01 NLTK (Lab)

This question use the Python Natural Language Toolkit. If needed, install by `pip3 install nltk` or `pip install nltk`. You may need to use `python3 -m pip install nltk` if you have multiple versions of Python. Following example is from the NLTK Book. It shows the ambiguity of the sentence:

`I shot an elephant in my pajamas`

This is from the Groucho Marx movie, *Animal Crackers* (1930): "While hunting in Africa, I shot an elephant in my pajamas. How he got into my pajamas, I don't know." First, a grammar is defined that is sufficient to show the ambiguity and a parser for that grammar is created:

```
import nltk
groucho_grammar = nltk.CFG.fromstring("""
S -> NP VP
PP -> P NP
NP -> Det N | Det N PP | 'I'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")
parser = nltk.ChartParser(groucho_grammar)
```

Now all parse trees for this sentence are created and printed:

```
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']))
for t in trees: print(t)
```

The output shows that there are two parse trees, printed with indentation. They can also be graphically visualized:

```
# trees[0] # draws graphically inline; works only locally, not on JupyterHub
# trees[0].draw() # draws graphically in separate windows, works only locally, not on Jupyte
trees[0].pretty_print() # draws textually, can sometimes be confusing, needs monospaced font
# trees[0].pprint() # prints textually, same as print(...)
```

```
trees[1].pretty_print()
```

Part 1

Let `G = (T, N, P, S)` where `T = {a, b}`, `N = {S}`, and productions `P` are:

```
S →
S → aSbS
S → bSaS
```

Draw all parse trees for the sentence `abab` with NLTK!

```
import nltk
groucho_grammar = nltk.CFG.fromstring("""
S -> A S B S | B S A S |
A -> 'a'
B -> 'b'
""")
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['a', 'b', 'a', 'b']))
for t in trees: print(t)
trees[0].pretty_print()

trees[1].pretty_print()
```

Part 2

Draw the parse tree of `id × (id + id)` in grammar `G` using NLTK!

```
import nltk
groucho_grammar = nltk.CFG.fromstring("""
E -> T | E N T
T -> F | T N F
F -> N | N E N
N -> 'id' | '+' | 'x' | '(' | ')'
""")
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['id', 'x', '(', 'id', '+', 'id', ')']))
# for t in trees: print(t)
trees[0].pretty_print()
```

02 Precedence and Associativity with NLTK

Consider expression made up of identifiers `a`, `b`, `c`, `d` and operators `+`, `-`, like

```
a - b + c - d
```

Write grammars as below with NLTK and draw the parse trees with NLTK.

1. Write a grammar such that `+` binds tighter than `-`, i.e. the above sentence would be evaluated as `(a - (b + c)) - d`. Draw the parse tree for `a - b + c - d`!

```
import nltk
groucho_grammar = nltk.CFG.fromstring("""
E -> E N T | T
T -> T P F | F
F -> T | N | 'a' | 'b' | 'c' | 'd'
N -> '-'
```

```
P -> '+'
""")
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['a', '-', 'b', '+', 'c', '-', 'd']))
trees[0].pretty_print()
```

2. Write a grammar such that `-` binds tighter than `+`, i.e. the above sentence would be evaluated as `(a - b) + (c - d)`. Draw the parse tree for `a - b + c - d`!

```
groucho_grammar = nltk.CFG.fromstring("""
E -> E N T | T
T -> T P F | F
F -> T | N  | 'a' | 'b' | 'c' | 'd'
N -> '+'
P -> '-'
""")
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['a', '-', 'b', '+', 'c', '-', 'd']))
# for t in trees: print(t)
trees[0].pretty_print()
```

3. Write a grammar such that `+` and `-` bind equally strong but associate to the left, i.e. the above sentence would be evaluated as `(( a - b) + c) - d`. Draw the parse tree for `a - b + c - d`!

```
groucho_grammar = nltk.CFG.fromstring("""
E -> E P T | T
T -> T N F | F
F -> E | N | 'a' | 'b' | 'c' | 'd'
N -> '+'
P -> '-'
""")
parser = nltk.ChartParser(groucho_grammar)
trees = list(parser.parse(['a', '-', 'b', '+', 'c', '-', 'd']))
# for t in trees: print(t)
trees[0].pretty_print()
```

4. Write a grammar such that `+` and `-` bind equally strong but associate to the right, i.e. the above sentence would be evaluated as `a - (b + (c - d))`. Draw the parse tree for `a - b + c - d`!

```
groucho_grammar = nltk.CFG.fromstring("""
E -> T | E N T
T -> F | T N F
F -> N | N E N
N -> 'a' | 'b' | 'c' | 'd' | '+' | '-'
""")
parser = nltk.ChartParser(groucho_grammar)
```

```
trees = list(parser.parse(['a', '-', 'b', '+', 'c', '-', 'd']))
# for t in trees: print(t)
trees[0].pretty_print()
```

04 Grammar for a b c d

Procedure `derivable` from the course notes can be used for unrestricted grammars, not just for context-sensitive grammar. The procedure will terminate with `true` or `false` for context-sensitive grammars (*decision procedure*) but may or may not terminate for unrestricted grammars (*semi-decision procedure*).

```
def derivable(S, P,  ):
    # S: start symbol, a string, P: productions, a set of pairs of strings,  : string
    d0, d = {}, {S} # set of strings
    while d != d0:
        d0 = d #; print(d)
        for ( ,  ) in P:
            for   in d0:
                i =  .find( , 0) #print(' , i',  , i)
                while i != - 1:
                      =  [0:i] +   +  [i + len( ):] #print(' ',  )
                    if   ==  : return True
                    elif len( ) <= len( ): d = d.union({ })
                    i =  .find( , i + 1) #print('d, i', d, i)
    return False
```

Use procedure `derivable` to show that `abc`, `aabbcc`, `aaabbbccc` are derivable in `G` but `aabc` and `abbc` are not! Define `S` and `P` in the cell below and run the next cell for testing.

```
S = ('S')
P = {('S', 'abc'), ('S', 'aBSc'), ('Ba', 'aB'), ('Bb', 'bb')}
```

```
assert derivable(S, P , 'abc')
assert derivable(S, P , 'aabbcc')
assert derivable(S, P , 'aaabbbccc')
assert not derivable(S, P , 'aabc')
assert not derivable(S, P , 'abbc')
```

Use procedure `derivable` to show that `abc`, `aabbcc`, `aaabbbccc` are derivable in `G '`, the context-sensitive version of `G` , but `aabc` and `abbc` are not!

```
S = ('S')
P = {('S', 'Abc'), ('S', 'ABSc'), ('BA', 'BX'), ('BX', 'AX'), ('AX', 'AB'), ('Bb', 'bb'), ('
```

```
assert derivable(S, P , 'abc')
assert derivable(S, P , 'aabbcc')
assert derivable(S, P , 'aaabbbccc')
```

```
assert not derivable(S, P , 'aabc')
assert not derivable(S, P , 'abbc')
```

Now consider the language {a b c d | i, n 1}. Write a grammar for this language by defining S and G below and use procedure derivable to check that abcd, aabccd, aabbbccddd are derivable in but aabbcd, abccdd, acbd are not! The grammar does not have to be context-sensitive, but procedure derivable has to terminate.

```
S = ('S')
P = {('S', 'abcd'), ('S', 'aXbcd'),('S', 'abcYd'),('S', 'aXbcYd'), ('X', 'aXc'), ('X', 'ac')
```

```
assert derivable(S, P , 'abcd')
assert derivable(S, P , 'aabccd')
assert derivable(S, P , 'aabbbccddd')
assert not derivable(S, P , 'aabbcd')
assert not derivable(S, P , 'abccdd')
assert not derivable(S, P , 'acbd')
```