Unit 1

Computation and formalization

Without a doubt the most unfortunate fact about the field of 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.

Computational linguistics can certainly be about solving language-related tasks with computers, for example speech recognition and speech synthesis, spellchecking, dialog system, or stylistic analysis. But computational linguistics can be so much more. Once one understands how broad the notion of computation is, it becomes clear that the reach of computational linguistics can go far beyond computers. In doing so, it provides a deeper understanding of what language looks like from a computational perspective. This goal sounds rather vague, for sure, but it is not without merit.

The book you are holding in your hands (and, presumably, reading) is all about this broader notion of computational linguistics. The rest of this chapter tries to sharpen the profile of what we may call "computational computational linguistics", or simply *computational^2 linguistics*. We will also discuss why this approach is worth pursuing, lest you put away the book before even finishing its first chapter. The specific merits of computational^2 linguistics depends a lot on whether you are a natural language engineer, a theoretical linguist, or a cognitive scientist, but each group stands to gain something.

1 Computers vs computation

Here is a statement that the average layperson will find puzzling: While computers are the most common tool for carrying out computations nowadays, they are not what computation is about. Computation, in its barest form, is the principled manipulation of information, of transforming some input into some output in a precise, step-wise fashion. 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. But the computation is not tied to that specific electrical process, it can take many physical instantiations in this world.

The movie buffs among you will remember the 90s masterpiece *MacGyver: Lost Treasure of Atlantis*. In this spiritual successor to the *Indiana Jones* movies, the adventurer MacGyver discovers the secret of Atlantis: a giant, steam-powered computer that operates without electricity. Of course one should not lend too much credence to a movie where the protagonist cruises through a military camp with a rocket car that he built in 20 minutes. But the idea of a steam-powered computer is not nearly as outlandish. Any device that can assume multiple different states and switch between those states in a controlled fashion is capable of computation.

The idea that the act of computation is about transitioning from one internal configuration to another in a principled fashion is the central insight behind the *Turing machine*. A Turing machine consists of

1. a tape,
2. a read/write head,
3. and a so-called state register.

The read/write head can move to any position on the tape, read the symbol that is there, and possibly overwrite it with a new symbol. The state register is like a knob that can be in one of finitely many positions, e.g. 7 out of 10 on a volume dial. The purpose of the state register is to control the behavior of the Turing machine. Based on the symbol that is currently under the read/write head and the state of the register, the Turing machine consecutively executes three specific operations:

1. a *write action* (overwrite or do nothing), and
2. a *move action* (move left, move right, stay in place), and
3. a *state register change* (keep state, switch to different state).

So a basic instruction of a Turing machine may read like “if the symbol under the head is 1 and the state is A, overwrite 1 with 0, move left, and switch to state B”. A finite collection of such instructions is a program that can be run on a Turing machine to carry out specific computations.

Example 1.1 Copying with a Turing machine

The table below describes a small program for a Turing machine.

state	*tape symbol*	*write action*	*move action*	*new state*
A	0	none	none	F
A	1	write(0)	$\Leftarrow$	B
B	0	none	$\Leftarrow$	C
B	1	none	$\Leftarrow$	B
C	0	write(1)	$\Rightarrow$	D
C	1	none	$\Leftarrow$	C
D	0	none	$\Rightarrow$	E
D	1	none	$\Rightarrow$	D
E	0	write(1)	$\Leftarrow$	A
E	1	none	$\Rightarrow$	E

This looks fairly cryptic, so let’s tease apart what’s going on here.

Each individual instruction is easy enough to decypher. The machine has 6 different states: A, B, C, D, E, and F. Only two kinds of symbols are used on the tape, 0 and 1. A command like write(1) means that the machine writes a 1 onto the tape, whereas the arrows $\Leftarrow$ and $\Rightarrow$ specify that the machine moves one symbol to the left or one symbol to the right after the write action is finished. So line 5, for example, tells us that if the machine is in state C and has a 0 under its read/write head, it writes a 1, moves to the next symbol to the right, and switches into state D.

That is all nice and dandy, but what is it the program does? So far this feels like a badly written instruction manual where each individual sentence makes sense but you can't figure out how they fit together (very much **un**like this book, I hope). Let us boost our understanding of the program by working through a concrete example.

Suppose that the machine starts in the following configuration: The tape consists mostly of 0s, except for two adjacent 1s, and the read-write head is positioned on the right 1, with the state register in state A. This is visualized below.

$$0\quad 0\quad 0\quad 0\quad 1\quad \boxed{\begin{matrix}1\\ \text{A}\end{matrix}}\quad 0\quad 0$$

This configuration is matched by the second line of the instruction table. Hence the machine overwrites the current symbol with a 0, moves to the left and switches into state B. Here is the resulting configuration, with the changed symbol highlighted in **boldface**.

$$0\quad 0\quad 0\quad 0\quad \boxed{\begin{matrix}1\\ \text{B}\end{matrix}}\quad \mathbf{0}\quad 0\quad 0$$

Since the read/write head is now over a 1 while the machine is in state B, the instruction on line 4 is triggered: the machine keeps the current symbol as is, moves to the left, and keeps the register in state B.

$$0\quad 0\quad 0\quad \boxed{\begin{matrix}0\\ \text{B}\end{matrix}}\quad 1\quad 0\quad 0\quad 0$$

This in turn triggers the third instruction, which tells the machine not to perform any write action, move one symbol to the left, and switche the register to state C.

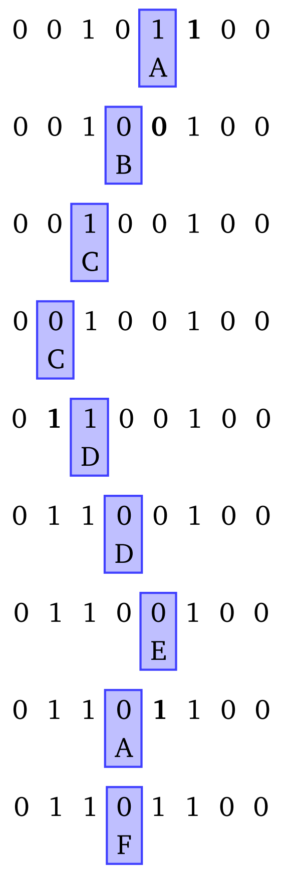
$$0\quad 0\quad \boxed{\begin{matrix}0\\ \text{C}\end{matrix}}\quad 0\quad 1\quad 0\quad 0\quad 0$$

The rest of the computation then proceeds as depicted below:

$$0\quad 0\quad \mathbf{1}\quad \boxed{\begin{matrix}0\\ \text{D}\end{matrix}}\quad 1\quad 0\quad 0\quad 0$$

$$0\quad 0\quad 1\quad 0\quad \boxed{\begin{matrix}1\\ \text{E}\end{matrix}}\quad 0\quad 0\quad 0$$

$$0\quad 0\quad 1\quad 0\quad 1\quad \boxed{\begin{matrix}0\\ \text{E}\end{matrix}}\quad 0\quad 0$$



The F state is special because it does not trigger any new instructions, so the machine halts once it reaches this state. Looking at the result of the individual steps, we can now see that the instructions at the beginning of the example program the Turing machine so that it copies sequences of 1s. If the tape had contained 11111 instead of 11, the final output would have contained two instances of 11111, assuming a sufficiently long tape.

The example above shows that a Turing machine can compute copies. As we will see in later chapters, copying is actually a very complex task that plays an important role in natural languages. Despite the complexity of copying, it can be understood in the very general terms of a Turing machine as simply a sequence of configuration changes: what does the tape look like, where are we on the tape, and what state is the machine in?

The generality of the Turing machine is what enables a broader understanding of computation that does not care about the actual hardware. A Turing machine is not a concrete object, it's not a tiny box with some tape and a state dial that you can order from Amazon. Instead, it is an abstract model of what it means to carry out a computation, and there are many different ways a Turing machine can be instantiated in the real world. This is particularly noteworthy because Turing machines act as a kind of standard model for computation. It is known that if there is no limit on how much tape is available, any problem that can be solved computationally can be solved by a Turing machine. So all kinds of computation can be regarded as a specific program that runs on a Turing machine. But this also means that any machine, system, or construct that provides the equivalent of a tape and a controllable read-write head is a computing device.

For example, a Turing machine can even take the form of a very selfish drinking game: Gather 4 friends of yours and 6 shot glasses — I am boldly assuming that you have enough of both. Put the shot glasses in a line and fill the rightmost two with a beverage of your choice. Then give each one of your friends a card with instructions they have to follow. For the sake of exposition, let's assume that your friends are called Bill, Cathy, Damian, and Edith. Their respective cards read as follows:

- **Bill**
  If the shotglass in front of you is empty, get out of line and put Cathy in front of the glass to the left. Otherwise, leave the glass alone (sorry!), and move to the glass to the left.

- **Cathy**
  If the shotglass in front of you is empty, fill it up, get out of line, and put Damian in front of the glass to the right. Otherwise, leave the glass alone (sorry!), and move to the glass to the left.

- **Damian**
  If the shotglass in front of you is empty, get out of line and put Edith in front of the glass to the right. Otherwise, move to the glass to the right.

- **Edith**
  If the shotglass in front of you is empty, fill it up, get out of line, and put me in front of the glass to the left. Otherwise, move to the glass to the right.

The instructions for yourself are slightly more fun. If the glass in front of you is full, drink it all, get out of line, and put Bill in front of the glass to the left. If the glass is empty, the game is over.

I suppose you can already tell what is going on here. When you play this game, it will proceed exactly like the Turing machine from our copying example (it is a selfish drinking game because you are the only one who gets to drink, i.e. rewrite a 1 as a 0). Even though you and your friends are separate individuals and do not form any kind of unified object, the combination of all of you and the shotglasses constitute a Turing machine.

We can make all kinds of changes to this setup without losing the connection to Turing machines. For example, we may use bowls instead of glasses, and fill them with M&Ms instead of some beverage. And maybe we do not actually put the bowls in a line but instead assume that they all differ in size like a set of baking bowls, so that "left" means "the next smaller one" and "right" means "the next bigger one". Perhaps we could even replace your friends with a very well-trained dog. Clearly a dog and a human are two very different things, but it changes nothing about the computation that is being carried out. No matter how we set things up, the same input will always be transformed into the same output.

Silly as these examples may be, the central point stands: changes to the physical make-up of the device that carries out the computations does not entail changes to the computation itself. The notion of computation operates at a higher level of abstraction, and that is what gives it such a unifying power. We can take computational concepts and apply them to systems that do not at all look like the computers we are familiar with: the human brain, the biological mechanisms of gene expression, even the universe itself. Despite the differences in physical substrates, structural changes, and sheer computing power, they are all equally valid examples of computing devices and we can discover interesting things about them by adopting this perspective.

Hopefully you can now appreciate why it is unfortunate that the term *computational linguistics* does not clearly disambiguate between computation and computers. While the latter emphasizes engineering concerns, the former strives for a more abstract perspective that applies to computers as well as humans. Seeing how humans are the only known computing device with excellent command of natural language, we would do well to study language at a level that is compatible with those devices and learn from them as much as possible. To clearly differentiate the two notions of computational linguistics, I will use *natural language processing* (NLP) in this book to refer to those aspects of computational linguistics that are solely concerned with computers. 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.

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

This is all still very vague, and to some extent things will only get clearer once we move on from our high-level vantage point and start looking at concrete issues. Still, it is always nice to have a rough idea of the road ahead and why it is road worth taking. So let us next consider why it makes sense to study language from a computational perspective and how one might go about this.

2 Computational linguistics: The what, how, and why

2.1 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). The argument starts with the plausible assumption that a world in which computers can successfully handle all kinds of language-related tasks is preferable to one where they cannot. This would create a second industrial revolution that boldly pushes automation into language-heavy domains: customer service and speech-driven user interfaces, language and writing instruction, knowledge aggregation, and much more. Admittedly there is also the risk of mass surveillance, mass unemployment, and the social upheavals that tend to follow both, but there is reason to believe that those would just be short-term growing pains on the way to a more prosperous future. If this is correct, then 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 intuitively pleasing and, in my humble estimate, ultimately correct. In the form presented above, though, it is too simplistic and easy prey to somebody playing devil's advocate: One of the most shocking aspects of 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 permitted margin of error is large enough that shortcuts, hacks, and brute force methods will get the job done just fine. 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, the founding father of modern linguistics (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. Many of its tools and techniques ignore well-established linguistic knowledge for the sake of simplicity and efficiency. Nonetheless these tools do surprisingly well and often outperform competing models that draw from what linguists have learned about language. The computational linguist Frederick Jelinek summarized this state of affairs very succinctly during his time at IBM in the 80s: “Every time I fire a linguist, the performance of the speech recognizer goes up”. A more recent example is a study carried out by computational linguists at Stony Brook University. This study evaluated the ability of various models to predict the success of novels. Naively, one would expect that a novel’s success is largely dependent on its writing style, narrative structure, and subject matter. Yet one of the best models completely ignored those aspects and focused exclusively on word frequencies. It seems, then, that there is still a large gap between “language as a computational problem” and “computers solving natural language tasks”, and that the latter is doing just fine without the former.

With the rapid rise of machine learning methods in recent years, one might even expect the gap to widen over the next few decades. General purpose machine learning strategies frequently result in better models than those produced by language-specific techniques in NLP. If this trend continues to its logical extreme, NLP may be completely absorbed into the field of machine learning. In this case, any language-specific insights would be even harder to incorporate given the domain-agnostic nature of the machine learning techniques. If “computers solving natural language tasks” simply turns into “computers solving tasks”, then “language as a computational problem” is too specific an enterprise to make worthwhile contributions.

There is more than a grain of truth in this counterargument, but at the same it is too narrow and overly reductionist. Admittedly some areas of application do not profit much from linguistic insight. For example, a government may want to improve the mental well-being of its citizens by screening tweets for signs of mental illness and offering preventive care to users that seem to be at risk. This task would not profit much from a deeper understanding of natural language and its computational intricacies. For one thing, tweets are often short enough that their intended meaning can be inferred just from a few keywords and hashtags. Second, the correlation between mental health and language use is fairly loose and not all that well understood. In the absence of a robust theory, the best approach is trial-and-error: identify a few reasonable indicators of mental illness, build several probabilistic models that pay attention to these indicators, and go with the model that performs best. This route is guaranteed to produce some kind of model within a short amount of time, and the model is very likely to perform better than any model that builds on the most recent scientific research, simply because the problem is too limited for the theoretical underpinning to pay off in a significant manner. The more specific and restricted the problem, the less need there is for deep understanding (though some exceptions prove the rule).

But not all problems are simple or very restricted. Language technology is still very much about going for the low-hanging fruits while many interesting problems are considered too hard. No computer currently understands instructions like the following: “Go to my pictures folder and rename the files. I want each filename to start with the date and then list all the names of the people on the picture.” A simple request, it seems, but it is actually bursting with linguistic complexity:

1. “my pictures folder”: Which folder is that? Who is speaking here? What if there’s multiple folders with pictures? Can we make a reasonable guess about which folder the user has in mind?

2. “rename the files”: What files? Presumably the user only wants to rename the files in the pictures folder, not all the files on the hard drive. But this piece of information needs to be inferred, it is not given explicitly.

3. “I want”: So the computer should make sure that the files end up looking like what the user has in mind, even though the user isn’t explicitly issuing a command.

4. “each filename”: Actually the user wants only image files like JPEGs or PNGs to be renamed, whereas the doc-file that they randomly dropped in there should be left alone.

5. “start with the date”: Dates can be formatted in various ways. Can we make a reasonable guess about what format the user wants? If not, how can we ask for clarification?

6. “then list”: Is this still part of the description of the filename or a new command to the computer? That is to say, should the filename include the names of people, or should the computer list their names for the user after each successful file renaming?

7. “list all the names of the people on the picture”: What is the intended interpretation here? People that appear on the picture should have their name listed in the filename? Each person should have their name listed on the picture? List every name that belongs to a group of people and appears on the picture?

8. “the picture”: Which picture? Probably the one that is being renamed, but again this restriction is left implicit.

In our amazement that computers seem to accomplish super-human feats such as translating a website in a split-second, we tend to forget that most of the language-related tasks that even 5-year olds solve with ease still pose insurmountable challenges for computers. Once one broadens the horizon from what NLP is currently working on to what NLP could be working on, it becomes abundantly clear that a deeper, computationally sophisticated understanding of language is sorely needed.

In a certain sense this holds even if one considers only those areas where NLP has been successful. Machine translation, for example, has made tremendous leaps in the last 10 years and can produce remarkable results now — if one picks the right source and target language. Translating a text from English to Spanish will work much better than translating the very same text from Suaheli to Inuktitut. This is because modern NLP-techniques are tremendously data-hungry. In order for a probabilistic model to perform well, it has to be fed millions of data points. Only a few languages have enough digitized text to meet the data hunger of these models, all other languages are considered *resource-poor*. Such resource poor languages are second-class citizens in the realm of NLP. Note that even languages like Swedish and Afrikaans, which are spoken in very affluent countries, are resource-poor for NLP-purposes. And the problem isn't limited to languages, every dialect that deviates to a significant extent from the standard is resource-poor. So Scottish English, Bavarian German, and African-American Vernacular English are all ill-served by current NLP solutions compared to their standard counterparts. As language technology becomes increasingly important, there is a real danger that current NLP techniques end up indirectly discriminating against (linguistic) minorities.

That this does not need to be the case is proven thousands of times each days as children in all kinds of linguistic communities effortlessly acquire their native language from very limited input. In fact, children are amazing language learners and frequently generalize in ways that go far beyond what the data provides. This is what linguists call the *poverty of stimulus*: even though the linguistic input children get is very limited, they infer a lot more from it than would be possible just via raw statistics. How exactly children manage to do this is still very much an open issue, in particular on the computational level. But the more our knowledge advances on this front, the less data future models will need, thereby shrinking the gap between resource-poor and resource-rich languages.

This shows, then, that there is at least in principle great potential for improving NLP via deep computational theories of natural language. The siren song of the possible is certainly hard to resist, but it cannot compete with the normative power of historical truth. And this truth is that there is a long tradition of theoretical insights improving NLP. Consider the problem of word structure, which linguists call *morphology*. A native speaker of English knows that *liked* is the combination of the verb *like* and a past tense marker, and in the other direction he or she also knows that the addition of a past tense marker turns *walk* into *walked*, *run* into *ran*, and *go* into *went*. Linguists have collected a detailed inventory of morphological processes, and computational linguists have developed formal systems such as two-level morphology to model these processes. These two steps did not take place independently of each other. The crucial contribution of linguists was to point out that the range of possible morphological processes is very limited. Once computational linguists saw what is (im)possible in morphology, they realized that almost all the processes are very simple and can be captured with very restricted tools that are still very flexible and scalable, which is exactly what one wants for practical purposes. At the same time, a few processes in morphology that involve copying did not fit into this class, prompting linguists to look even more closely at them. This way theoretical and computational linguists feed each other's research, with NLP as the happy consumer of the final results.

Admittedly not all examples work out as nicely as the one above. In many cases the theoretical insights tend to be grounded in pure math rather than linguistic fieldwork or analysis. But that is not to say that linguistics has contributed only very little of use for NLP. Formal language theory — which is still an integral part of computational linguistics but also indispensable for the design of programming languages and file standards like XML — grew out of Noam Chomsky's linguistic theories of natural language. Chomsky's *Transformational grammar* also served as an important inspiration for William Rounds' work on tree transducers, and those are now used in machine translation (outside of computational linguistics, they are also heavily used in compiler design, which one may regard as machine translation for programming languages). The computational linguist Aravind Joshi relied on key insights about the nature of language when he developed Tree Adjoining Grammars (TAG), which are still a highly influential and have even been used to model biological structures such as messenger RNA. The same can be said about the formalism of *Combinatory Categorial Grammar*, which is equally driven by engineering concerns and pure linguistic theory. Overall, then, there is ample evidence that theoretical inquiry does benefit practical applications eventually, although the connections are often indirect.

In sum, NLP is no different from other fields in that it, too, can profit from more theoretically minded endeavors. This holds whether the theory is grounded in mathematics, computer science, or linguistics, each one of them has already made valuable contributions and will continue to do so. The value of theory is not constant across all applications. Hard problems tend to benefit more than easy ones. But it is also important that the problem is well-defined — if it isn't even clear what exactly the problem is, theory will be of very limited use. This is why Twitter-based mental health assessment requires a healthy dose of opportunistic trial-and-error with all available tools, whereas computational morphology, for instance, benefits a lot from a good theoretical foundation. In my humble opinion, though, the future of NLP lies in dealing with hard problems that are ripe with language-specific issues, rather than the simple tasks that can be done equally well with a general-purpose machine learning algorithms.

2.2 Scientific arguments

The previous section has defended the approach of computational^2 linguistics in terms of its utility for NLP. Utilitarian arguments may convince politicians, taxpayers, and engineers, but they hold no sway in the court of science. A linguist might 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^2 linguistics is a lot more clear-cut than the utilitarian argument: language is intrinsically a computational problem, so it should be studied as such.

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).

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 that exists independent of reality. It is done by humans, it is an algorithm that runs on the hardware of the human brain. Since the brain is pretty moist and mushy, this is often called the *wetware*. But if language is a computation carried out by wetware, this immediately raises the question how language is computed.

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 the resource restrictions on 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 wetware. 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 neuroscience and psychology: 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 grammatically correct, native speakers usually say no.

(1) The player tossed a frisbee smiled.

However, the sentence is actually well-formed, 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. Where the reach of these approaches ends, the realm of computational^2 linguistics begins.

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. Actually, though, a lot of non-graphical tasks are outsourced from the processor to the graphics chip nowadays. In fact, even sound chips can be co-opted this way. When the first-person shooter *Doom* was ported to the Atari Jaguar, buyers and reviewers alike lamented the lack of any in-game music. The reason for that is that *Doom* was originally too demanding for the console, so the developers had to make the chip that would usually handle the music function as a co-processor for other calculations. But that meant, of course,

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.

that the chip could no longer handle the music, and there was no other chip this task could be outsourced to.

The human brain is not as content-agnostic as a modern computer, specific tasks seem to be tied to specific areas of the brain. But the connection is very loose. Suppose task X is localized in brain area A. Then a computationally similar task Y may or may not be localized to area A. In the other direction, not every task Z that is localized to A is necessarily computationally comparable to X. There is no strict implication in either direction, the locus of computation reveals very little about the type of computation.

Beyond the qualitative computational uncertainty, there is a severe granularity mismatch. Even a simple task like searching through a list can be accomplished in various ways, each with their own advantages and disadvantages. The two most fundamental strategies are linear search and binary search.

Example 1.2 Linear search and binary search

Suppose you ask somebody to memorize the list A, C, D, F, G, X. Then you ask them if the list contained the symbol G. How could they answer your question? I honestly don't know how a human does it, nor does anybody else. But there are at least two safe strategies: *linear search* and *binary search*.

In linear search, we simply scan the list from left to right and check one symbol after the other. If we find the symbol we are looking for, we reply that it is in the list. If we make it through the whole list without ever seeing the requested symbol, we reply with a no. The table below shows what happens at each step of the computation.

Step	Symbol	Remaining list
0		A, C, D, F, G, X
1	A	C, D, F, G, X
2	C	D, F, G, X
3	D	F, G, X
4	F	G, X
5	G	done

As you can see, linear search finds the requested item in 5 steps.

Binary search is a more efficient search strategy that exploits the fact that the symbols in the list are not in a random shuffle but ordered according to the Latin alphabet. Rather starting at the beginning of the list, binary search goes straight to the middle. If the middle point is the symbol we are looking for, we stop there. If the symbol there is alphabetically **before** the searched symbol, then we throw away the **left** half of the list and perform binary search on the remainder. And if the symbol at the middle point is alphabetically **after** the search symbol, then we throw away the **right** half of the list and perform binary search on the remainder.

Step	Symbol	Remaining list
0		A, C, D, F, G, X
1	D	F, G, X
2	G	done

Binary search can be much faster than linear search. The further to the right in a list an item occurs, the longer linear search will take compared to binary search.

On average, binary search is much faster than linear search for sorted lists. 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. 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 memory 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.

Thus it isn't exactly surprising that no neuroscientist on this planet knows whether at least some computations carried out by the wetware of the human brain involve anything resembling linear search or binary search. But this is one of the simplest distinctions that can be made when it comes to search. Many of the concepts discussed in this book are much more abstract, which makes it even harder to bridge the gap between the hardware/wetware-level and computation. It might be possible to do so if one approaches the issue from both directions. But a unidirectional process that seeks to build a computational theory of the human mind, or even just language, purely in terms of how the wetware operates is very unlikely to succeed due to the enormous mismatch in granularity and the concomitant problem of underspecification. At least in the domain of language, computational^2 linguistics can act as a linking theory by identifying computational core mechanisms of language that may be reflected in the wetware.

Psychology

In contrast to neuroscientists, psychologists aren't interested in the physical instantiation of cognition. They are perfectly happy to treat the brain as a black box that produces a certain output (= reaction) given a certain input (= stimulus). Their goal is to develop models that replicate these input-output mappings. But again there are certain shortcomings that comutational^2 linguistics can help address.

Very often psycholinguistic models are highly specific in what cognitive parameters they presuppose. A common assumption is that humans use *content addressable memory* (CAM). CAM operates very differently from a computer's *random accessible memory* (RAM). RAM is similar to a very long street where one can instantaneously travel to any house as long as one knows its house number. So if a computer has stored a list in memory at a given address, all one needs is this address to immediately retrieve the list from memory. Information stored in CAM, on the other hand, 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 large, maze-like network of intersecting roads, where every road has a descriptive title such as *doctor*, *crab*, and *cartoon*. At the point where those three roads intersect, you will find the house of Doctor John A. Zoidberg, the alien crab doctor from the cartoon show *Futurama*. There is no longer a need for arbitrary addresses because the properties of any given thing carve out the path for finding this thing. Psycholinguists couple CAM with certain models of memory activation and retrieval to explain specific aspects of human cognition, e.g. that memory retrieval for item X takes less time if a content-related item Y already had to be retrieved immediately before (this is known as *priming*).

The psycholinguistic approach has multiple downsides. First, it relies on assumptions that are hard if not even impossible to prove conclusively. Second, there's often many alternative solutions with no obvious way of choosing between them. Finally, psycholinguistic accounts tend to be complex combinations of multiple assumptions and postulations, which makes it very hard to assess how much work is done by each piece of the theory.

The lack of conclusive proof for psycholinguistic assumptions 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 too many solutions: if it is possible to change the value of parameter P in a model and still get the same result as long as one also changes parameter Q, what might really matter is not P and Q as such but the more abstract dependency between them. The different psychological models would then be but a few of many, possibly infinitely many different ways of encoding this dependency.

Computational^2 linguistics can be of help here by characterizing the abstract dependency in more neutral terms and proving in formal terms how much a model can be altered without breaking this crucial dependency. This also makes it easier to identify which parts of a model are truly indispensable. For psycholinguists, then, the central promise of computational linguistics^2 is a theory of model classes that reduces indeterminacy and disentangles the essential from the dispensable.

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. morphology.

- Do alternative competence models describe the same class of computations?
- Is there an alternative representation of a given model that lowers memory requirements?
- How can we carve up complex models into simple subparts?
- What kind of computational universals hold of language? That is to say, what kind of computational properties hold of every natural language?

3 Marr's three levels of analysis, and the virtue of abstractness

The final observations regarding abstractness are far from new, they were already encompassed in *Marr's three levels of analysis*. 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 1.3 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, do you want to store the sets as lists or something else, and just how does one actually construct an object that is the intersection of two sets?
- On the physical level, finally, things are so complicated that it is nigh 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. In the human brain, neurons are firing in intricate patterns that somehow give rise to the desired output. Unless you already have a good idea of the higher levels and the computational process being carried out, it is pretty much hopeless to reverse engineer the program from the physical evidence.

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 these differences. And of course this hierarchy is continuous: Assembly code is closer to the physical level than the code of the programming language C, which in turn is closer to the hardware than Python. However, the more you are interested in stating succinct generalizations, the more you will be drawn towards abstractness and hence the computational level at the top of the continuum. And this is exactly the level computational^2 linguistics is aiming for.

4 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, which means 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 transphobic slurs in humoristic epitaphs, but it has no place in a scientific theory. So abstraction necessarily requires a certain degree of care and 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 truly difficult, but it takes patience and dedication. If some of the material covered in this book seems overwhelming to you and makes you doubt your own intellect, remember that the most important ingredient for efficient learning is the ability 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.

Central concepts

- Computation $\neq$ computers
- Turing machines
- Linear VS binary search
- Marr's levels of description
- Virtue of abstraction

References and recommended readings

- Manning on machine learning and NLP
- SIGMORPHON competition with neural networks
- Heinz on SL versus LSTM
- Sean Fulop quote
- Joshi on TAG
- Steedman on CCG
- Rounds on tree transducers
- Chomsky 57

Code

```python
def linear_search(search_list, item):
    """Search search_list from left to right for item.

    Parameters
    ----------
    search_list : list
        list to be searched
    item : any
        item we are looking for

    Returns
    -------
    int or False

    Examples
    --------
    >>> linear_search([0,1,7,9], 7)
    2

    >>> linear_search([0,1,7,9], 8)
    False
    """
    for i in range(len(search_list)):
        if item == search_list[i]:
            return i
    return False
```

```python
    >>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
    >>> linear_search(test_list, 'e')
    3
    >>> linear_search(test_list, 'g')
    False
```

```python
def binary_search(search_list, item):
    """Perform binary search for item in search_list.

    This algorithm is more efficient than linear search,
    but only works for sorted lists!

    Parameters
    ----------
    search_list : list
        list to be searched
    item : any
        item we are looking for

    Returns
    -------
    int or False

    Examples
    --------
    >>> binary_search([0,1,7,9], 7)
    2

    >>> binary_search([0,1,7,9], 8)
    False
    """
    start = 0
    end = len(search_list) - 1

    while start <= end:
        # pick middle element of list
        middle = int(start + (end - start)/2)

        # Case 1: our item is to the left of the item at the midpoint,
        #         limit next search to left half
        if item < search_list[middle]:
            end = middle - 1
        # Case 2: our item is to the right of the item at the midpoint,
        #         limit next search to right half
        elif item > search_list[middle]:
            start = middle + 1
        # Case 3: we found our item, return its index
        elif item == search_list[middle]:
            return middle
    return False
```

```python
    >>> test_list = ['a', 'c', 'd', 'e', 'b', 'f']
    >>> binary_search(sorted(test_list), 'e')
    4
    >>> binary_search(sorted(test_list), 'g')
    False
    >>> binary_search(test_list, 'c')
    1
    >>> binary_search(test_list, 'e')
    False
```