Code Preparations

In [ ]:
```
import nltk
from nltk import grammar, parse
from nltk.parse.generate import generate
from platform import python_version
python_version()
```

Out[ ]:
```
'3.7.16'
```

In [ ]:
```
print(nltk.__version__)
```

```
3.5
```

to use Angela Liu's implementation from mapping the word

In [ ]:
```
from typing import Callable, List, Set

def to_model_str(word: str, special_rels: List[Callable[[str], str]]=[]) ->
    """
    Creates the string form of the model for the input word. This string is
    By default, the function will only add the relations mapping i => i for
    mapping char => the set of tuples (i, word[i]). The `special_rels` funct
    be added to the valuation string.

    :param word: The word to create a model string for.
    :param special_rels: A list of functions that when called return a strin
    :returns: a string representing the model for word
    """
    n = len(word)
    model_str = []
    char = []
    for i in range(1, n+1):
        model_str.append(f'{i} => {i}')
        char.append((i, word[i-1]))
    model_str.append(f'char => {set(char)}'.lower())
    return '\n'.join(model_str + [rel(word) for rel in special_rels]).replac
# Angela Liu
import re


get_vowel = lambda w: f'vowel => {set(re.findall(r"[AEIOUaeiou]", w))}'.lowe
get_cons = lambda w: 'cons => {}'.format(set(re.findall(r"[^AEIOUaeiou\W0-9]
follows = lambda w: f'le => {set([(i+1,j+1) for i in range(len(w)) for j in
get_capital = lambda w: f'capital => {set([m.span()[0] + 1 for m in re.findi
# Angela Liu

get_glide = lambda w: f'glide => {set(re.findall(r"[YWyw]", w))}'.lower()
# is_final = lambda w: f'final => {set((len(w),))}'
get_adj = lambda w: f'adj => {set([(i+1,j+1) for i in range(len(w)) for j in
# added

def emptysets(val:nltk.sem.evaluate.Valuation):
  val.update([(k,set()) for (k,v) in val.items() if v == 'set()'])

words = ['cat', 'mAtch', 'peRiLOuSy']
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, get_cons, foll
for v in vals: emptysets(v)
models = [nltk.Model(val.domain, val) for val in vals]
for w, m in zip(words, models):
    print(f'{w}\n-----------------\n{m}\n')
# Angela Liu
```

```
cat
-----------------
Domain = {'1', '2', 't', 'a', '3', 'c'},
Valuation =
{'1': '1',
 '2': '2',
```

```
 '3': '3',
 'adj': {('2', '1'), ('1', '2'), ('3', '2'), ('2', '3')},
 'capital': set(),
 'char': {('2', 'a'), ('1', 'c'), ('3', 't')},
 'cons': {('t',), ('c',)},
 'glide': set(),
 'le': {('1', '2'), ('2', '3'), ('1', '3')},
 'vowel': {('a',)}}

mAtch
----------------
Domain = {'m', '1', 'a', '2', 't', '3', '4', 'h', '5', 'c'},
Valuation =
{'1': '1',
 '2': '2',
 '3': '3',
 '4': '4',
 '5': '5',
 'adj': {('1', '2'),
         ('2', '1'),
         ('2', '3'),
         ('3', '2'),
         ('3', '4'),
         ('4', '3'),
         ('4', '5'),
         ('5', '4')},
 'capital': {('2',)},
 'char': {('5', 'h'), ('2', 'a'), ('3', 't'), ('1', 'm'), ('4', 'c')},
 'cons': {('t',), ('c',), ('m',), ('h',)},
 'glide': set(),
 'le': {('1', '2'),
        ('1', '3'),
        ('1', '4'),
        ('1', '5'),
        ('2', '3'),
        ('2', '4'),
        ('2', '5'),
        ('3', '4'),
        ('3', '5'),
        ('4', '5')},
 'vowel': {('a',)}}

peRiLOuSy
----------------
Domain = {'9', '1', '8', 'r', 'u', 's', 'y', 'e', '2', 'l', 'o', 'i', '7', '
p', '3', '4', '6', '5'},
Valuation =
{'1': '1',
 '2': '2',
 '3': '3',
 '4': '4',
 '5': '5',
 '6': '6',
```

```
         '7': '7',
         '8': '8',
         '9': '9',
         'adj': {('1', '2'),
                 ('2', '1'),
                 ('2', '3'),
                 ('3', '2'),
                 ('3', '4'),
                 ('4', '3'),
                 ('4', '5'),
                 ('5', '4'),
                 ('5', '6'),
                 ('6', '5'),
                 ('6', '7'),
                 ('7', '6'),
                 ('7', '8'),
                 ('8', '7'),
                 ('8', '9'),
                 ('9', '8')},
         'capital': {('8',), ('5',), ('3',), ('6',)},
         'char': {('1', 'p'),
                  ('2', 'e'),
                  ('3', 'r'),
                  ('4', 'i'),
                  ('5', 'l'),
                  ('6', 'o'),
                  ('7', 'u'),
                  ('8', 's'),
                  ('9', 'y')},
         'cons': {('l',), ('p',), ('s',), ('r',), ('y',)},
         'glide': {('y',)},
         'le': {('1', '2'),
                ('1', '3'),
                ('1', '4'),
                ('1', '5'),
                ('1', '6'),
                ('1', '7'),
                ('1', '8'),
                ('1', '9'),
                ('2', '3'),
                ('2', '4'),
                ('2', '5'),
                ('2', '6'),
                ('2', '7'),
                ('2', '8'),
                ('2', '9'),
                ('3', '4'),
                ('3', '5'),
                ('3', '6'),
                ('3', '7'),
                ('3', '8'),
                ('3', '9'),
                ('4', '5'),
```

```
        ('4', '6'),
        ('4', '7'),
        ('4', '8'),
        ('4', '9'),
        ('5', '6'),
        ('5', '7'),
        ('5', '8'),
        ('5', '9'),
        ('6', '7'),
        ('6', '8'),
        ('6', '9'),
        ('7', '8'),
        ('7', '9'),
        ('8', '9')},
 'vowel': {('o',), ('e',), ('u',), ('i',)}}
```

Start of the work:

```
Semantics of sentences about strings
Computational Linguistics Spring 2023
Problems Set 2
```

The text for this module is the NLTK book Chapter 9. Building Feature Based Grammars and Chapter 10. Analyzing the Meaning of Sentences

See also lecture8_2023.ipynb and string_2023.ipynb

The purpose of the assingnment is to develop feature-based grammars that include logical semantics, and to evaluate the adequacy of the semantics by computing truth in logically constructed models. For instance, we want to be able to evaluate whether the sentence

every consonant is capitalized

is true or false as description of the word

CINEMA

or

Cinema

or

CINEmA.

In each problem n do these steps. The problem statement gives sentence sn. See Chapters 9 and 10 for the methodology.

(i) Define a feature based grammar gn that includes all the words in sentence sn its lexicon. The feature grammars will usually add a word and/or construction to a base grammar which will be similar to simple-sem.fcfg. This base grammar will be distributed. (It will be helpful to figure out how to add a lexical item or production to a grammar in Python. Discuss the method for this on the forum. Or you can define the grammar from scratch.)

(ii) Parse the sentenced and display the tree.

(iii) Map sn to a logical formula fn by parsing with gn and extracting the semantics that annotates the root.

(iv) Define a combination four words (serving as models) and the intuitive truth values of sentence sn as a description of the word.

(v) Transform the four words into four models or valuations in the sense of Chapter 10. This can be done as in Lecture 8, or by using a function. Code for this may be shared and discussed on the forum.

(vi) Evaluate formula fn in the four models to obtain four truth values. Compare them to the target truth values.

Work individually, except that code for mapping a word to a valuation may be shared. Post techinical questions and requests for hints on the forum.

Notes The problems are selected so that quantifiers are used only in subject position. So it is not necessary (except perhaps in challenge problems) to apply the strategy from simple-sem.fcfg to fit quantified NPs into object positions.

The words 'precedes' and 'follows' are interpreted in the sense of 'not necessarily immediately'.

Problems

1. letter three is final

   Define "final" in a way that works for words of any length. Don't include a corresponding constant in the valutions. Decide what should happen with a words of length one.

I added a grammar rule

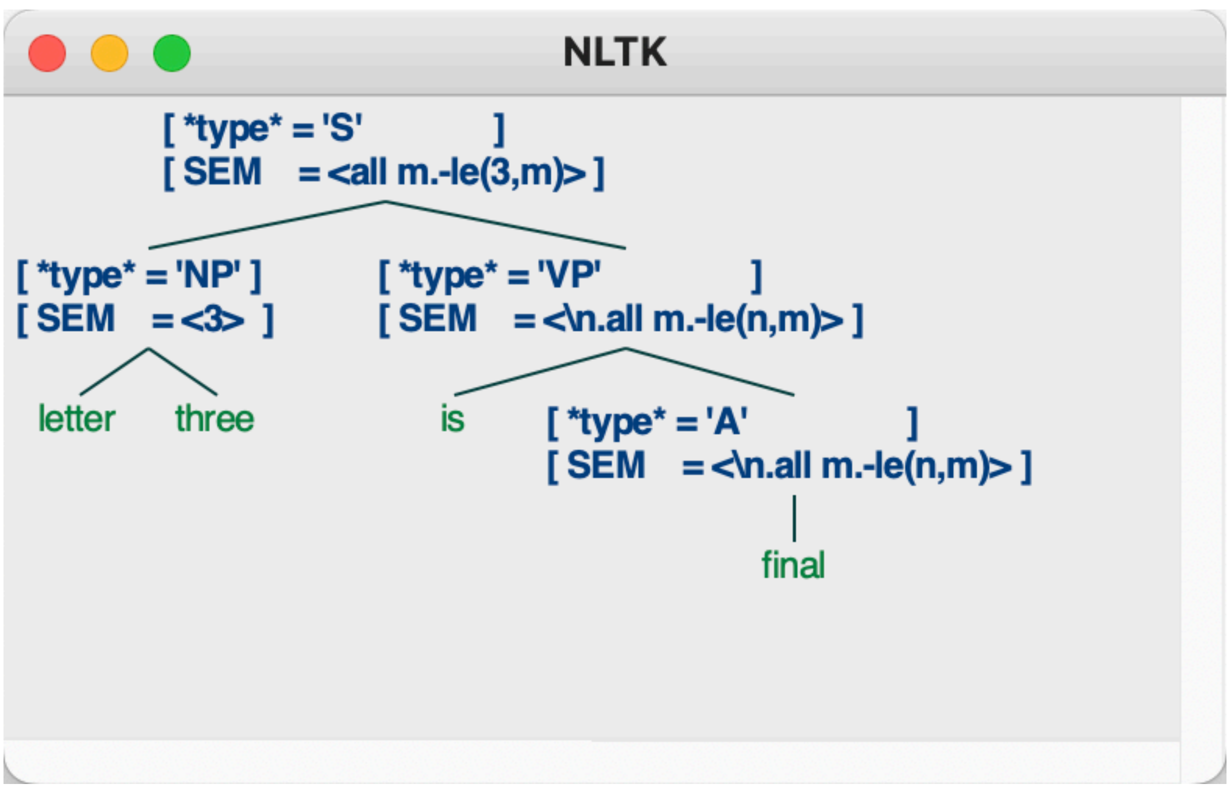
```
A[SEM=<\n. all m.-le(n,m)  >] -> 'final'
```

parse and display

In [ ]:
```
g6 = nltk.load_parser('h2.fcfg', trace=0, cache=False)
s6 = 'letter three is final'
s6_split = s6.split()
for tree in g6.parse(s6_split): print(tree)
```

```
(S[SEM=<all m.-le(3,m)>]
  (NP[SEM=<3>] letter three)
  (VP[SEM=<\n.all m.-le(n,m)>] is (A[SEM=<\n.all m.-le(n,m)>] final)))
```

In [ ]:
```
# tree.draw()
```



now the logical formula

In [ ]:
```
t6=next(g6.parse(s6_split))
f6 = t6.label()['SEM']
print(f6)
```

```
all m.-le(3,m)
```

define a sample four word

In [ ]:
```
e6 = [('emu',True),('batd',False),('cbjasaa',False),('awe',True)]
```

In [ ]:
```
words = [e[0] for e in e6]
truths = [e[1] for e in e6]
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, follows])) for
assignments = [nltk.Assignment(val.domain) for val in vals]
for val in vals: emptysets(val)
models = [nltk.Model(val.domain, val) for val in vals]
```

In [ ]:
```
print(f'{s6}\n---------------')
for w, a, m in zip(words, assignments, models):
    print(f'{w}\n{m.evaluate(str(f6),a)}\n----------------')
```

```
letter three is final
---------------
emu
True
----------------
batd
False
----------------
cbjasaa
False
----------------
awe
True
----------------
```

```
+ The answer is correct! emu & awe has the third letter being the
final
```

1. every vowel is adjacent to letter three

   Include 'adjacent' in the grammar. Use the strategy with PP[to] to select the preposition. Either define the semantics of 'adjacent' in terms of the available primitives, or add to the function that constructs valuations.

I added a grammar rule

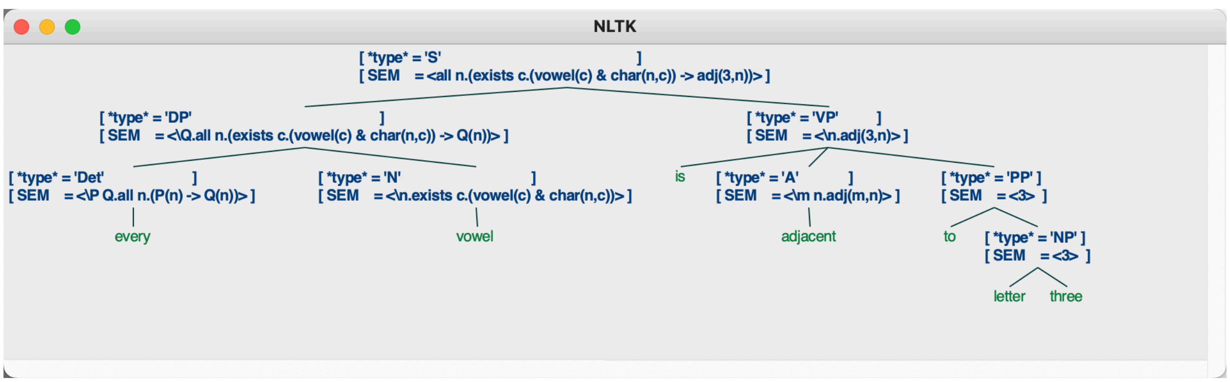
```
VP[SEM=<?P(?Q)>] -> 'is' A[SEM=?P] PP[SEM=?Q]

PP[SEM=?Q] -> 'to' NP[SEM=?Q]
```

parse and display

In [ ]:
```
g7 = nltk.load_parser('h2.fcfg', trace=0, cache=False)
s7 = 'every vowel is adjacent to letter three'
s7_split = s7.split()
for tree in g7.parse(s7_split): print(tree)
```

```
(S[SEM=<all n.(exists c.(vowel(c) & char(n,c)) -> adj(3,n))>]
  (DP[SEM=<\Q.all n.(exists c.(vowel(c) & char(n,c)) -> Q(n))>]
    (Det[SEM=<\P Q.all n.(P(n) -> Q(n))>] every)
    (N[SEM=<\n.exists c.(vowel(c) & char(n,c))>] vowel))
  (VP[SEM=<\n.adj(3,n)>]
    is
    (A[SEM=<\m n.adj(m,n)>] adjacent)
    (PP[SEM=<3>] to (NP[SEM=<3>] letter three))))
```

In [ ]:
```
# tree.draw()
```



now the logical formula

In [ ]:
```
t7=next(g7.parse(s7_split))
f7 = t7.label()['SEM']
print(f7)
```

```
all n.(exists c.(vowel(c) & char(n,c)) -> adj(3,n))
```

define a sample four word

In [ ]:
```
e7 = [('emuqe',False),('batd',True),('cbjas',True),('aweqwef',False)]
```

In [ ]:
```
words = [e[0] for e in e7]
truths = [e[1] for e in e7]
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, follows,get_ad
assignments = [nltk.Assignment(val.domain) for val in vals]
for val in vals: emptysets(val)
models = [nltk.Model(val.domain, val) for val in vals]
```

In [ ]:
```
print(f'{s7}\n----------------')
for w, a, m in zip(words, assignments, models):
    print(f'{w}\n{m.evaluate(str(f7),a)}\n-----------------')
```

```
every vowel is adjacent to letter three
---------------
emuqe
False
----------------
batd
True
----------------
cbjas
True
----------------
aweqwef
False
----------------
```

```
 + The answer is correct!
```

1. every vowel that follows letter two is capitalized
   This has a subject relative clause . Methodology for the semantics of subject relative clauses is in lecture8.ipynb.

I added a grammar rule

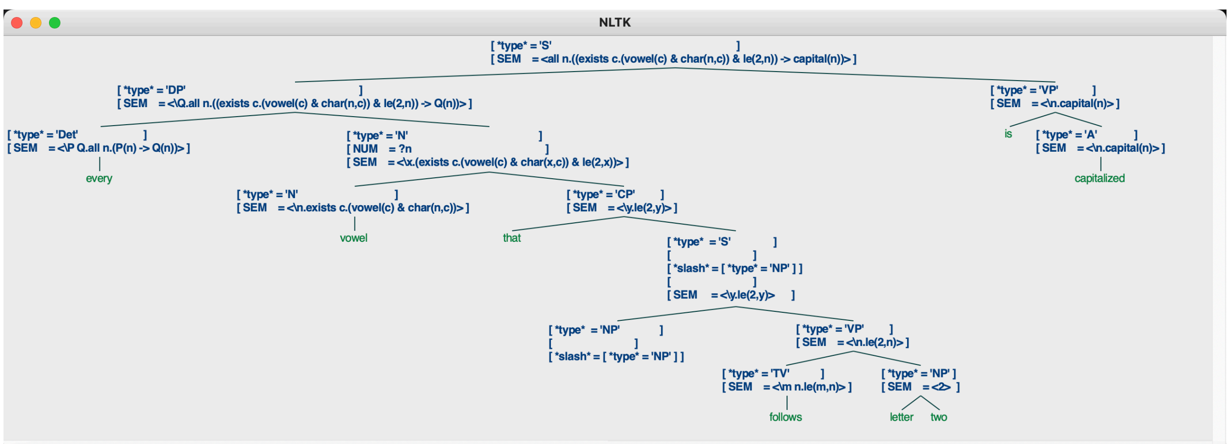
```
CP[SEM=?P] -> 'that' S[SEM=?P]/NP
N[NUM=?n,SEM=<\x.(?P(x) & ?Q(x))>] -> N[NUM=?n,SEM=?P]
CP[SEM=?Q]
NP/NP ->
VP[NUM=?n,SEM=<\y x.(?v(y)(x))>]/NP -> TV[NUM=?n,SEM=?v]
NP/NP[SEM=?obj]
S[SEM = <\y.(?vp(y)(?subj))>]/NP -> NP[NUM=?n,SEM=?subj]
VP[NUM=?n,SEM=?vp]/NP
S[SEM = <\y.(?vp(y))>]/NP -> NP/NP VP[NUM=?n,SEM=?vp]
```

parse and display

In [ ]:
```python
g8 = nltk.load_parser('h2.fcfg', trace=0
                      , cache=False)
s8 = 'every vowel that follows letter two is capitalized'
s8_split = s8.split()
for tree in g8.parse(s8_split): print(tree)
```

```
(S[SEM=<all n.((exists c.(vowel(c) & char(n,c)) & le(2,n)) -> capital(n))>]
  (DP[SEM=<\Q.all n.((exists c.(vowel(c) & char(n,c)) & le(2,n)) -> Q(n))>]
    (Det[SEM=<\P Q.all n.(P(n) -> Q(n))>] every)
    (N[NUM=?n, SEM=<\x.(exists c.(vowel(c) & char(x,c)) & le(2,x))>]
      (N[SEM=<\n.exists c.(vowel(c) & char(n,c))>] vowel)
      (CP[SEM=<\y.le(2,y)>]
        that
        (S[SEM=<\y.le(2,y)>]/NP[]
          (NP[]/NP[] )
          (VP[SEM=<\n.le(2,n)>]
            (TV[SEM=<\m n.le(m,n)>] follows)
            (NP[SEM=<2>] letter two))))))
  (VP[SEM=<\n.capital(n)>] is (A[SEM=<\n.capital(n)>] capitalized)))
```

In [ ]:
```python
# tree.draw()
```



now the logical formula

In [ ]:
```python
t8=next(g8.parse(s8_split))
f8 = t8.label()['SEM']
print(f8)
```

```
all n.((exists c.(vowel(c) & char(n,c)) & le(2,n)) -> capital(n))
```

define a sample four word

In [ ]:
```python
e8 = [('emUqE',True),('baTde',False),('cbJAs',True),('aweqwef',False)]
```

In [ ]:
```
words = [e[0] for e in e8]
truths = [e[1] for e in e8]
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, follows,get_ad
assignments = [nltk.Assignment(val.domain) for val in vals]
for val in vals: emptysets(val)
models = [nltk.Model(val.domain, val) for val in vals]
```

In [ ]:
```
print(f'{s8}\n----------------')
for w, a, m in zip(words, assignments, models):
    print(f'{w}\n{m.evaluate(str(f8),a)}\n----------------')
```

```
every vowel that follows letter two is capitalized
---------------
emUqE
True
----------------
baTde
False
----------------
cbJAs
True
----------------
aweqwef
False
----------------
```

```
+ The answer is correct!
```

1. some vowel immediately precedes letter three
   some vowel immediately follows letter three

   Define "immediately" in a way that works for both "precedes" and "follows".

I added a grammar rule

```
TV[SEM=<\m n.(le(m,n) & adj(m,n))>] -> 'immediately'
'follows'
TV[SEM=<\m n.(le(n,m) & adj(n,m))>] -> 'immediately'
'precedes'
```

parse and display

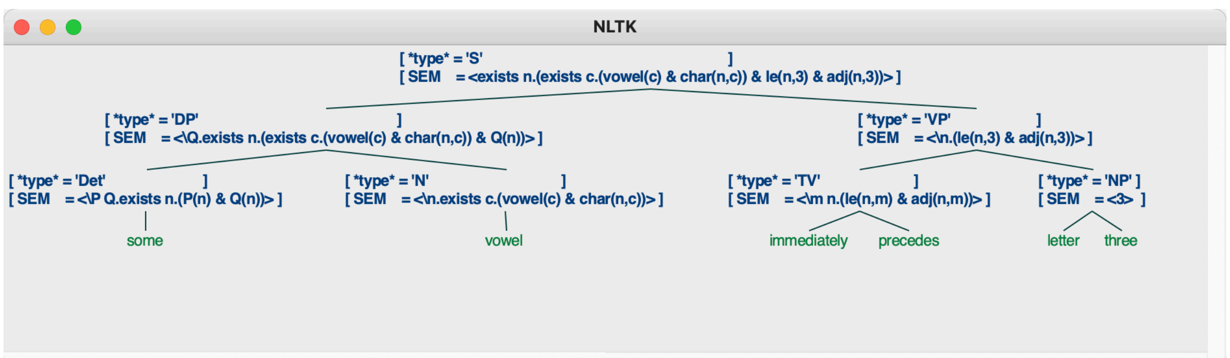
In [ ]:
```python
g9 = nltk.load_parser('h2.fcfg', trace=0
                      , cache=False)
s9 = 'some vowel immediately precedes letter three'
s9_split = s9.split()
for tree in g9.parse(s9_split): print(tree)
```

```
(S[SEM=<exists n.(exists c.(vowel(c) & char(n,c)) & le(n,3) & adj(n,3))>]
  (DP[SEM=<\Q.exists n.(exists c.(vowel(c) & char(n,c)) & Q(n))>]
    (Det[SEM=<\P Q.exists n.(P(n) & Q(n))>] some)
    (N[SEM=<\n.exists c.(vowel(c) & char(n,c))>] vowel))
  (VP[SEM=<\n.(le(n,3) & adj(n,3))>]
    (TV[SEM=<\m n.(le(n,m) & adj(n,m))>] immediately precedes)
    (NP[SEM=<3>] letter three)))
```

In [ ]:
```python
g91 = nltk.load_parser('h2.fcfg', trace=0
                      , cache=False)
s91 = 'some vowel immediately follows letter three'
s91_split = s91.split()
for tree in g91.parse(s91_split): print(tree)
```

```
(S[SEM=<exists n.(exists c.(vowel(c) & char(n,c)) & le(3,n) & adj(3,n))>]
  (DP[SEM=<\Q.exists n.(exists c.(vowel(c) & char(n,c)) & Q(n))>]
    (Det[SEM=<\P Q.exists n.(P(n) & Q(n))>] some)
    (N[SEM=<\n.exists c.(vowel(c) & char(n,c))>] vowel))
  (VP[SEM=<\n.(le(3,n) & adj(3,n))>]
    (TV[SEM=<\m n.(le(m,n) & adj(m,n))>] immediately follows)
    (NP[SEM=<3>] letter three)))
```

In [ ]:
```python
tree.draw()
```



now the logical formula

In [ ]:
```python
t9=next(g9.parse(s9_split))
f9 = t9.label()['SEM']
print(f9)
```

```
exists n.(exists c.(vowel(c) & char(n,c)) & le(n,3) & adj(n,3))
```

define a sample four word

In [ ]:
```
e9 = [('emUqE',False),('baTde',True),('cbJAs',False),('aeqwef',True)]
```

In [ ]:
```
words = [e[0] for e in e9]
truths = [e[1] for e in e9]
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, follows,get_ad
assignments = [nltk.Assignment(val.domain) for val in vals]
for val in vals: emptysets(val)
models = [nltk.Model(val.domain, val) for val in vals]
```

In [ ]:
```
print(f'{s9}\n---------------')
for w, a, m in zip(words, assignments, models):
    print(f'{w}\n{m.evaluate(str(f9),a)}\n----------------')
```

```
some vowel immediately precedes letter three
---------------
emUqE
False
----------------
baTde
True
----------------
cbJAs
False
----------------
aeqwef
True
----------------
```

+ The answer is correct!

1. Post at least one challenge problem to PS2 challenge on the forum before the target date for challenge problems. Be creative rather than varying challenge problems from others in mechanical ways. Include your challenge problem here.

+Define "symmetric" as in the sentence below

+every letter is symmetric.

+the word symmetric means that the letter on some position is the same as the letter holding the same position only counted backwards of the word.
+Put together, this sentence is referring to the situation where the word can be read backward and still lookes the same, AKA a palindrome.
for example, abba, steponnopets, nisioisin are satisfied words.

1. Solve at least one challenge problem. When you solve it, post core part of the result to ed. Give your solution here in the format above. Don't pick a problem that somebody else has solved.

Define "repeated" as used in the sentence below.

```
Some consonant is repeated.
```

Intended meaning: some consonant type (e.g. "b/B") occurs at least twice.

I added a grammar rule

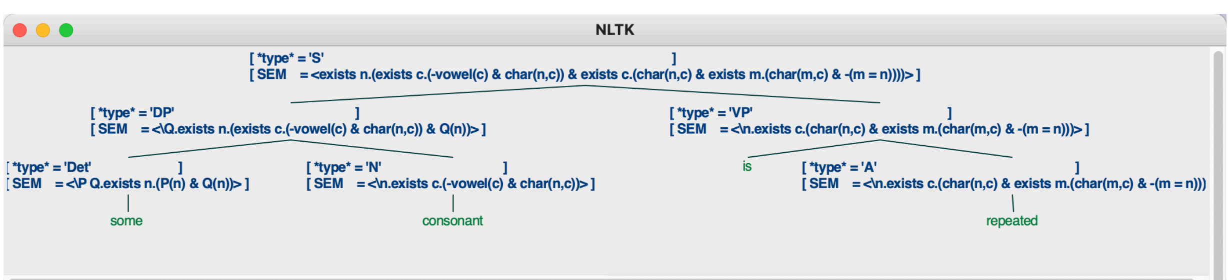
```
A[SEM=<\n. exists c.(char(n,c) & (exists m.(char(m,c) &
m!=n)) )  >] -> 'repeated'
```

parse and display

In [ ]:
```python
g11 = nltk.load_parser('h2.fcfg', trace=0
                       , cache=False)
s11 = 'some consonant is repeated'
s11_split = s11.split()
for tree in g11.parse(s11_split): print(tree)
```

```
(S[SEM=<exists n.(exists c.(-vowel(c) & char(n,c)) & exists c.(char(n,c) & e
xists m.(char(m,c) & -(m = n)))))>]
  (DP[SEM=<\Q.exists n.(exists c.(-vowel(c) & char(n,c)) & Q(n))>]
    (Det[SEM=<\P Q.exists n.(P(n) & Q(n))>] some)
    (N[SEM=<\n.exists c.(-vowel(c) & char(n,c))>] consonant))
  (VP[SEM=<\n.exists c.(char(n,c) & exists m.(char(m,c) & -(m = n)))>]
    is
    (A[SEM=<\n.exists c.(char(n,c) & exists m.(char(m,c) & -(m = n)))>]
      repeated)))
```

In [ ]:
```python
tree.draw()
```



now the logical formula

In [ ]:
```
t11=next(g11.parse(s11_split))
f11 = t11.label()['SEM']
print(f11)
```

```
exists n.(exists c.(-vowel(c) & char(n,c)) & exists c.(char(n,c) & exists m.
(char(m,c) & -(m = n))))
```

define a sample four word

In [ ]:
```
e11 = [('emUqEm',True),('baTde',True),('cbJAs',False),('aeqwWef',True)]
```

In [ ]:
```
words = [e[0] for e in e11]
truths = [e[1] for e in e11]
vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, follows,get_ad
assignments = [nltk.Assignment(val.domain) for val in vals]
for val in vals: emptysets(val)
models = [nltk.Model(val.domain, val) for val in vals]
```

In [ ]:
```
print(f'{s11}\n---------------')
for w, a, m in zip(words, assignments, models):
    print(f'{w}\n{m.evaluate(str(f11),a)}\n----------------')
```

```
some consonant is repeated
---------------
emUqEm
True
----------------
baTde
False
----------------
cbJAs
False
----------------
aeqwWef
True
----------------
```

below is the function that maps word strings to valuations that i modified. This is still mostly based on Angela Liu's work

In [ ]:
```
# get_vowel = lambda w: f'vowel => {set(re.findall(r"[AEIOUaeiou]", w))}'.lc
# get_cons = lambda w: 'cons => {}'.format(set(re.findall(r"[^AEIOUaeiou\W0-
# follows = lambda w: f'le => {set([(i+1,j+1) for i in range(len(w)) for j i
# get_capital = lambda w: f'capital => {set([m.span()[0] + 1 for m in re.fin
# get_glide = lambda w: f'glide => {set(re.findall(r"[YWyw]", w))}'.lower()
# # is_final = lambda w: f'final => {set((len(w),))}'
# get_adj = lambda w: f'adj => {set([(i+1,j+1) for i in range(len(w)) for j
# # added

# def emptysets(val:nltk.sem.evaluate.Valuation):
#   val.update([(k,set()) for (k,v) in val.items() if v == 'set()'])

# words = ['cat', 'mAtch', 'peRiLOuSy']
# vals = [nltk.Valuation.fromstring(to_model_str(w, [get_vowel, get_cons, fc
# for v in vals: emptysets(v)
# models = [nltk.Model(val.domain, val) for val in vals]
# for w, m in zip(words, models):
#     print(f'{w}\n----------------\n{m}\n')
```