

LING/C SC/PSYC 438/538

Lecture 15

Prof. Sandiway Fong

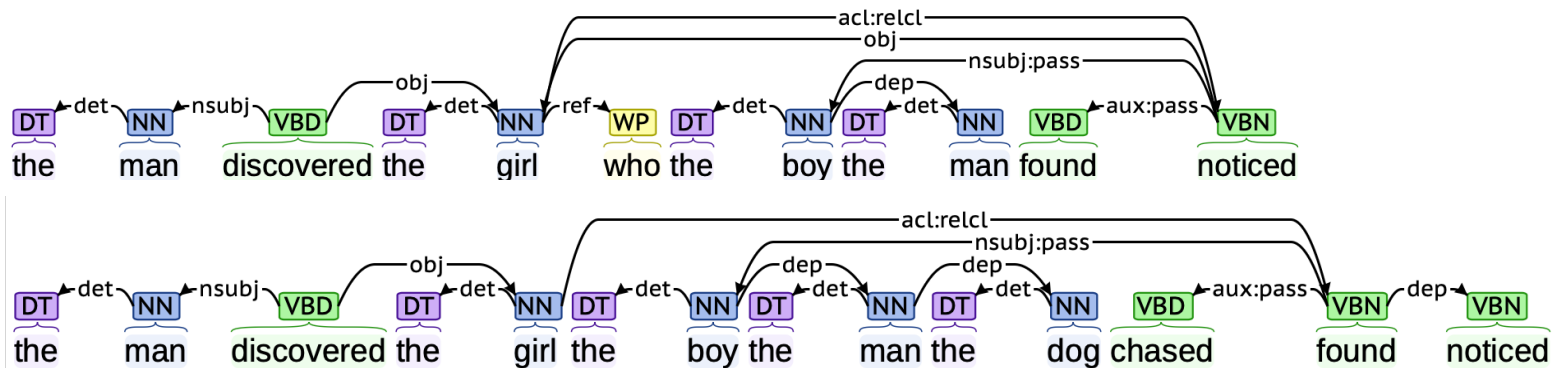
Today's Topics

- **Regex** recursion **unlocks** new pattern-matching powers!
- Also today:
 - How to show a number is **prime** using **regex** (backreferences)!

Last Time

- Subject relative clauses:
 - *the man noticed the girl who noticed the boy who found the person who chased the squirrel*
 - *who-did-what-to-whom*: can use a Perl left-to-right expression
 - `/the (\w+) (who )?(\w+) (?:the (\w+) )/g`
- Object relative clauses:
 - *the man discovered the girl (who) the boy (who) the man (who) the dog chased found noticed*
 - *who-did-what-to-whom*: **cannot** use a Perl left-to-right expression

CoreNLP: problematic analyses



Center-embedding (nested) character:

- *the man discovered the girl (who) the boy (who) the man (who) the dog chased found noticed*

the man discovered [the girl [the boy [the man_{OBJ} the dog_{NSUBJ} chased_(dog,man)] found_(man, girl)] noticed_(boy,girl)]

The man discovered the girl whom the boy whom the person whom the dog chased found noticed. Who noticed whom?

Let's take this one apart carefully — it's a **center-embedded relative clause chain**:

The man discovered [the girl
whom [the boy
whom [the person
whom [the dog chased] found] noticed]].

We'll unpack it **from the inside out**.

The man discovered the girl whom the boy whom the person whom the dog chased found noticed. Who noticed whom?

Let's take this one apart carefully — it's a **center-embedded relative clause chain**:

Step 1: Innermost clause

the dog chased the person

✓ dog → chased → person

Step 2: Next layer out

the person whom the dog chased found the boy

✓ person (chased by dog) → found → boy

Step 3: Next layer

the boy whom the person (whom the dog chased) found noticed the girl

✓ boy (found

Step 4: Main clause

the man discovered the girl (whom the boy ... noticed)

✓ man → discovered → girl

The man discovered the girl whom the boy whom the person whom the dog chased found noticed. Who noticed whom?

Let's take this one apart carefully — it's a **center-embedded relative clause chain**:

The man d
whom
w

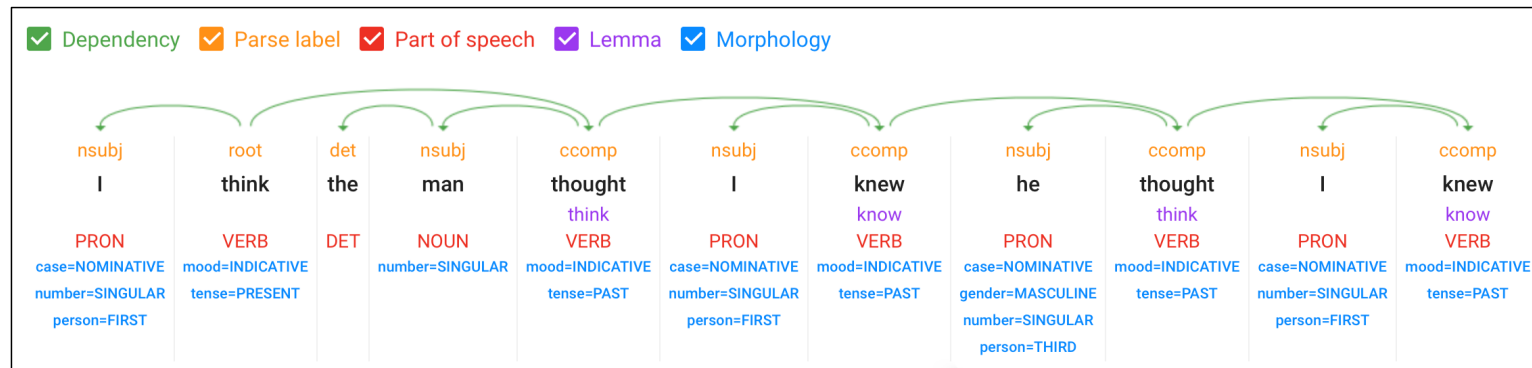
We'll unpack it

So, who noticed whom?

- **The dog** chased **the person**.
- **The person** found **the boy**.
- **The boy** noticed **the girl**.
- **The man** discovered **the girl**.

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 c comp 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*
(also required for our object relative clause structures)
- **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)`

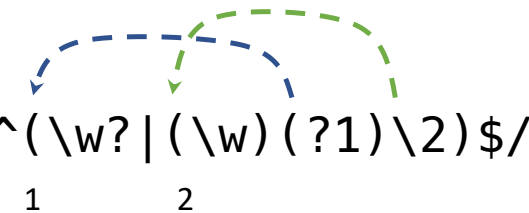
- `(?PARNO)` `(?-PARNO)` `(?+PARNO)` `(?R)` `(?0)`

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)` and points back to the start of the first capture group. A green dashed arrow originates from the `\2` and points back to the start of the second capture group.

- `(?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)\$/

1.	k	k	kayak
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. $P \rightarrow \lambda$ (*empty string*)

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

...

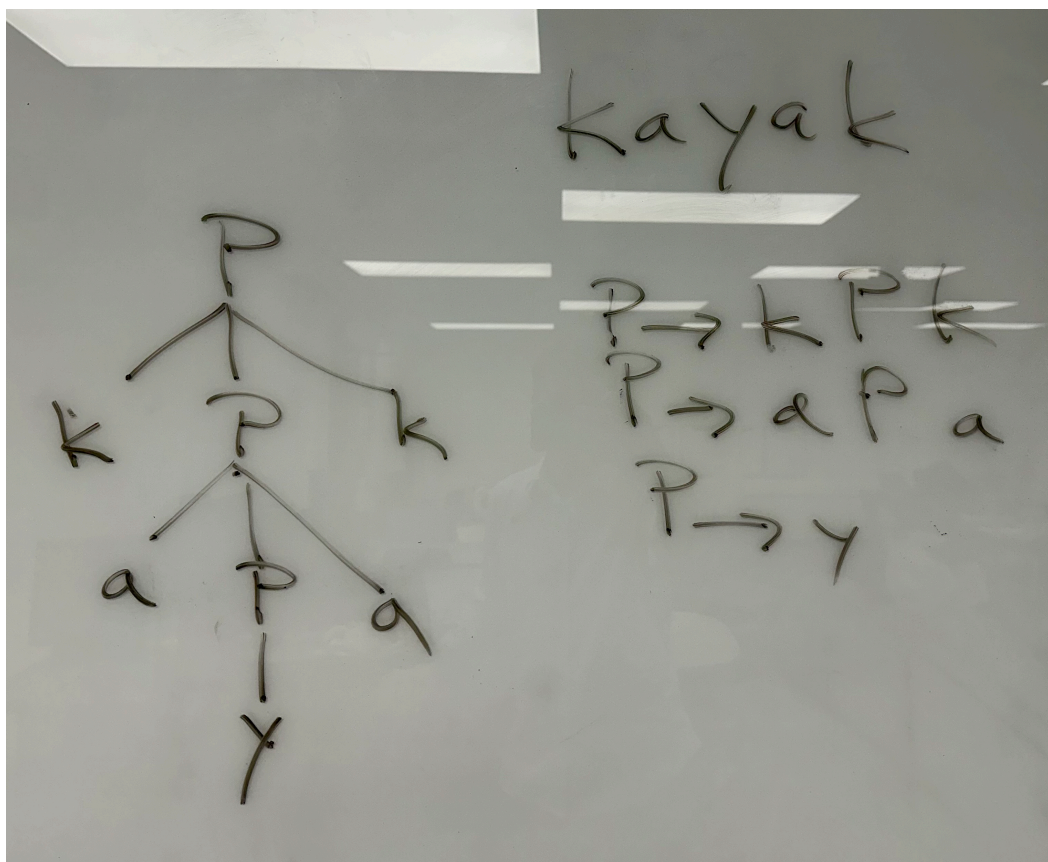
28. $P \rightarrow a P a$

...

53. $P \rightarrow z P z$ (*another 26 rules*)

- Conceptually simpler...

Context-Free Grammar (CFG)



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.

Prime Number Testing using Perl Regular Expressions

- Another example:
 - the set of prime numbers is **not** a **regular** language (*can't do it with FSA/regex*)
 - $L_{\text{prime}} = \{2, 3, 5, 7, 11, 13, 17, 19, 23, \dots\}$

[Prime number - Wikipedia, the free encyclopedia](https://en.wikipedia.org/wiki/Prime_number)

en.wikipedia.org/wiki/Prime_number ▼

A **prime number** (or a **prime**) is a natural number greater than 1 that has no positive divisors other than 1 and itself. A natural number greater than 1 that is not a ...

Turns out, we can use a short Perl regex to determine membership in this set .. and to factorize numbers

`/^(11+?)\1+$/`

Prime Number Testing using Perl regex

- $L = \{1^n \mid n \text{ is prime}\}$ is not a **regular** language
- Keys to making this work:
 - \1 backreference
 - unary notation for representing numbers, e.g.
 - 11111 “five ones” = 5
 - 111111 “six ones” = 6
 - unary notation allows us to factorize numbers by repetitive pattern matching
 - (11)(11)(11) “six ones” = 6
 - (111)(111) “six ones” = 6
 - numbers that can be factorized in this way aren’t prime!
 - no way to get nontrivial subcopies of 11111 “five ones” = 5
- Then `/^(11+?)\1+$/` will match anything that’s greater than 1 that’s not prime



can be proved mathematically
using the Pumping Lemma for
regular languages
(later)

Prime Number Testing using Perl regex

Question: is the non-greedy operator necessary?

- Let's analyze this Perl regex `/^(11+?)\1+$/`
 - `^` and `$` anchor both ends of the strings, forces `(11+?)\1+` to cover the entire string
 - `(11+?)` is the non-greedy (*shortest*) match version of `(11+)`
 - `\1+` provides one or more copies of what we previously matched in `(11+?)`

- Examples:

```
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~  
/^(11+?)\1+$/ ' 101
```

101 prime

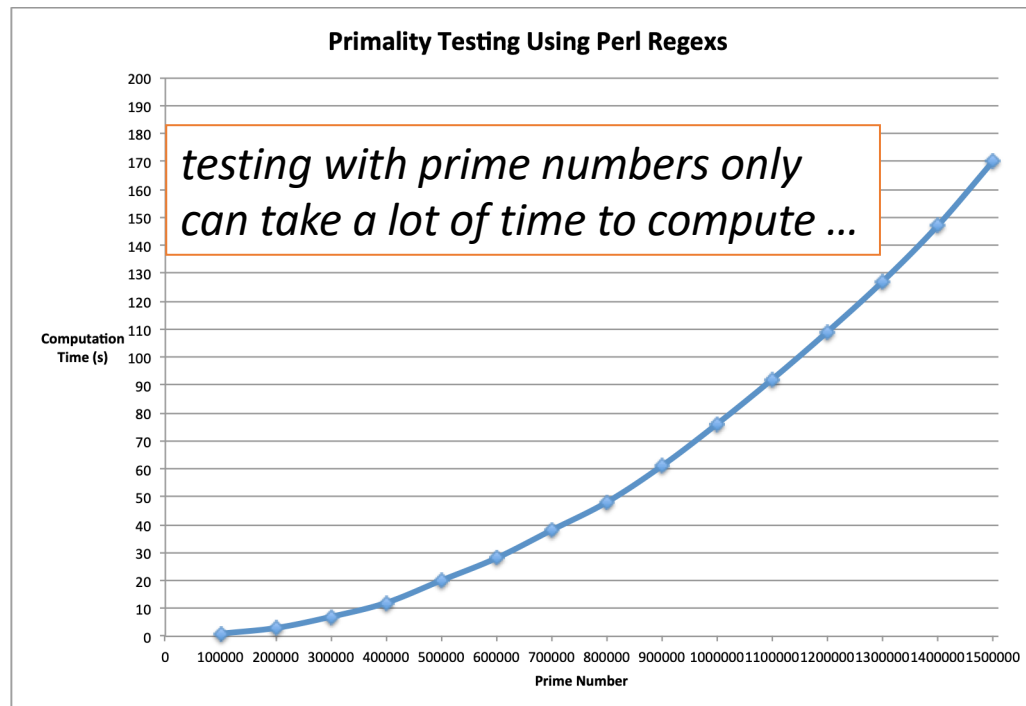
```
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~  
/^(11+?)\1+$/ ' 103
```

103 prime

```
perl -le '$n = shift; $u = "1" x $n; print "$n prime" if $u !~  
/^(11+?)\1+$/ ' 105
```

Prime Number Testing using Perl regex

Prime Numbers
100003
200003
300007
400009
500009
600011
700001
800011
900001
1000003
1100009
1200007
1300021
1400017
1500007



Prime Number Testing using Perl regex

- `/^(11+?)\1+$/` vs. `/^(11+)\1+$/`
- i.e. non-greedy vs. greedy matching
- finds smallest factor vs. largest
 - 90021 factored using 3, not a prime (0 secs)
 - vs.
 - 90021 factored using 30007, not a prime (0 secs)

Puzzling behavior: same output non-greedy vs. greedy
900021 factored using 300007, not a prime (48 secs vs. 13 secs)

Prime Number Testing using Perl regex

- <http://www.xav.com/perl/lib/Pod/perlre.html>

The following standard quantifiers are recognized:

```
*      Match 0 or more times
+      Match 1 or more times
?      Match 1 or 0 times
{n}    Match exactly n times
{n,}   Match at least n times
{n,m}  Match at least n but not more than m times
```

(If a curly bracket occurs in any other context, it is treated as a regular character.) The ``\*'' modifier is equivalent to {0,}, the ``+'' modifier to {1,}, and the ``?' modifier to {0,1}. n and m are limited to integral values less than a preset limit defined when perl is built. This is usually 32766 on the most common platforms.

nearest primes to preset limit



Prime Number Testing using Perl regex

- $32749 \times 3 = 98247$
- $32771 \times 3 = 98313$
- *When preset limit is exceeded: Perl's regex matching fails quietly*
- **Why does it report 32771?**

```
bash-3.2$ perl prime.perl 98247
Time 0: 98247 factored using 3, not a prime
bash-3.2$ perl prime.perl 98313
Time 1: 98313 factored using 32771, not a prime
```

Prime Number Testing using Perl Regular Expressions

- Can also get non-greedy to skip several factors
- Example: pick non-prime 164055 = 3 x 5 x 10937 (prime factorization)

```
bash-3.2$ perl prime.perl 164055
Time 0: 164055 factored using 15, not a prime
bash-3.2$ perl primeg.perl 164055
Time 1: 164055 factored using 54685, not a prime
```

Non-greedy: missed
factors 3 and 5 ...

greedy
version

Because
3 * 54685 = 164055
5 * 32811 = 164055
32766 limit
15 * 10937 = 164055

Prime Number Testing using Perl Regular Expressions

- Results are still right so far though:
 - *wrt.* prime vs. non-prime
- But we predict it will report an incorrect result for
 - 1,073,938,441
 - It should claim (incorrectly) that this is prime since $1073938441 = 32771^2$
 - (32766 is the limit for the number of bundles)

32611	32621	32633	32647	32653	32687	32693	32707	32713	32717
32719	32749	32771	32779	32783	32789	32797	32801	32803	32831
32833	32839	32843	32869	32887	32909	32911	32917	32933	32939

<https://primes.utm.edu/lists/small/10000.txt>