Handout 17: Grammars

1. Rules in a grammar

a. Load grammar

```
>>> from nltk import CFG
>>> g1 = CFG.fromstring(open(’g1.cfg’).read())
```

b. Start symbol. Note: it is not a string.

```
>>> g1.start()
S
>>> type(_)
<class ’nltk.grammar.Nonterminal’>
>>> g1.start() == ’S’
False
```

c. Rules

```
>>> rules = g1.productions()
>>> rules[0]
S -> NP VP
>>> len(rules)
26
```

d. Rule lhs and rhs

```
>>> r = rules[0]
>>> r
S -> NP VP
>>> r.lhs()
S
>>> r.rhs()
(NP, VP)
```

2. Nonterminal elements are `Nonterminals`, not strings

a. Nonterminals

```
>>> r.lhs()
S
>>> from nltk import Nonterminal as NT
>>> r.lhs() == NT(’S’)
True
```

b. Terminals

```
>>> rules[7]
V -> ’saw’
>>> v = rules[7].rhs()[0]
>>> isinstance(v, NT)
False
>>> isinstance(v, str)
True
```

3. Accessing rules: By lhs or *first* of rhs.

```
>>> g1.productions(lhs=NT('VP'))
[VP -> V NP, VP -> V NP PP]
>>> g1.productions(rhs=NT('NP'))
[S -> NP VP]
```

4. How can we print out just the lexical rules?

5. Creating rules

```
>>> from nltk import Production
>>> r1 = Production(NT('S'), [NT('NP'), NT('VP')])
>>> r1
S -> NP VP
>>> r1 == rules[0]
True
```

6. Creating a grammar

```
>>> r2 = Production(NT('NP'), ['Fido'])
>>> r3 = Production(NT('VP'), ['barks'])
>>> g = CFG(NT('S'), [r1, r2, r3])
>>> print(g)
Grammar with 3 productions (start state = S)
    S -> NP VP
    NP -> 'Fido'
    VP -> 'barks'
```

7. Define a function `save_grammar` that takes a grammar and filename and saves out the grammar in a form that can be read back in.

Random generation

8. To see if a grammar is over-generating

9. How do we generate a tree?

- **a.** Root of tree = start symbol

```
>>> x1 = g1.start()
```

- **b.** What are the possible expansions?

```
>>> options1 = g1.productions(lhs=x1)
>>> options1
[S -> NP VP]
```

- **c.** Choose one at random

```
>>> import random
>>> r1 = random.choice(options1)
```

d. Iterate through the right-hand side of the rule:

```
>>> r1.rhs()
(NP, VP)
```

e. Generate from the first rhs category

```
>>> x2 = r1.rhs()[0]
>>> x2
NP
>>> options2 = g1.productions(lhs=x2)
>>> options2
[NP -> Det N, NP -> Det N PP, NP -> Name]
>>> r2 = random.choice(options2)
>>> r2
NP -> Det N PP
```

f. Keep going until we reach a terminal symbol

```
>>> x3 = r2.rhs()[0]
>>> x3
Det
>>> options3 = g1.productions(lhs=x3)
>>> options3
[Det -> 'a', Det -> 'an', Det -> 'the', Det -> 'my']
>>> r3 = random.choice(options3)
>>> r3
Det -> 'a'
>>> x4 = r3.rhs()[0]
>>> x4
'a'
```

g. Now we have bottomed out

```
>>> isinstance(x3, str)
False
>>> isinstance(x4, str)
True
>>> print x4
a
```

10. Packaging it up as a recursive function

a. What is the basic loop?

```
# input x
options = g1.productions(lhs=x)
r = random.choice(options)
for y in r.rhs():
    # recurse: y becomes the next input
```

b. What happens when we bottom out? We need to test for that.

```
def generate_from (x):
    if isinstance(x, str):
        print(x)
    else:
        options = g1.productions(lhs=x)
        r = random.choice(options)
        for y in r.rhs():
            generate_from(y)
```

c. Call it

```
>>> generate_from(g1.start())
a
cat
walked
Mary
```

11. Clean up:

a. Define a `Generator` class. Constructor takes a grammar.

```
class Generator (object):
    def __init__ (self, grammar):
        self.grammar = grammar
```

b. `generate_from` becomes a method. Use `self.grammar` instead of `g1`.

c. Modify it so that the words print out on one line:

```
print(x, end=' ')
```

d. Wrap it in `__call__`. Argument n (number of sentences to generate). Prints the newline to terminate each sentence.