

# LING/C SC/PSYC 438/538

Lecture 15

Sandiway Fong

# Today's Topics

- Finish describing Homework 9
- Perl regex; even more power – recursive regex

# Ungraded regex exercises

## Exercises (from the textbook)

If you'd like  
a bit more  
practice

...

- 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.

# Homework 9: Part 2

Question 1: We will compute predicate-argument structure using Perl regex with recursively embedded subject relative clauses:

1. the woman saw the boy
  2. the woman saw the boy who saw the girl
  3. the woman saw the boy who saw the girl who found the man
  4. the woman saw the boy who saw the girl who found the man who chased the cat
- You can assume *the noun* for noun phrases (NPs) and *who* for the relative pronoun (see *next slide*)
  - Write a Perl program using a regex to compute the predicate-argument relations and print them:
    1. `saw(woman, boy)`
    2. `saw(woman, boy) saw(boy, girl)`
    3. `saw(woman, boy) saw(boy, girl) found(girl, man)`
    4. `saw(woman, boy) saw(boy, girl) found(girl, man) chased(man, cat)`

## Homework 9: Part 2

- Code should be general, i.e. you can swap out the verbs and common nouns etc., and it should still work.
- For simplicity, you may assume the patterns:
  - the *noun<sub>1</sub>* *verb* the *noun<sub>2</sub>*                   ⇒ *verb(noun<sub>1</sub>, noun<sub>2</sub>)*
  - the *noun<sub>1</sub>* who *verb* the *noun<sub>1</sub>*                   ⇒ *verb(noun<sub>1</sub>, noun<sub>2</sub>)*
- **Hints:**
  - note the pattern overlap, use lookahead (*?=pattern*)
  - you can collect the words together on the command line into a single string with `$sentence = qq/@ARGV/;`

## Homework 9: Part 2

- Code should be general, i.e. you can swap out the verbs and common nouns etc., and it should still work.
- At the terminal, assume input/output will be something like:

```
$ perl hw9.perl the woman saw the boy
saw(woman, boy)
$ perl hw9.perl the woman saw the boy who saw the girl who found the man
saw(woman, boy)
saw(boy, girl)
found(girl, man)
$ perl hw9.perl the woman saw the boy who saw the girl who found the man who chased the cat
saw(woman, boy)
saw(boy, girl)
found(girl, man)
chased(man, cat)
$ perl hw9.perl the woman saw the boy who saw the girl who found the man who chased the cat who sensed the mouse
saw(woman, boy)
saw(boy, girl)
found(girl, man)
chased(man, cat)
sensed(cat, mouse)
$
```

## Homework 9: Part 2

Question 2: a second type of embedded relative clauses.

Examples:

2. the woman sensed the boy the girl saw
  3. the woman sensed the boy the girl the man found saw
  4. the woman sensed the boy the girl the man the cat chased found saw
- Explain the differences between sentences 2–4 in Q2 vs. Q1 with respect to predicate-argument structure.

## Homework 9: Part 2

Question 3: try [Google Natural Language](#) on the sentences with relative clauses from Q2.

2. the woman sensed the boy the girl saw.
  3. the woman sensed the boy the girl the man found saw.
  4. the woman sensed the boy the girl the man the cat chased found saw.
- Which one(s) does/do Google get wrong?
  - As a human processor, which of 2–4 do you find very difficult to parse?



## Homework 9: Part 2

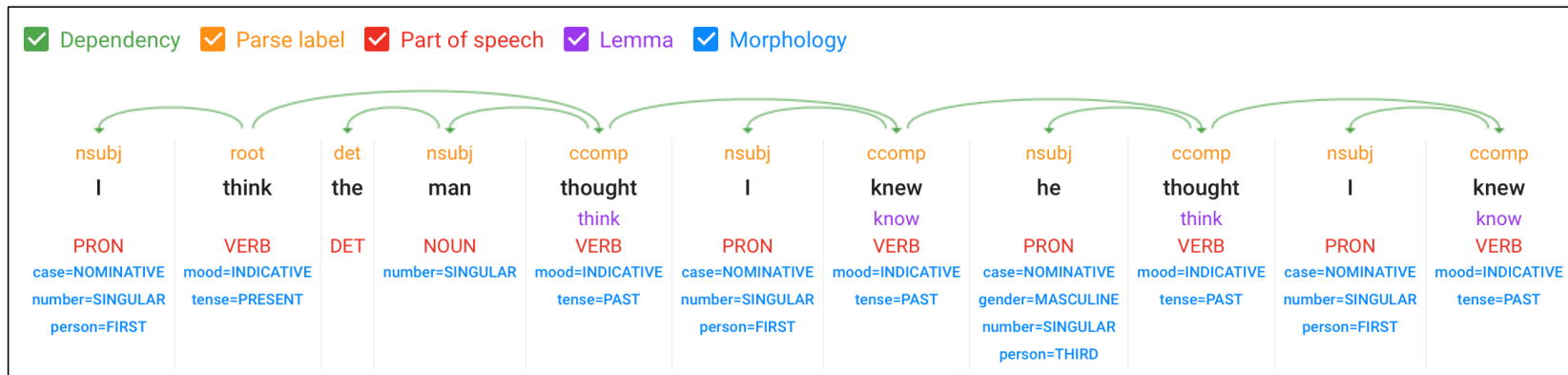
- Extra Credit Question 4: based on what we've learnt so far, do you think it's possible to write a Perl regex program that prints the correct predicate-argument structure for the following examples from Q2?

Embedded relative clauses:

2. the woman sensed the boy the girl saw
3. the woman sensed the boy the girl the man found saw
4. the woman sensed the boy the girl the man the cat chased found saw

# 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)



# Recursion

- The concept of recursion:
  - $n! = n \times (n-1)!$  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. *l*, *dad*, *noon*, *kayak*, *redder*, *racecar* and *divider*
  - 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

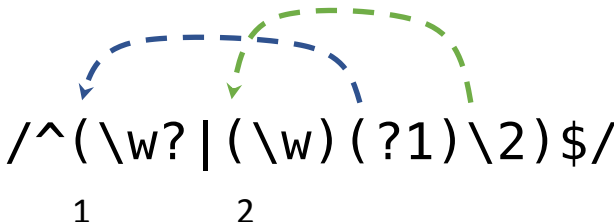
- Program: `(?group-ref)`

- `(?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 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

-   
The diagram shows the regex pattern `/^(\w? | (\w) (?1) \2) $/`. Below the pattern, the numbers 1 and 2 are positioned under the first and second capture groups, respectively. A blue dashed arrow originates from the `(?1)` backreference and points back to the start of the first capture group (group 1). A green dashed arrow originates from the `\2` backreference and points back to the start of the second capture group (group 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.     $\epsilon$                     no | on

$\epsilon$  = *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.  $P \rightarrow \lambda$  (*empty string*)

2.  $P \rightarrow t$  (*terminal, for  $t \in [a-z]$ , 26 rules*)

...

28.  $P \rightarrow a P a$

...

53.  $P \rightarrow z P z$  (*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)$',"reveler")
```

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.