>] -> 'dog'
N[SEM=<\x.mammal(x)>] -> 'mammal'
```

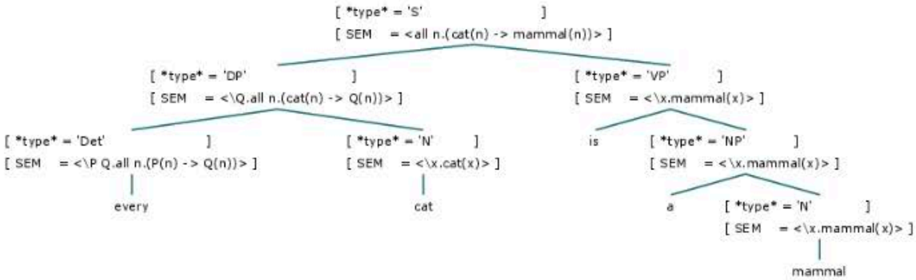
Here is an example of a feature trees licensed by the grammar.

[12]:
```
s1 = 'every cat is a mammal'.split()
s2 = 'some cat is a mammal and some dog is a mammal'.split()
```

[13]:
```
pr = load_parser('/local/teach/cl23/T/semantics-midterm.fcfg',␣
 ↪trace=0,cache=False)
```

[14]:
```
gen = pr.parse(s1)
next(gen)
```

[14]:



[15]:
```
# gen = pr.parse(s2)
# next(gen)
```

Part A

These are some additional data, involving the new terminals *happy* and *sad*. Notice that they can occur in either the DP or the VP.

[16]:
```
s3 = 'every happy cat is a cat'.split()
s4 = 'some cat is a sad cat'.split()
```

Add productions at the bottom to this grammar that will allow **s3** and **s4** to be parsed and assigned a correct semantics. You need one production for each adjective, and one to three non-lexical productions, depending on your strategy.

[17]:
```
Gs = r'''
... S[SEM=<?subj(?vp)>] -> DP[SEM=?subj] VP[SEM=?vp]
... S[SEM=<?p & ?q>] -> S[SEM=?p] 'and' S[SEM=?q]
... VP[SEM=?Q] -> 'is' NP[SEM=?Q]
... DP[ SEM=<?X(?P)>] -> Det[SEM=?X] N[SEM=?P]
... NP[SEM=?Q] -> 'a' N[SEM=?Q]
... Det[SEM=<\P Q.all n.(P(n) -> Q(n))>] -> 'every'
... Det[SEM=<\P Q.exists n.(P(n) & Q(n))>] -> 'some'
... N[SEM=<\x.cat(x)>] -> 'cat'
... N[SEM=<\x.dog(x)>] -> 'dog'
... N[SEM=<\x.mammal(x)>] -> 'mammal'
...
...
...
...
...
...
...
...
...
...
```

```
...
'''
```

Part B

Draw the feature tree for s3 that is licensed by your grammar. Include a SEM and *type* feature on each vertex. Make sure that the semantics for S is plausible, and that the tree shape and semantics on each node correspond to the grammar. The VP is provided. (In our solution the tree is generated with a parser, loading the grammar from a file. You should just draw the tree.)

[21]:
```
print(" ".join(s3))
print("\n\n\n\n\n")
pr2 = load_parser('/local/teach/cl23/T/semantics-midterm-incremented.fcfg',␣
 ↪trace=0,cache=False)
gen = pr2.parse(s3)
next(gen)
```

```
every happy cat is a cat
```