00960222: Language, Computation and Cognition
Homework Assignment 5

11 June 2025

1 Using regular expressions to syllabify English words

The Colab notebook for this problem can be found a here.

Write a Python function that uses regular expressions to syllabify English words. You're not expected to come up with something that performs perfectly, as that is not an easy problem, but you should come up with something that performs reasonably well on a range of English words. Although syllabification is really best described on the phonemic representation of words, as available e.g. in English via the CMU Pronouncing Dictionary, for this exercise we will stick with the orthographic representation of English.

Here are some examples of syllabifications of English words:

Word	Syllabification
`i`	`i`
`air`	`air`
`big`	`big`
`strength`	`strength`
`steal`	`steal`
`ideal`	`i deal`
`quiet`	`qui et`
`enter`	`en ter`
`able`	`a ble`
`pandas`	`pan das`
`intake`	`in take`
`capable`	`ca pa ble`
`serendipity`	`se ren di pi ty`

Your function should also work for "nonce" English orthographic words—letter sequences that don't happen to be words, but that could be. For example:

`sneed`	`sneed`
`snoded`	`sno ded`
`ilskig`	`il skig`

In the Colab notebook for this pset, provide your implementation in the section for this problem. For cases your program is not able to capture from the above examples, provide an explanation of why you think that is the case. Further, try your implementation on new English words of your choosing, and describe what kinds of words you find hard to handle. Make sure to include examples where the program succeeds, and examples where it fails.

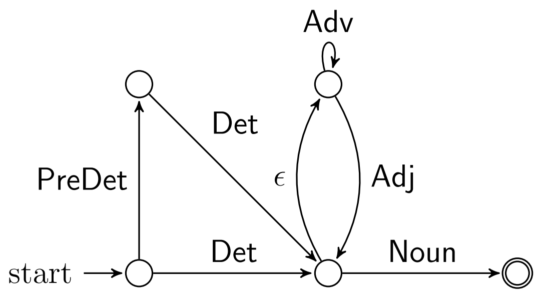
2 Regular expressions and finite-state automata

For drawing finite-state automata, there is a nice web-based tool at http://madebyevan.com/fsm/ that you might want to use for this problem.

Task 1: write a finite-state automaton that accepts and rejects the same strings as the following regular expression, and list the words that it accepts:

```
((in)(disputabl|conceivabl)|(un)(believabl|trustabl))(e|y)
```

Task 2: Write a regular expression that accepts and rejects the same strings as the following finite-state automata, and list some sentences of English exemplifying what the automata/regex accepts. (Make sure that each edge in the automaton is traversed by at least one of your example sentences.)



Part of speech	**Example words**
PreDeterminer	*all*, *only*
Determiner	*the*, *a*
Adverb	*very*, *especially*
Adjective	*big*, *green*
Noun	*woman*, *table*

Questions 3 and 4 are on the next page.

3 Writing context-free grammars

The Colab notebook for this problem can be found at here.

(This question is worth 5 bonus points) This is an exercise in writing context-free grammars (CFGs) to capture generalizations about natural language syntax. You can use an automatic parser available in `NLTK` to check whether your grammar accounts for the key generalizations. To get you going, consider the following grammar that we covered in the CFG lecture notes:

NP $\rightarrow$ Det N	N $\rightarrow$ woman
NP $\rightarrow$ NP PP	N $\rightarrow$ umbrella
PP $\rightarrow$ P NP	N $\rightarrow$ street
Det $\rightarrow$ a	P $\rightarrow$ about
Det $\rightarrow$ an	P $\rightarrow$ with
N $\rightarrow$ joke	P $\rightarrow$ on

Running the following code will parse the string *a joke about the woman with an umbrella on the street* with start symbol (i.e., goal category) NP, generating the five parses that we saw in the CFG lecture notes. Note that `NLTK`'s `CFG.fromstring()` function takes the left-hand-side of the first rule listed as the goal category, and allows multiple rewrites for a single category to be expressed with a disjunction on the right-hand side of a single rule, so that the rule `X -> Y1 ... Ym | Z1 ... Zn` is shorthand for the two rules $X \rightarrow Y_1 \ldots Y_m$ and $X \rightarrow Z_1 \ldots Z_n$.

```python
import nltk
from nltk import Nonterminal, nonterminals, Production, CFG

grammar = CFG.fromstring("""
NP -> Det N | NP PP
PP -> P NP
Det -> 'the' | 'a' | 'an'
N -> 'joke' |'woman' | 'umbrella' | 'street'
P -> 'about' | 'with' | 'on'
""")

parser = nltk.parse.BottomUpChartParser(grammar)
sentence = ['a', 'joke', 'about', 'the', 'woman',
    'with', 'the', 'umbrella', 'on', 'the', 'street']
for tree in parser.parse(sentence):
    tree.pretty_print()
```

1. Write a context-free grammar that will capture the structural ambiguity in the sentence *They are flying planes.* Your grammar should respect the facts that (i) an NP

should be substitutable with a pronoun given the right context; and (ii) a verb and an immediately following NP can combine to form a VP. You can check your work with the `NLTK` parser to make sure that your grammar behaves the way you think it will behave.

2. Extend your grammar to capture the structural ambiguity in the sentence *Flying planes can be dangerous.* (**Hint:** a non-finite VP can serve as the subject of an English sentence, such as in the sentences *To err is human* or *Defeated by the Miami Heat is not how I expected the Milwaukee Bucks to finish in the NBA playoffs*, and it is OK to use a unary rewrite rule with a right-hand-side element that is a phrasal category. See SLP3 Section 12.3.1 for an example of this, though it is a different unary rewrite than you would use for this problem.)

3. The ambiguity of the preceding sentence is eliminated if you change *can be* to either *is* or *are*. Why? Modify your grammar so that it captures this disambiguation effect.

4 Argument structure and unbounded dependencies in context-free grammars

(This question is worth 5 bonus points)

Let's start with the following grammar in `NLTK` style (i.e., `X -> Y1 ... Ym | Z1 ... Zn` means that there are two separate rules, `X -> Y1 ... Ym` and `X -> Z1 ... Zn`):

```
S       → NP VP
NP      → Det N | Pronoun
VP      → V
VP      → V NP
VP      → V SBAR
SBAR    → WHNP S
SBAR    → COMP S
COMP    → 'that' |
WHNP    → 'who' | 'what'
Det     → 'the' | 'a' | 'an' | 'my' | 'your' | 'her' | 'his' | 'their'
N       → 'joke' | 'women' | 'umbrella' | 'street' | 'apple'
Pronoun → 'I' | 'you' | 'she' | 'he' | 'they'
V       → 'slept' | 'devoured' | 'know' | 'said' | 'know'
```

(**Note:** the rule `COMP -> 'that' |` expands out to `COMP -> 'that'` and `COMP ->` , the latter of which is `NLTK`'s way of expressing an empty rewrite (i.e. it's equivalent to `COMP` $\rightarrow \epsilon$).

This grammar will give correct parses for sentences like:

(1) I devoured the apple

(2) I said you slept

(3) you know what I devoured

However, it will incorrectly accept sentences like:

(4) *I slept that you said

(5) *the women devoured

(6) *I know what you said the joke

and incorrectly reject sentences like:

(7) you know who slept

(8) the women know who I said devoured the apple

1. Revise the grammar so that it correctly accounts for the different **argument structures** of the different verbs. Your grammar should now correctly reject (5) and (6) but incorrectly reject (3).

2. **Want a challenge?** Examples like (8) involve *unbounded dependency* constructions as covered in class. Implement meta-rules, using S, NP, VP, and SBAR as your *basic categories*, and the non-terminal rewrites in the grammar as your *basic rules*, to add a set of new *derived categories* and *derived rules* to your grammar. Your grammar should now correctly accept and reject all the above examples.