Course Intro

Computational Linguistics (LING 455)

Rutgers University September 3, 2021

Logistics

Welcome to Computational Linguistics (615:455)!

See you ‘here’ Tues and Fri 11am–12:20pm.

Your instructor (me):

Simon Charlow
Associate Professor (and Graduate Director)
Dept of Linguistics
simon.charlow@rutgers.edu

My student support hours/“office” hours are Weds and Fri, 1–2pm.
Zoom link on Canvas. Also available by appointment.

Everything runs through Canvas:

- Readings, assignments, discussions, announcements, grades.

Notifications go to your **Rutgers email** (...`@rutgers`.edu) by default. Make sure you're receiving them (you should already have gotten 2).

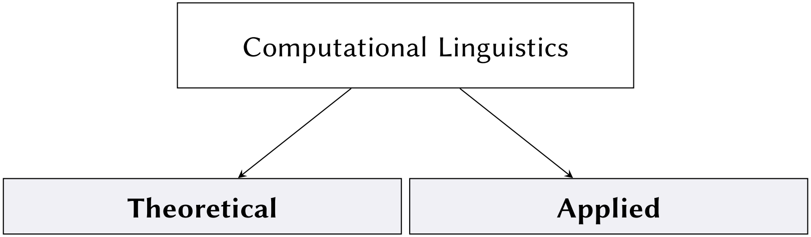
Please **read the syllabus** carefully before you do anything.

- Roadmap for the course
- What's expected of you, and what you can expect of me
- Policies on attendance, late work, and collaboration

What *is* Computational Linguistics?

Computers that take linguistic input and computers that produce linguistic output (and computers that do both) are *everywhere*.

- Segmenting sound streams into words (speech recognition)
- Taking sequence of words to structured representation (parsing)
- Creating new structures/sequences of words (generation)
- Turning sequences of words into sound (speech synthesis)



- How is language computed?
- What are the computational properties of morphology, phonology, syntax, semantics?

- Aka natural language processing
- Designing tools that produce and understand natural language, not necessarily in the way we do.

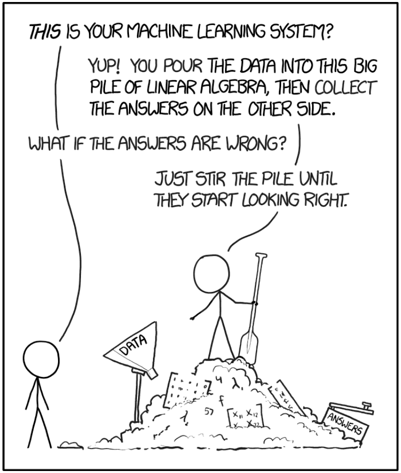
Computational Linguistics is **huge**. And it's arguably one of the most consequential fields in the world at this moment in history.

A big reason for this is the incredible success of Natural Language Processing (NLP) and Machine Learning (ML), which extracts fancy statistical generalizations from huge linguistic data sets.

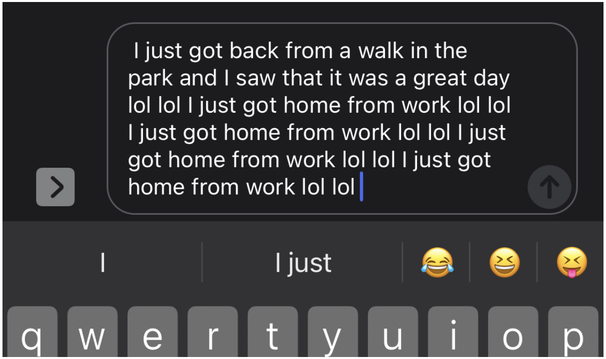
Computational Linguistics is **huge**. And it's arguably one of the most consequential fields in the world at this moment in history.

A big reason for this is the incredible success of Natural Language Processing (NLP) and Machine Learning (ML), which extracts fancy statistical generalizations from huge linguistic data sets.

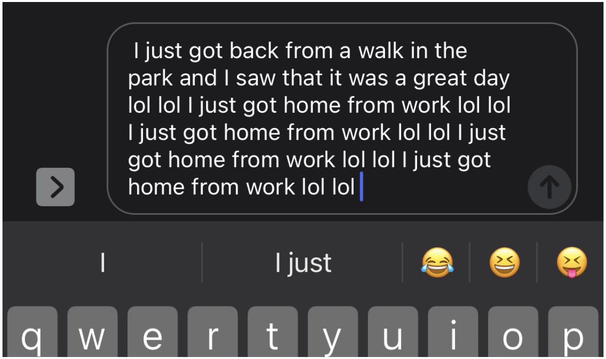
- This is **not** what this course is about! :P
- We're linguists, interested in the structure of language. ML models are sophisticated, extremely data-hungry black boxes.
- NLP is miraculous in many ways. But we want to **understand language**, from a computational perspective. Most everybody (including NLPers) agrees: that's not what (current) NLP does.


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

https://xkcd.com/1838/



I just got back from a walk in the park and I saw that it was a great day lol lol I just got home from work lol lol I just got home from work lol lol I just got home from work lol lol I just got home from work lol lol



I just got back from a walk in the park and I saw that it was a great day lol lol I just got home from work lol lol I just got home from work lol lol I just got home from work lol lol I just got home from work lol lol

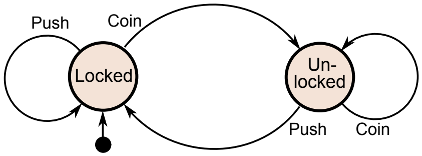
Perhaps computers are not quite ready to take over the world

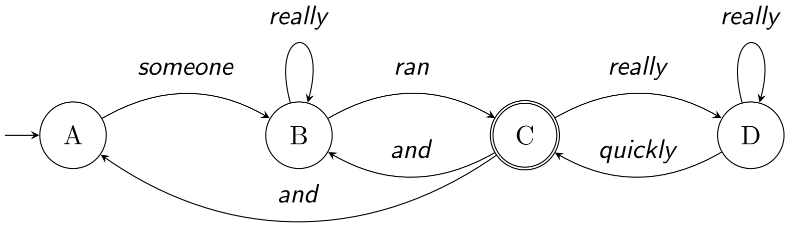
Linguists are preoccupied with the incredible **generativity** of language. The famous words of Wilhelm von Humboldt:

> *[Language] makes infinite use of finite means.*

The infinite creativity of language use is something that must be captured, and ultimately explained.

Finite State Automata (FSAs)





Infinity from loops. Creativity from optional loops.

Phrase structure grammars (PSGs)

A more familiar PSG that represents some of the same information (though by no means identical to the prev FSA):

```
S  -> NP VP
NP -> someone
VP -> ran
VP -> really VP
VP -> VP quickly
VP -> VP AdvP
S  -> S and S
VP -> VP and VP
VP -> VP
```

Do FSA and PSG grammars differ? If they do, how? Does this difference correspond to anything real in natural language?

Might some domains of our linguistic competence be better described in terms of one, and other domains in terms of some other?

We'll address these important questions, and many more.

We'll learn about **data structures** and **algorithms** that allow us to *implement* finite-state grammars, phrase structure grammars, and more, and allows us to compute with them.

We'll see how these techniques can be used both for **generation** of language, and **parsing** of language.

- Think back to your first homework in syntax. How do you associate a syntactic analysis with a string of words that doesn't have any on its sleeve? How do people do this?
- We'll consider implications of different parsing techniques for how people process language (e.g., garden path sentences).

Computation is like language in some ways:

- It's practically bounded in space, time, and memory, yet nonetheless capable of doing an infinite variety of things.
- This behavior is specified, constrained, and ultimately enabled by finite, rule-governed systems.

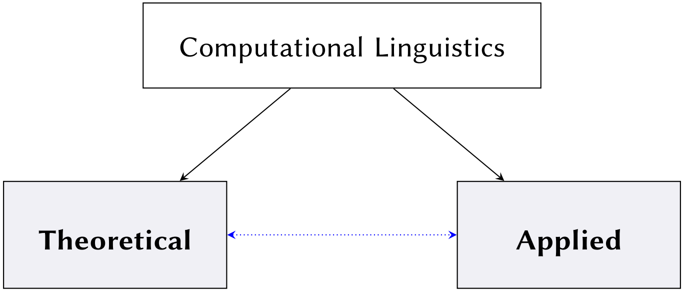
Taking a computational perspective on language offers both practical and scientific benefits:

- It helps us gain a deeper understanding of the computational nature of language, and of linguistic creativity/generativity.
- Once a task is stated coherently and in enough detail, computers can perform boring and/or huge tasks extremely quickly…
    - Enables us to test theories quickly at scale
    - Helps us to clarify and sharpen our thinking

Connections to applied settings

It’s not that our techniques are *opposite* of those used in applied settings, but that our focus as linguists is different.

The techniques and skills you’ll learn in this course will give you a **strong basis** for future explorations in computational linguistics and computer science, and for doing applied work.


Computational Linguistics
Theoretical
Applied

A little bit of Haskell

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>
```

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>
```

```
Prelude> 2+3
5
```

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>

Prelude> 2+3
5

Prelude> take 20 "My very educated mother just served"
"My very educated mot"
```

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>

Prelude> 2+3
5

Prelude> take 20 "My very educated mother just served"
"My very educated mot"

Prelude> drop 20 "My very educated mother just served"
"her just served"
```

Firing up the Haskell interpreter

```
> ghci
GHCi, version 8.10.4: https://www.haskell.org/ghc/
Prelude>

Prelude> 2+3
5

Prelude> take 20 "My very educated mother just served"
"My very educated mot"

Prelude> drop 20 "My very educated mother just served"
"her just served"

Prelude> [n | n <- [1..100], n `mod` 17 == 0]
[17,34,51,68,85]
```

Types in Haskell: preventing many errors

```
Prelude> not True
False

Prelude> not 2
<interactive>:7:5: error:
  • No instance for (Num Bool) arising from the literal ‘2’
  • In the first argument of ‘not’, namely ‘2’
    In the expression: not 2
    In an equation for ‘it’: it = not 2
```

Not all languages work in this way. For example, Javascript:

```
>> !2
false
```

Writing .hs scripts

```haskell
-- W1.hs (a comment!)
-- Edit this with a text editor (VS Code is great!)

myList :: [Integer] -- myList is a list of Integers
myList = [1..100]   -- types can generally be left off

add :: [Integer] -> Integer
add []     = 0
add (n:ns) = n + add ns -- a recursive defnition
                        -- 'add' defined using 'add'!
{- this
      is
    a
      multiline
   comment -}
```

Interpreting .hs scripts

```
> ghci W1.hs
GHCi, version 8.10.4
[1 of 1] Compiling Main ( W1.hs, interpreted )
Ok, one module loaded.

*Main> add myList
5050
```

```
myListEvens = [n | n <- myList, even n]
```

What do you suppose the `length` of `myListEvens` is?

```
myListEvens = [n | n <- myList, even n]
```

What do you suppose the `length` of `myListEvens` is?

```
*Main> length myListEvens
50
```

How would you define `myListOdds`?

```
myListEvens = [n | n <- myList, even n]
```

What do you suppose the `length` of `myListEvens` is?

```
*Main> length myListEvens
50
```

How would you define `myListOdds`?

```
myListEvens = [n | n <- myList, odd n]
```

What's the difference btw `add myListEvens` and `add myListOdds`?

```
myListEvens = [n | n <- myList, even n]
```

What do you suppose the `length` of `myListEvens` is?

```
*Main> length myListEvens
50
```

How would you define `myListOdds`?

```
myListEvens = [n | n <- myList, odd n]
```

What's the difference btw `add myListEvens` and `add myListOdds`?

```
*Main> add myListEvens - add myListOdds
50
```

(The thing to notice is that both lists have 50 numbers, and each of those 50 even numbers is 1 greater than its corresponding odd partner.)

Pattern matching

```haskell
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x) -- equivalent to
                     -- swap p = (snd p, fst p)
-- *Main> swap (True, 3.14)
-- (3.14, True)

firstOdd :: [a] -> Bool
firstOdd [] = False
firstOdd (x:xs) = odd x
-- *Main> [1..100]
-- True
-- *Main> [2..100]
-- False
```

Note: you can't pattern match on (`xs:x`). This has to do with how lists are *constructed*/defined in Haskell. More on this next week.

Your first assignment

Reading for Tuesday Sept 7

Read the ‘Starting Out’ chapter of *Learn You a Haskell* (on Canvas).

It’s about 17pp printed, and a very quick and accessible read.

HW1

1. Install the Haskell platform
2. Do the reading
3. Download and complete `HW1.hs` (very short!)
4. Submit via Canvas (using the Assignments tool) before our next meeting (Tues at 11am).