# LING/C SC/PSYC 438/538

Lecture 15

Sandiway Fong

# Today's Topics

- Homework 9 review
- Ungraded regex exercises
- Regex recursion

# Homework 9 Review

- First, notice I said you may assume the patterns:
  - the *noun*$_1$ *verb* the *noun*$_2$ ➠ *verb*(*noun*$_1$, *noun*$_2$)
  - the *noun*$_1$ who *verb* the *noun*$_2$ ➠ *verb*(*noun*$_1$, *noun*$_2$)
- Perl regex patterns:
  - `/the (\w+) (\w+) the (\w+)/` ➠ `print "$2($1, $3)"`
  - `/the (\w+) who (\w+) the (\w+)/` ➠ `print "$2($1, $3)"`

# Homework 9 Review

Perl regex pattern testing:

- `perl –le '$_ = qq/@ARGV/; /the (\w+) (\w+) the (\w+)/; print "$2($1, $3)"' the woman encountered the boy who encountered the girl`
- `encountered(woman, boy)`
- `perl –le '$_ = qq/@ARGV/; /the (\w+) who (\w+) the (\w+)/; print "$2($1, $3)"' the woman encountered the boy who encountered the girl`
- `encountered(boy, girl)`

# Homework 9 Review

- Next thing to notice is the regex overlap:

```
• the woman encountered the boy   who encountered the girl who found the man
• the (\w+) (\w+)       the (\w+)
•                       the (\w+) who (\w+)       the (\w+)
•                                                 the (\w+) who (\w+) the (\w+)
```

- Recall regex matching goes from left to right (*keeping track of a pointer*)
- We want to iterate this matching using the g (global) flag
- One solution: make the overlapping part a lookahead (so the pointer is not advanced):
  - i.e. `(?=the (\w+))`

# Homework 9 Review

- Two regex patterns:
  - `/the (\w+) (\w+) the (\w+)/` ➠ `print "$2($1, $3)"`
  - `/the (\w+) who (\w+) the (\w+)/` ➠ `print "$2($1, $3)"`
- One regex pattern:
  - `/the (\w+) (who )?(\w+) the (\w+)/`

You can also use a non- capturing group `(?:`*`regexp`*`)`

- With lookahead:
  - `/the (\w+) (who )?(\w+) (?=the (\w+))/`
- Global match using a while loop:
  - `while (/the (\w+) (who )?(\w+) (?=the (\w+))/g) {`
  - `  print "$3($2, $4)"`
  - `}`

# Homework 9 Review

- Example:

```
$ perl –le '$_ = qq/@ARGV/; while (/the (\w+) (who )?(\w+) (?=the
(\w+))/g){ print "$3($1, $4)"}' the woman encountered the boy
```

```
encountered(woman, boy)
```

```
$ perl –le '$_ = qq/@ARGV/; while (/the (\w+) (who )?(\w+) (?=the
(\w+))/g){ print "$3($1, $4)"}' the woman encountered the boy who
encountered the girl
```

```
encountered(woman, boy)
encountered(boy, girl)
```

# Homework 9 Review

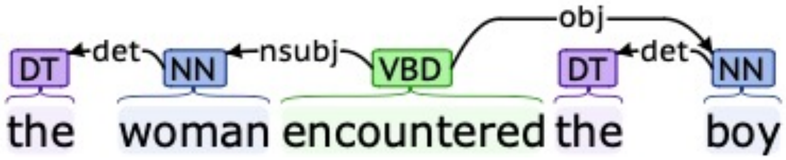
- Example:

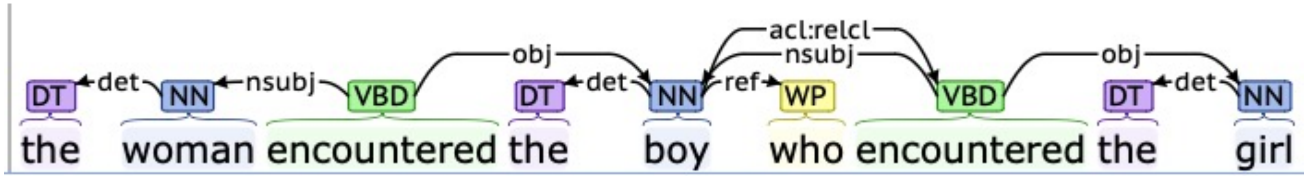
```
$ perl -le '$_ = qq/@ARGV/; while (/the (\w+) (who )?(\w+) (?=the
(\w+))/g){ print "$3($1, $4)"}' the woman encountered the boy who
encountered the girl who found the man
encountered(woman, boy)
encountered(boy, girl)
found(girl, man)
```

```
$ perl -le '$_ = qq/@ARGV/; while (/the (\w+) (who )?(\w+) (?=the
(\w+))/g){ print "$3($1, $4)"}' the woman encountered the boy who
encountered the girl who found the man who chased the cat
encountered(woman, boy)
encountered(boy, girl)
found(girl, man)
chased(man, cat)
```

# CoreNLP

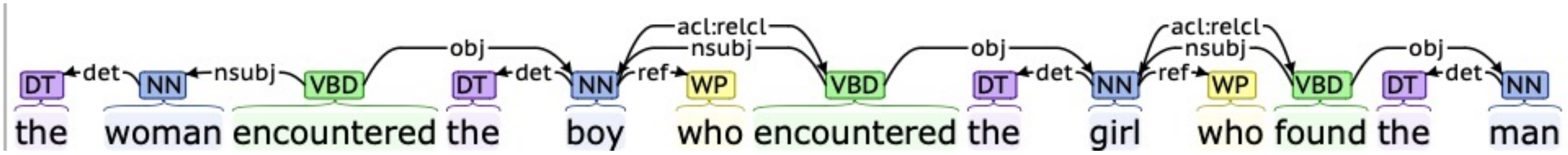
1. the woman encountered the boy
2. the woman encountered the boy who encountered the girl

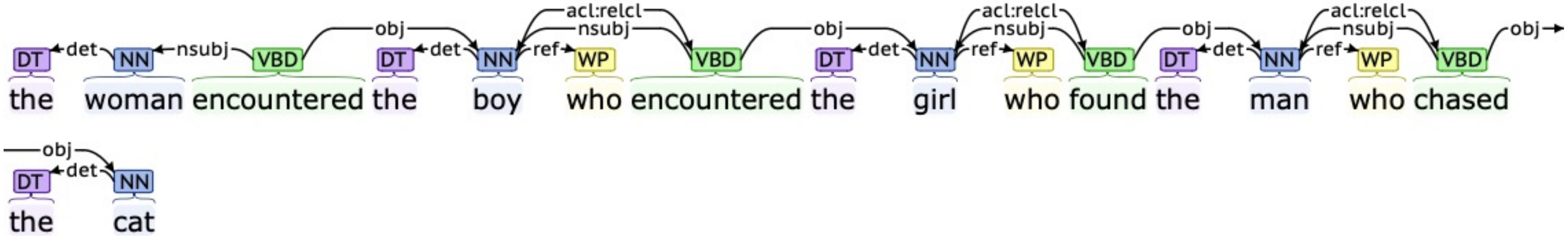




# CoreNLP

3. the woman encountered the boy who encountered the girl who found the man
4. the woman encountered the boy who encountered the girl who found the man who chased the cat



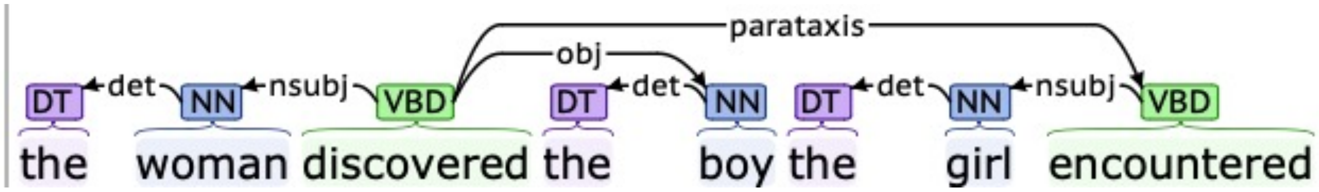


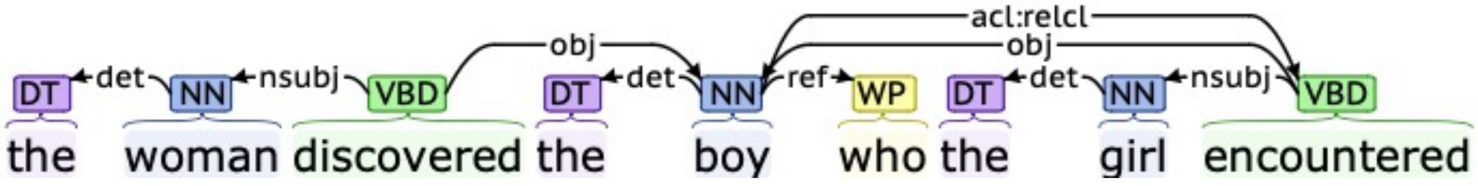
# Homework 9 Review

- Q1 (`nsubj` relativization):
  - the woman encountered the boy who encountered the girl
  - *relative pronoun is obligatory*
  - *the woman encountered the boy encountered the girl
- Q2 (`dobj` relativization):
  - the woman discovered the boy the girl encountered
  - the woman discovered the boy who the girl encountered
  - *relative pronoun is optional*

# CoreNLP

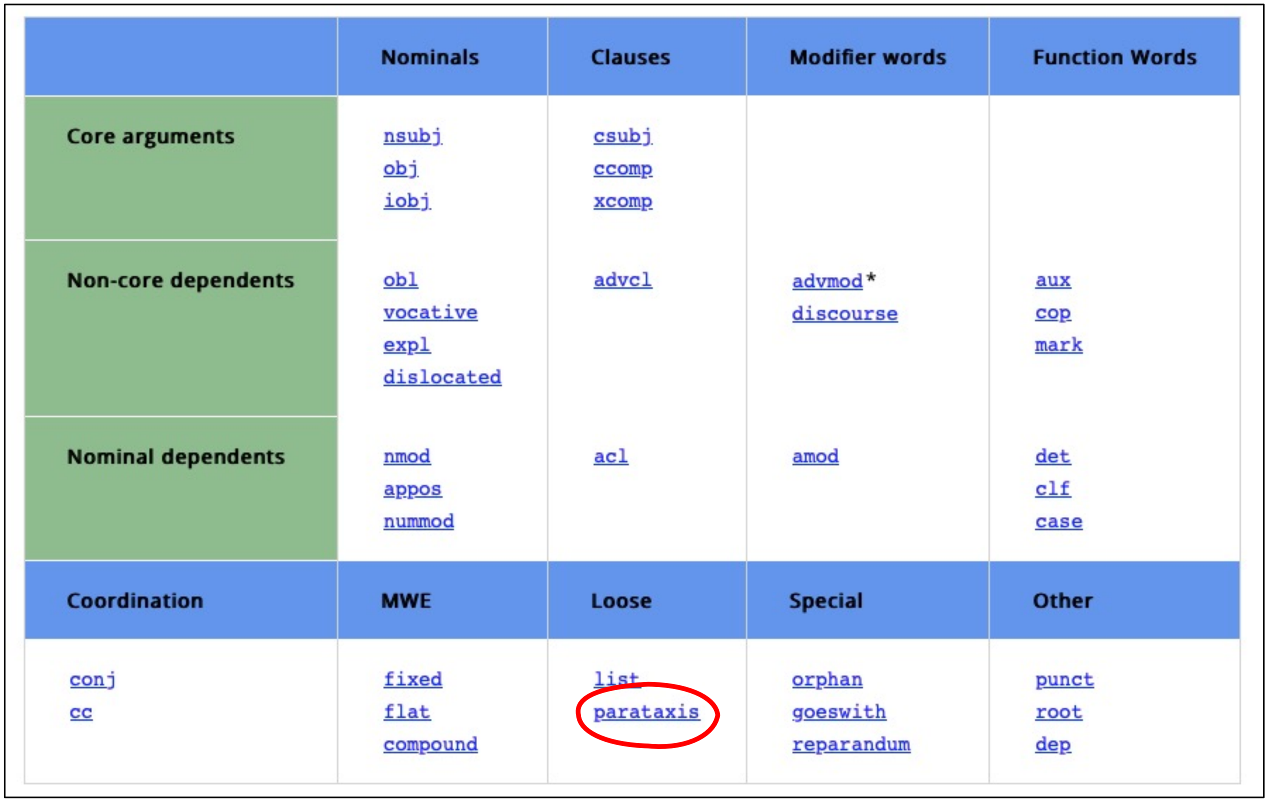
5. the woman discovered the boy the girl encountered


parataxis
obj
DT det NN nsubj VBD DT det NN DT det NN nsubj VBD
the woman discovered the boy the girl encountered


acl:relcl
obj
obj
DT det NN nsubj VBD DT det NN ref WP DT det NN nsubj VBD
the woman discovered the boy who the girl encountered

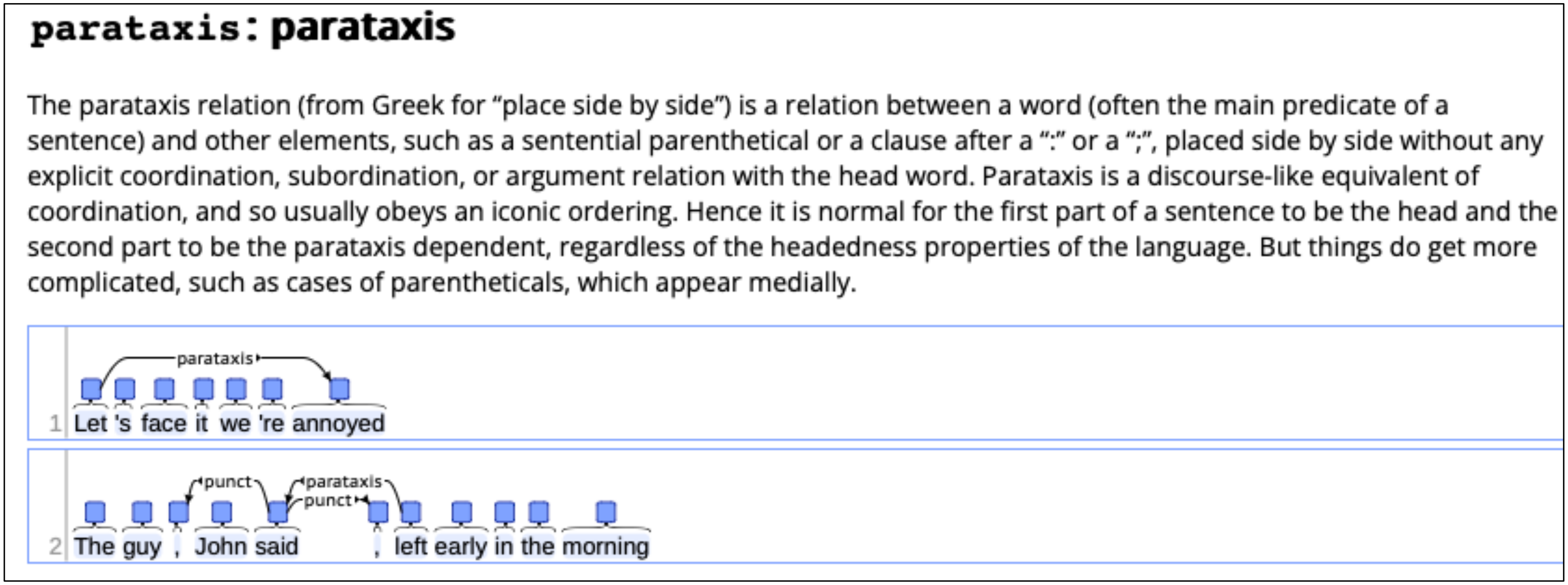
# Background: Universal Dependencies

https://universaldependencies.org/u/dep/index.html



| | Nominals | Clauses | Modifier words | Function Words |
|---|---|---|---|---|
| **Core arguments** | nsubj<br>obj<br>iobj | csubj<br>ccomp<br>xcomp | | |
| **Non-core dependents** | obl<br>vocative<br>expl<br>dislocated | advcl | advmod*<br>discourse | aux<br>cop<br>mark |
| **Nominal dependents** | nmod<br>appos<br>nummod | acl | amod | det<br>clf<br>case |
| **Coordination** | **MWE** | **Loose** | **Special** | **Other** |
| conj<br>cc | fixed<br>flat<br>compound | list<br>parataxis | orphan<br>goeswith<br>reparandum | punct<br>root<br>dep |

# Background: Universal Dependencies



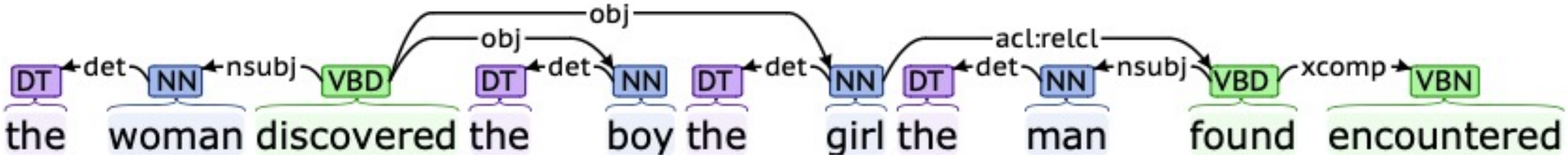
**parataxis: parataxis**

The parataxis relation (from Greek for “place side by side”) is a relation between a word (often the main predicate of a sentence) and other elements, such as a sentential parenthetical or a clause after a “:” or a “;”, placed side by side without any explicit coordination, subordination, or argument relation with the head word. Parataxis is a discourse-like equivalent of coordination, and so usually obeys an iconic ordering. Hence it is normal for the first part of a sentence to be the head and the second part to be the parataxis dependent, regardless of the headedness properties of the language. But things do get more complicated, such as cases of parentheticals, which appear medially.

1
parataxis
Let 's face it we 're annoyed
2
punct
parataxis
punct
The guy , John said , left early in the morning

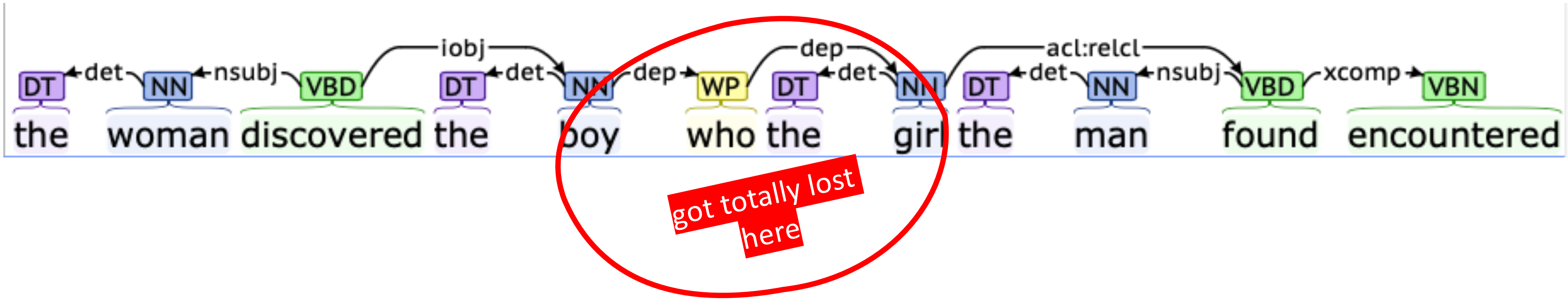
# CoreNLP

6. the woman discovered the boy the girl the man found encountered



# CoreNLP

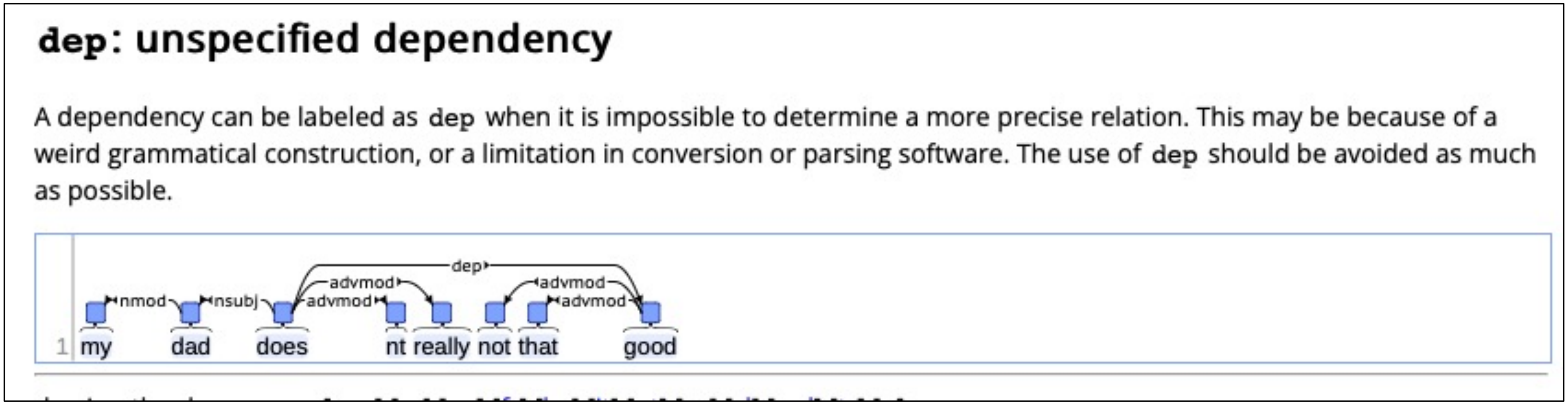
6. the woman discovered the boy who the girl the man found encountered



got totally lost here

# Background: Universal Dependencies

- *i.e. when we have no clue what's going on!*

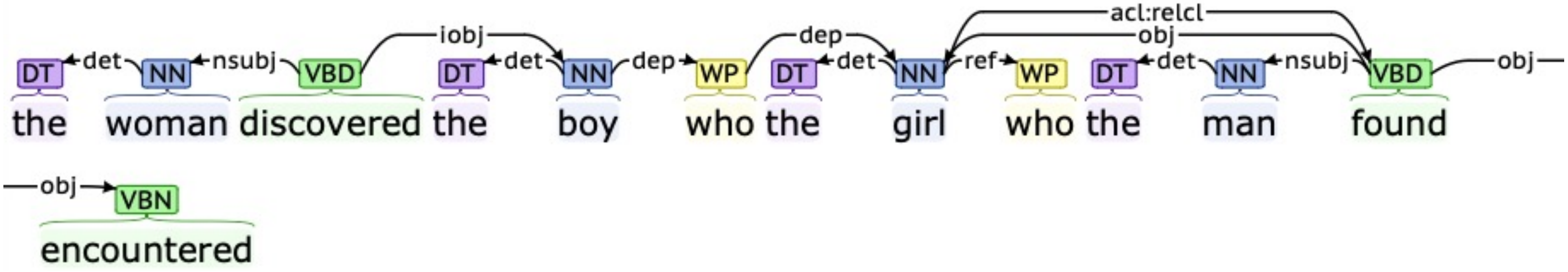


## dep: unspecified dependency

A dependency can be labeled as dep when it is impossible to determine a more precise relation. This may be because of a weird grammatical construction, or a limitation in conversion or parsing software. The use of dep should be avoided as much as possible.

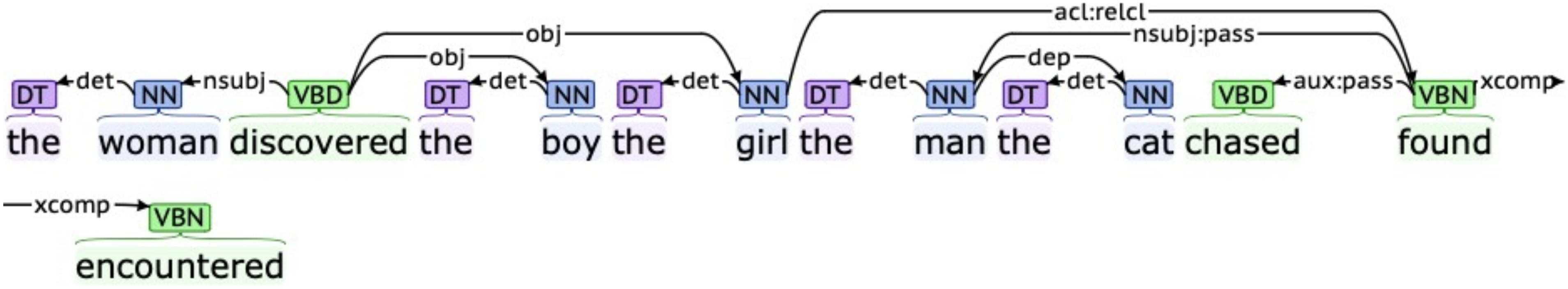
# CoreNLP

6. the woman discovered the boy who the girl who the man found encountered

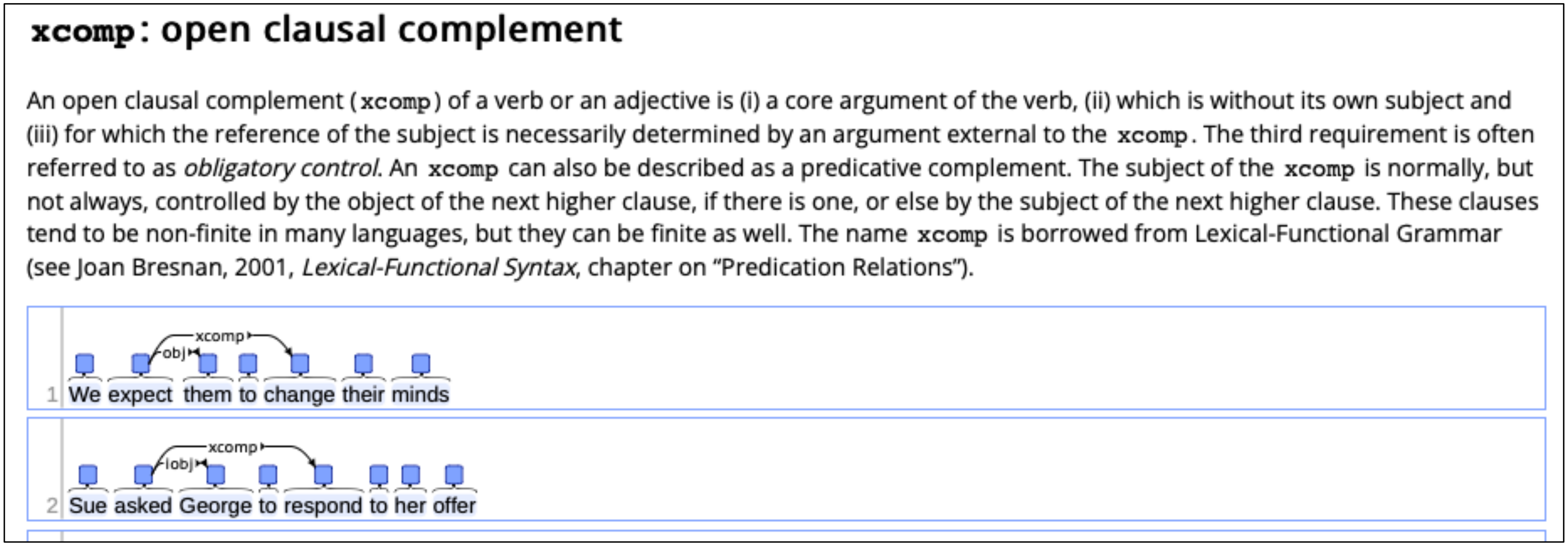


# CoreNLP

7. the woman discovered the boy the girl the man the cat chased found encountered



# Background: Universal Dependencies



## xcomp: open clausal complement

An open clausal complement (xcomp) of a verb or an adjective is (i) a core argument of the verb, (ii) which is without its own subject and (iii) for which the reference of the subject is necessarily determined by an argument external to the xcomp. The third requirement is often referred to as *obligatory control*. An xcomp can also be described as a predicative complement. The subject of the xcomp is normally, but not always, controlled by the object of the next higher clause, if there is one, or else by the subject of the next higher clause. These clauses tend to be non-finite in many languages, but they can be finite as well. The name xcomp is borrowed from Lexical-Functional Grammar (see Joan Bresnan, 2001, *Lexical-Functional Syntax*, chapter on "Predication Relations").

1 We expect them to change their minds

2 Sue asked George to respond to her offer

# Homework 9 Review

Center-embedding (*distance problem*):

2. the woman discovered the boy the girl encountered

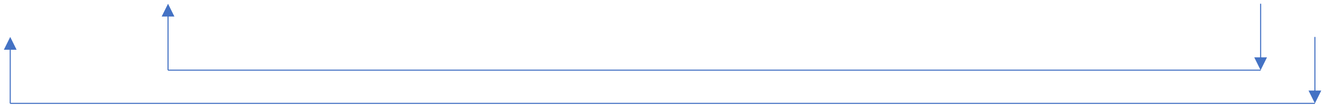
   the woman discovered [ the boy$_{\text{OBJ}}$ the girl$_{\text{NSUBJ}}$ encountered$_{\text{(girl, boy)}}$ ]

3. the woman discovered the boy the girl the man found encountered

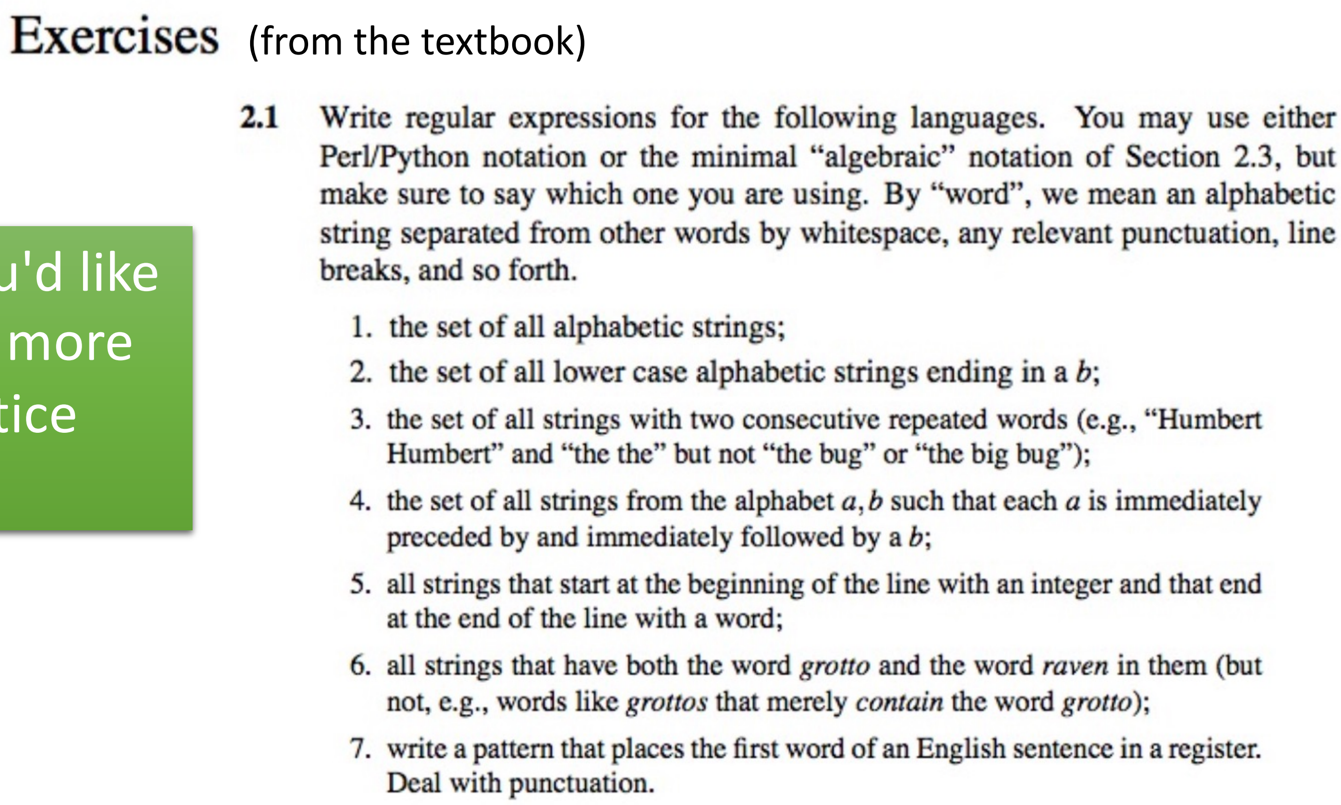
   the woman discovered [ the boy$_{\text{OBJ}}$ [ the girl$_{\text{DOBJ}}$ the man$_{\text{NSUBJ}}$ found$_{\text{(man, girl)}}$ ] encountered$_{\text{(girl, boy)}}$ ]

4. the woman discovered the boy the girl the man the cat chased found encountered

the woman discovered [ the boy [ the girl [ the man$_{\text{OBJ}}$ the cat$_{\text{NSUBJ}}$ chased$_{\text{(cat, man)}}$ ] found$_{\text{(man, girl)}}$ ] encountered$_{\text{(girl, boy)}}$ ]



# Ungraded regex exercises



## Exercises (from the textbook)

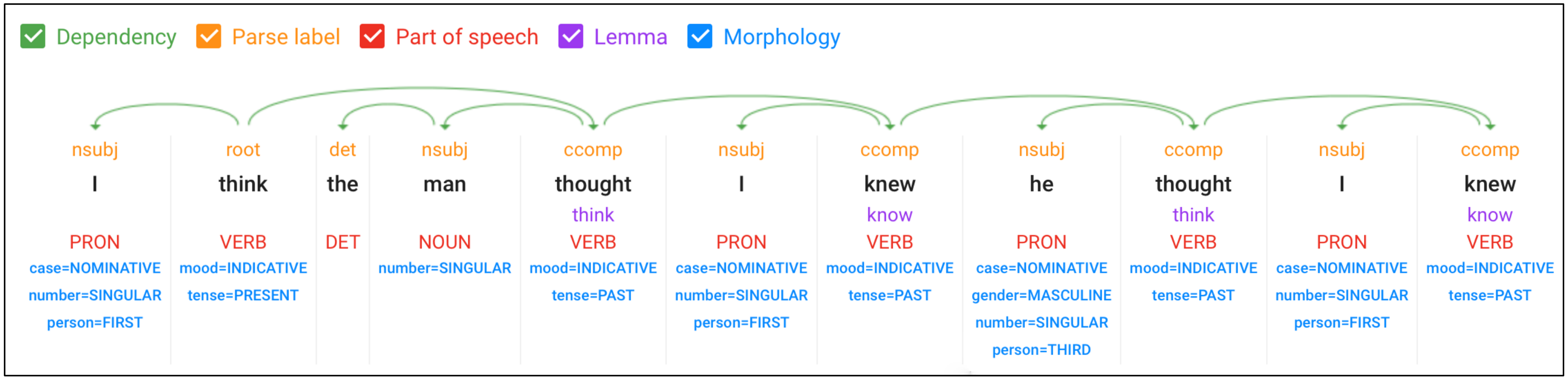
**2.1** Write regular expressions for the following languages. You may use either Perl/Python notation or the minimal "algebraic" notation of Section 2.3, but make sure to say which one you are using. By "word", we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.

1. the set of all alphabetic strings;
2. the set of all lower case alphabetic strings ending in a *b*;
3. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug");
4. the set of all strings from the alphabet *a*, *b* such that each *a* is immediately preceded by and immediately followed by a *b*;
5. all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;
6. all strings that have both the word *grotto* and the word *raven* in them (but not, e.g., words like *grottos* that merely *contain* the word *grotto*);
7. write a pattern that places the first word of an English sentence in a register. Deal with punctuation.

If you'd like a bit more practice …

# Recursion

- The concept of recursion:
  - *I think the man thought I knew he thought I knew*
  - [$_S$ I think [$_S$ the man thought [$_S$ I knew [$_S$ he thought … ]]]] (S = sentence/clause)
- Constituent structure (embedding – *potentially indefinitely*)
- There may be performance limitations on types of embedding
- Dependency structure (chaining `ccomp` relations)



☑ Dependency ☑ Parse label ☑ Part of speech ☑ Lemma ☑ Morphology

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| nsubj | root | det | nsubj | ccomp | nsubj | ccomp | nsubj | ccomp | nsubj | ccomp |
| I | think | the | man | thought | I | knew | he | thought | I | knew |
| | | | | think | | know | | think | | know |
| PRON | VERB | DET | NOUN | VERB | PRON | VERB | PRON | VERB | PRON | VERB |
| case=NOMINATIVE<br>number=SINGULAR<br>person=FIRST | mood=INDICATIVE<br>tense=PRESENT | | number=SINGULAR | mood=INDICATIVE<br>tense=PAST | case=NOMINATIVE<br>number=SINGULAR<br>person=FIRST | mood=INDICATIVE<br>tense=PAST | case=NOMINATIVE<br>gender=MASCULINE<br>number=SINGULAR<br>person=THIRD | mood=INDICATIVE<br>tense=PAST | case=NOMINATIVE<br>number=SINGULAR<br>person=FIRST | mood=INDICATIVE<br>tense=PAST |

# Recursion

- The concept of recursion:
  - n! = n x (n-1)! for $n \in \mathbb{N}$, and 0! = 1 (*factorial function*)

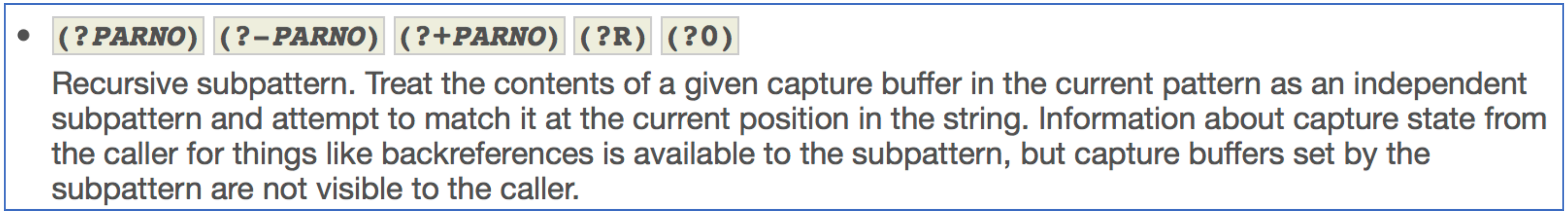
  can be defined non-recursively equivalently as:
  - $n! = \prod_{i=1}^{n} i$ (*product-based factorial function*)

# Regex Recursion

- Example: palindrome words
  - e.g. *I*, *dad*, *noon*, *kayak*, *redder*, *racecar* and *redivider*
  - or phrases (if we ignore white space and punctuation):
  - e.g. *Was it a car or a cat I saw?*
- Normally, you can't write a regex for palindromes. Why?
  - Fundamentally, it involves embedding, e.g. *the use of a stack*
- **Perl** regexs can because we can use backreferences **recursively**.
- regex **recursion** refers to the ability to repeatedly embed regexs using:
  - `(?Group–Number)`

# Regex Recursion

- Program: `(?group-ref)`

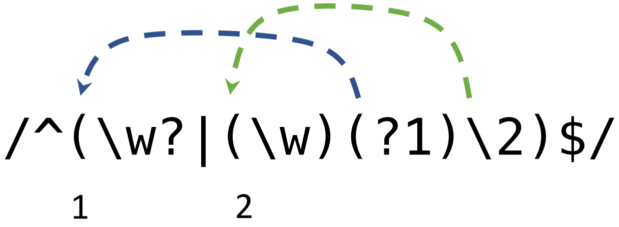


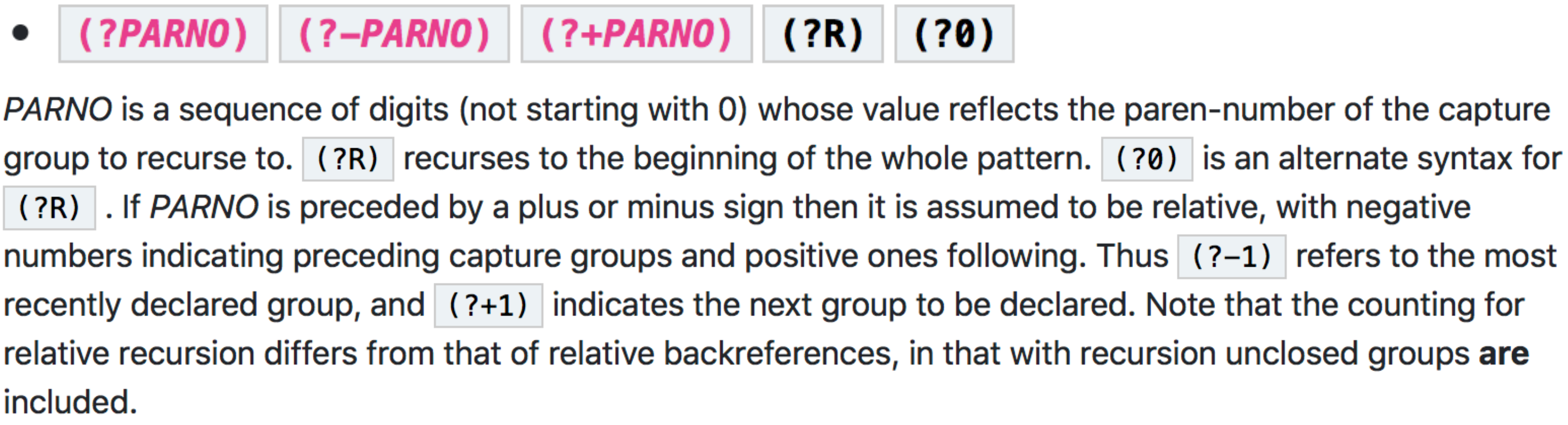
- `(?PARNO)` `(?-PARNO)` `(?+PARNO)` `(?R)` `(?0)`

  Recursive subpattern. Treat the contents of a given capture buffer in the current pattern as an independent subpattern and attempt to match it at the current position in the string. Information about capture state from the caller for things like backreferences is available to the subpattern, but capture buffers set by the the subpattern are not visible to the caller.

```
perl -e '$word = shift; print $word; print " not" if $word !~
/^(\w?|(\w)(?1)\2)$/; print " a palindrome\n"' kayak
kayak a palindrome
perl -e '$word = shift; print $word; print " not" if $word !~
/^(\w?|(\w)(?1)\2)$/; print " a palindrome\n"' abacus
abacus not a palindrome
```

# Regex Recursion

- 
  `/^(\w?|(\w)(?1)\2)$/`

- 
  `(?PARNO)` `(?-PARNO)` `(?+PARNO)` `(?R)` `(?0)`

  *PARNO* is a sequence of digits (not starting with 0) whose value reflects the paren-number of the capture group to recurse to. `(?R)` recurses to the beginning of the whole pattern. `(?0)` is an alternate syntax for `(?R)` . If *PARNO* is preceded by a plus or minus sign then it is assumed to be relative, with negative numbers indicating preceding capture groups and positive ones following. Thus `(?-1)` refers to the most recently declared group, and `(?+1)` indicates the next group to be declared. Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed groups **are** included.

# Regex Recursion

- Successful match with *kayak*

```
 /^(\w?|(\w)(?1)\2)$/  |kayak
1.        k      k     k|ayak
2.        a      a     ka|yak
3.   y                 kay|ak
```

# Regex Recursion

- Failed match with *abacus*

| | | | | |
|---|---|---|---|---|
| `/^(\w?|(\w)(?1)\2)$/` | | | \|abacus | |
| 1. | a | a | a\|bacus | a |
| 2. | b | b | ab\|acus | ba |
| 3. | a | a | aba\|cus | aba |
| 4. | c | c | abac\|us | caba |
| 5. … | | | | |

# Regex Recursion

-
  ```
  perl –e '$word = shift; print $word; print " not" if $word !~
  /^(\w?|(\w)(?1)\2)$/; print " a palindrome\n"' noon
  ```

-
  ```
  noon a palindrome
  ```

-
  ```
  perl –e '$word = shift; print $word; print " not" if $word !~
  /^(\w?|(\w)(?1)\2)$/; print " a palindrome\n"' I
  ```

-
  ```
  I a palindrome
  ```

-
  ```
  perl –e '$word = shift; print $word; print " not" if $word !~
  /^(\w?|(\w)(?1)\2)$/; print " a palindrome\n"'
  ```

```
 a palindrome
```

# Regex Recursion

- Successful match with *noon*

```
 /^(\w?|(\w)(?1)\2)$/  |noon
1.        n      n      n|oon
2.        o      o      no|on
3.   ε                  no|on
ε = empty string
```

# Context-Free Grammar (CFG)

Assume for now, the alphabet is lowercase English letters.

- 53 (26+26+1) rules for the palindrome grammar:

  1. `P --> λ` (*empty string*)<br>
  2. `P --> t` (*terminal*, for `t` ∈ `[a-z]`, 26 rules)<br>
  …<br>
  28. `P --> a P a`<br>
  …<br>
  53. `P --> z P z` (*another* 26 rules)

- Conceptually simpler…

# Regex Recursion

Python:

```
import re
re.match(r'^(\w?|(\w)(?1)\2)$',"releveler")
```

- *Let's see what happens…*

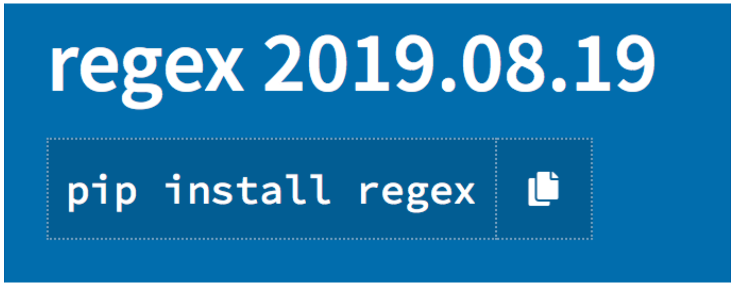
# Regex Recursion

```
python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import re
>>> re.match(r'^(\w?|(\w)(?1)\2)$',"releveler")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/re.py", line 173, in match
    return _compile(pattern, flags).match(string)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/re.py", line 286, in _compile
    p = sre_compile.compile(pattern, flags)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_compile.py", line 764, in compile
    p = sre_parse.parse(p, flags)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_parse.py", line 930, in parse
    p = _parse_sub(source, pattern, flags & SRE_FLAG_VERBOSE, 0)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_parse.py", line 426, in _parse_sub
    not nested and not items))
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_parse.py", line 816, in _parse
    p = _parse_sub(source, state, sub_verbose, nested + 1)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_parse.py", line 426, in _parse_sub
    not nested and not items))
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/sre_parse.py", line 806, in _parse
    len(char) + 1)
re.error: unknown extension ?1 at position 11
```

# Regex Recursion

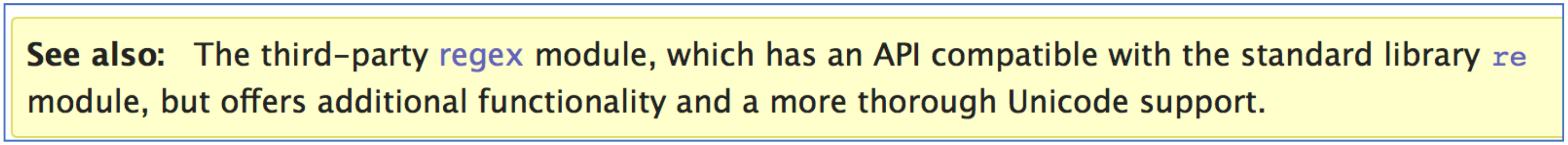
Python: alternate regex module handles recursion

- https://pypi.org/project/regex/



regex 2019.08.19

```
pip install regex
```



**See also:** The third-party regex module, which has an API compatible with the standard library `re` module, but offers additional functionality and a more thorough Unicode support.