Lecture 1

Computation and Formalization

Without a doubt the most unfortunate fact about computational linguistics is that its name is *computational linguistics*. The term inherits a dichotomy that is hard to tease apart for the uninitiated: the distinction between computers and computing. While computers are the most common tool for carrying out computations nowadays, they are not what computing is about. Computation, in its barest form, is the principled manipulation of information. When a computer verifies $1 + 1 = 2$, this act of computation is instantiated via a series of electrical impulses that affect some of the millions of transistors that make up its hardware. The same computation will look very different when executed by a human brain, with neurons firing in a specific cascade that gives rise to a three-dimensional activation pattern. Or maybe we are just dealing with a few beads being moved around in an abacus. Despite these differences in biological substrates, structural changes, and sheer computing power, all three are equally valid examples of computation. In fact, the notion of computation is so fundamental a concept that even the universe itself can be viewed as one giant computation — computers, in contrast, are just a handy gadget for carrying out computations.

Computational linguistics is an unfortunate name because it does not clearly disambiguate these two terms: are we talking about language and computers, or language and computation? For the purposes of this course, computational linguistics will always refer to "language and computation", while "language and computers" will be subsumed under the term *natural language processing* (NLP). NLP is about solving language-related tasks with computers, e.g. speech synthesis, machine translation, or even the basic search function in your text editor. Computational linguistics is about studying language as an instance of computation. And that will be our topic for the next 15 weeks.

At least it is better than the German term *Computerlinguistik*, which can only have the first meaning when interpreted compositionally.

More precisely, this course is concerned with the computational aspects of natural language and how we can analyze them from a formal perspective. While it may seem innocuous, this statement is actually very ambiguous and contentious. What exactly do we mean by *computational aspects*, why should those be interesting questions to ask, and what is the supposed advantage of this *formal perspective* over the proposals made by theoretical linguists?

1 Computational Linguistics: Why Should Anybody Care?

1.1 Practical Arguments

It seems fairly easy to make a case for the importance of studying computational aspects of language (putting aside for now what exactly we mean by that). Usually the first argument to be presented is that a world in which computers can successfully handle all kinds of language-related tasks is preferable to one where they cannot, and consequently it is imperative that we do whatever we can to get computers to this level of aptitude. And just like some understanding of physics had to be in place before engineers could bless mankind with the radio or the combustion engine, we cannot have successful NLP applications without a minimum understanding of language and the computational challenges it poses. Computational linguistics thus is a prerequisite for NLP.

This argument is too simplistic, though. One of the most shocking things about the applied sciences and engineering is how little genuine understanding one needs to construct a useful tool. To give but one example: relativity theory is not an integral part of calculating artillery ballistics. In many areas of life the allowed margin of error is large enough that shortcuts, hacks, and brute force methods will get the job done. For practical purposes it is also perfectly fine to make stipulations that fly in the face of scientific consensus but improve the final results. In the words of Noam Chomsky (Chomsky 1990:147):

> Throughout history, those who built bridges or designed airplanes often had to make explicit assumptions that went beyond the understanding of the basic sciences.

Similar things can be observed in NLP, where very shallow methods often yield surprisingly useful results. A word cloud, for instance, requires only the most basic statistics but can summarize texts in a very effective manner. Things might change in the future as users demand more and more accurate tools for increasingly difficult tasks, but at this stage there is still a large gap between "language as a computational problem" and "computers solving natural language tasks".

A more refined version of the practical applicability argument points out that while a theoretically informed perspective may not be of any practical use for now, it can lay the foundation for entirely new fields, tools, and businesses in the future. The poster child for this is of course number theory, which for the longest time was considered a purely theoretical and utterly useless subfield of mathematics but is now the central pillar on which all of modern cryptography rests. Whether you are encrypting files on your computer or talking to a server through a safe channel, it all builds on number theory. But we do not have to look at mathematics to see examples of theoretical research giving rise to important new developments, linguistics has several examples of its own. Formal language theory — which is an integral part of computer science and indispensable for the design of programming languages and file standards like XML, among other things — grew out of Chomsky's attempts to develop formal models of natural language syntax. Chomsky's writings about syntactic transformations also served as an inspiration for William Rounds' (Rounds 1970) work on tree transducers, which are now used in compiler design and machine translation. More recently, Aravind Joshi's Tree Adjoining Grammar (TAG) has even been used to model messenger RNA (Uemura et al. 1999; Matsui et al. 2005). Overall, then,

there is ample evidence that theoretical inquiry does benefit practical applications eventually and consequently we may hope that studying the computational aspects of language will benefit not only NLP, but also future areas that we cannot envision at this point.

1.2 Scientific Arguments

Utilitarian arguments may convince politicians, taxpayers, and engineers, but they hold no sway in the court of science. A linguist would shrug at the arguments above on a good day, and publicly denounce us as nimrods on a bad one. Fortunately, the scientific merit of computational linguistics is a lot more clear-cut: language is intrinsically a computational problem, so it should be studied as such.

Probably the most important development in 20th century linguistics is the *cognitive turn*, the shift from viewing language as an external system of rules and words to its reinterpretation as one of humans' many cognitive abilities. Language is not an abstract platonic object, it is done by humans, it is an algorithm that runs on the human brain (the *wetware*). That immediately raises the question how language is computed.

Semantics is noteworthy for being the one subfield of linguistics that is still very strongly aligned with the externalist view of language and does not think of its formalism as the high-level description of a specific part of human cognition. For a mentalist approach to natural language meaning see Pietroski (2014).

Linguists actually split this up into two subquestions:

Competence What is the specification of the computations?

Performance How is this specification implemented and used?

Competence questions are concerned with the rules of grammar and how they are encoded. Somewhat sloppily, one could describe competence as "language modulo resource restrictions like limited working memory and attention span". Of course nothing of this sort can be observed in nature — competence is an artificial abstraction of performance, which is about how the specification behaves when it is run on an actual machine, i.e. the human brain. The distinction between competence and performance also exists in computer science to some extent. For example, complexity theory studies the difficulty of problems of unbounded size, even though in practice problems are usually bounded, e.g. because a computer can only store so much information. Nonetheless complexity theory has produced results that are also relevant in practice, and competence questions in linguistics have similarly shed some light on performance.

Since competence cannot be directly observed, research into how language is computed usually operates in the realm of performance. Linguists approach this questions through the lens of neuroscientists and psychologists: how does the wetware behave when carrying out specific linguistic tasks, and can we design a procedure that mimics humans' behavior? For example, native speakers of English usually have no problem understanding English sentences, it is an incredibly fast and effortless process. But it does exhibit surprising quirks. When asked whether the sentence (1) is grammatical, they usually say yes.

(1) The player tossed a frisbee smiled.

However, the sentence is actually grammatical, it has the same structure as the minimally different (2).

(2) The player thrown a frisbee smiled.

For some reason the algorithm native speakers of English use has no problem with (2) but is completely thrown off by the exchange of a single word that serves exactly the same grammatical function. This is almost like a computer that can compute $1 + 2$ but not $2 + 1$. There is no obvious reason for this behavior, and it doesn't exactly look like good engineering (so much for intelligent design). Linguists have come up with various elaborate explanations of such phenomena over the years — some more successful than others — but they all share a variety of properties that necessarily limit their scope and thus the questions they can address.

Neuroscience

It is certainly interesting to see how the human brain works on the hardware level, but that does not tell us anything about the actual computations — just like studying a computer's hardware tells us little about what it is computing. If we hear the hard drive spin up we can make the reasonable assumption that some file is being accessed, but that is about all we can deduce with our own senses. Inquisitive minds that we are, we may then decide to connect a set of thermal diodes to various points of the hardware in order to detect whether that piece of hardware starts heating up, suggesting an increase of computational activity there. But that is still very inconclusive, for various reasons.

First of all, a higher load on the graphics chip might indicate that some kind of 3D graphics is being rendered, e.g. for a video game, but actually a lot of non-graphical tasks are outsourced from the processor to the graphics chip nowadays. And even if it were possible to tie each piece of hardware to a specific task, we still could not determine whether the computer is, say, sorting a list or searching through it, two very different tasks. More fine-grained distinctions are completely unthinkable, such as whether one of the two search algorithms below is being used, and if so, which one.

Graphics chips are also called Graphics Processing Units (GPUs). The usage of GPUs for non-graphical tasks is known as GPGPU: General Purpose Computing on GPUs.

```
def linear_search(search_list, searched_item):
    """Search trough list from left to right"""
    for i in range(len(search_list)):
        if searched_item == search_list[i]:
            print("Item found at index " + str(i))
            return i
    return False
```

```python
def binary_search(search_list, item, start=None, end=None):
    """
    Binary search algorithm with optional subrange of a list.

    This algorithm is more efficient but only works for sorted lists!

    Arguments:
    search_list -- list to be searched
    item        -- item we are looking for
    start       -- start index of search range
    end         -- end index of search range
    """
    # instantiate default values
    if start is None:
        start = 0

    if end is None:
        end = len(search_list) - 1

    # if a range is given by user, make sure it's sane
    if start < 0 or end > len(search_list) - 1:
        print("Indices outside valid range for given list")
        return False

    # if the range is ill-defined it cannot contain the item
    if end < start:
        print("Item not found")
        return False

    # find middle of the current range
    middle = start + (end - start)/2

    # Case 1: our item is to the left of the item at the midpoint
    if item < search_list[middle]:
        binary_search(search_list, item, start, middle - 1)
    # Case 2: our item is to the right of the item at the midpoint
    elif item > search_list[middle]:
        binary_search(search_list, item, middle + 1, end)
    # Case 3: we found our item, return its index
    elif item == search_list[middle]:
        print("Item found at index " + str(middle))
        return middle
```

The binary search algorithm is a lot more sophisticated than linear search, and it is also much faster on average. It does have the disadvantage that it only works for sorted lists, so unsorted lists need to be sorted first, and in cases where no sorting order can be defined, the algorithm does not work at all. So if we had two computers with exactly the same hardware, one using linear search and the other binary search, the latter would show vastly superior performance in general but would fail miserably in specific cases. The behavior of this computer might even look similar to humans' puzzling problems with certain grammatical sentences. Yet the behavior cannot be explained in terms of hardware, because we know for a fact that the two machines are exactly the same.

In the case of computers, we have the advantage that we already know how their architecture works because we designed it. So if that still isn't enough to deduce from the hardware what kind of computations a computer is carrying out, it seems rather unlikely that a similar process could derive from wetware the functioning of

the human brain. Granted, we can uncover limiting factors and some basic facts, just like a close analysis of a processor can reveal a maximum limit on its working memory (Level 1, 2, and 3 caches) and that all information is encoded in a categorical fashion via series of on and off states — the proverbial 0s and 1s. But all the probing, all the high-tech machinery tells us very little about the actual computations. That does not mean it is a fruitless endeavor, and eventually it will be possible to connect these two levels via some linking theory, but the crucial aspect is that we do need both levels, neuroscience alone cannot give us the full picture of language as a cognitive ability.

Psychology

Psychologists aren't interested with the physical instantiation of cognition, they are perfectly happy to treat the brain as a black box that produces a certain output given a certain input. Their goal is to develop models that replicate these input-output mappings. But more often than not these models are highly specific in what cognitive parameters they presuppose.

A very common assumption is that humans use *content addressable memory* (CAM). In contrast to computer's *random accessible memory* (RAM), information stored in CAM is not retrieved via an address that specifies its precise location in memory. Instead, the individual pieces of the information themselves act as a way of specifying the path to its location in memory. Imagine traversing a network where *doctor* takes you in one direction, and then *crab* in another so that you end up at a node that stores all the information about Doctor John A. Zoidberg from *Futurama*. Memory is also considered to be very limited in size and possibly stratified according to certain types of content. These properties are then coupled with certain models of memory activation and retrieval to explain specific aspects of human cognition, e.g. priming effects.

The downside of this approach is that is relies on assumptions that are hard if not even impossible to prove conclusively, that there's often many alternative solutions with no obvious way of choosing between them, and that complex accounts — by virtue of being complex — make it very hard to assess how much work is done by each individual part. The lack of conclusive proof is not too much of an issue, uncertainty is a given in almost every scientific endeavor. One cannot help but wonder, though, if the degree of uncertainty could be lowered. If there is a higher-level explanation that does not rely on quite as specific assumptions about working memory, that should be preferable. The same goes for the problem of many solutions: if you can cook up multiple models that use slightly different memory architectures but yield the same result if each one of them is also coupled with slightly different amount of memory, that indicates that the essential property may be more abstract, with the proposed psychological model being but one of many different ways of enforcing this property. And since there are usually many alternative solutions, it is very hard to show that certain parts of the machinery are indispensable to get the desired result.

Interim Summary and a Promise

All the limitations of neuroscience and psychology pointed out so far can be reduced to a lack of abstraction. Both fields operate at levels that specify a lot of information — like memory addressing and neural connections — whose relevance to language isn't apparent; in particular if one cares mostly about competence questions, as most theoretical linguists do. The great promise of computational linguistics, the one

advantage that sets it apart from neuroscience and psychology, is that it can completely abstract away from all extraneous detail and performance aspects. The methods used by computational linguists allow us to connect language and computation at the competence level. We can conclusively answer questions such as

- What is the weakest memory architecture that is sufficiently powerful to support a specific model of competence?
- What is the weakest competence model that is sufficiently powerful for a given empirical domain, e.g. local processes in phonology.
- Do alternative competence models describe the same class of computations?
- Is there an alternative representation of a given model that lowers memory requirements?
- How we carve up complex models into simple subparts?
- What kind of computational universals hold of language?

2 Marr's Three Levels and the Virtue of Abstractness

The observations made so far are far from new, they were already encompassed in *Marr's three levels of analysis* (Marr and Poggio 1976). Marr proposes that any aspect of cognition can be described on three levels of increasing abstraction:

physical the wetware or hardware instantiation; e.g. neural circuitry for vision or the machine code running on your computer

algorithmic what kind of computational steps does the system carry out and in which order, what are its data structures and how are they manipulated; basically the level of programming

computational what problem space does the system operate on, how are the solutions specified

Example 1.1 Set Intersection on Three Levels

Suppose you have two sets of objects, A and B, and you want to write a computer program that tells you which objects belong to both sets.

- On a computational level, that program is easily specified: it takes two sets as input and returns their intersection ($A \cap B$).
- On the algorithmic level, things get trickier. For instance, what kind of data structure do you want to use for the input (sets, lists?), and just how does one actually construct an object that's the intersection of two sets?
- On the physical level, finally, things are so complicated that it's basically impossible to tell what exactly is being computed by the machine. Voltages increase or decrease in various transistors spread over the CPU, memory and mainboard,

> and that's about all you can make out. Unless you already have a good idea of the higher levels and the computational process being carried out, it's pretty much hopeless to reverse engineer the program from the electrical signals.

Marr's levels of analysis highlight that one and the same object can be described in very different ways, and all three levels are worth studying. A computational specification can be implemented in various distinct ways on the algorithmic level, and algorithms can be realized in a myriad of physical ways — for instance, your laptop and your tablet use very different processor architectures (x86 and ARM, respectively), but a Python program will run just fine on either platform despite the differences in electric signals. And of course this hierarchy is continuous: Assembly code is closer to the physical level than C, which in turn is closer to it than Python. However, the more you are interested in stating succinct generalizations, the more you'll be drawn towards abstractness and hence the computational level at the top of the continuum. And this is exactly the level computational linguistics is aiming for.

3 Closing Remark: The Need for Formalization

The problem with abstraction is that one can no longer reason on a purely intuitive level. Since the objects are characterized by a few basic properties, it is important that these properties are described as precisely as possible. A minor misunderstanding may be enough to lead to completely contradictory conclusions. In the worst case, we may end up with an *inconsistent* theory, meaning that there is at least one property that is both true and false at the same time. This may be perfectly fine in a post-modern analysis of ableist slurs in humoristic epitaphs, but it has no place in a scientific theory. So abstraction necessarily requires a certain degree of rigor.

This shouldn't come as a big shock to you. Computer science can be very rigorous, in particular its theoretical subfields like complexity theory and formal language theory. The same is also true of linguistics: Generative syntax and phonology are abstract and involve a lot of technical machinery that seems arcane and intimidating to outsiders. The technical machinery is indispensable for each field's areas of scientific inquiry, and you all got the hang of it eventually after a few initial struggles.

The same is true of the machinery we will use in this course. It is more technical than linguistics, but only because we cannot make do with less. It does involve some math, but nothing that one couldn't pick up in a week. It is harder to read at the beginning, but I will try to keep notation to a minimum. You will get stuck sometimes, but that just means you have to think about the problem a couple more times until you get it. Nothing we do in here is really difficult, but it takes patience and dedication. Remember: the most important trait of a good researcher is to enjoy feeling stupid.

If you don't know it yet, I highly recommend that you read Martin Schwarz's excellent essay The importance of stupidity in scientific research. And if you do know it, I highly recommend that you read it again.