Unit 0

Computation(al) linguistics

Perhaps 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.

When the layperson hears the term *computational*, they immediately think of doing things with computers. The actual hardware might be a laptop, a phone, or the NSA's giant server farm in Utah, but in the end it always boils down to some kind of electronic device that was deliberately designed and engineered by humans. In the case of language, there certainly is no shortage of tasks we want these devices to handle: word completion for text messages, speech recognition and speech synthesis for our GPS, detecting spam mails, translating websites on the fly, and much more. Thanks to decades of research in computational linguistics, computers now do surprisingly well at these tasks. Sure, the existing solutions are far from perfect and occasionally make bewildering mistakes unlike anything even a highly inebriated human would produce. The more linguistically complex the task, the worse computers tend to fare. But the technology has proven good enough to become an essential part of our daily lives, and it keeps improving at a rapid rate. Computational linguistics as the study and design of language technology has been a resounding success.

But language technology isn't all there is to computational linguistics. Limiting the notion of computation to "doing things with computers" is doing it a great disservice. The scope of computational linguistics goes far beyond computers. It encompasses any object that is capable of computing, be it computers, the human brain, or abstract computing devices like the Turing machine, which isn't tied to a specific physical medium. With this general notion of computing, the goal is no longer to solve language-related tasks. No, **language is the task**: what does language look like from a computational perspective?

The book you are holding in your hands (and, presumably, reading) is all about this broader notion of computational linguistics. For lack of a better term, I call this *computation linguistics*. Its focus on understanding language makes computation linguistics a subfield of linguistics, even if its methods borrow heavily from theoretical computer science and mathematics (formal language theory, learnability, algebra, lattices, parsing theory, and so on). The remainder of this chapter sharpens the profile of computation linguistics and how it differs from other varieties of computational linguistics. I will also discuss why this approach is worth pursuing. The specific merits of computation linguistics depends a lot on whether you are a natural language engineer, a theoretical linguist, or a cognitive scientist. But rest assured, each group stands to gain something.

One more remark: given its subject matter, this chapter is necessarily very meta-theoretic. If you would rather get right into the thicket of things, feel free to jump ahead to the next few chapters and come back later. A general sales pitch for computation linguistics is all nice and dandy, but the proof of the pudding is in the eating.

1 Computers vs computation

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 1990s masterpiece *MacGyver: Lost Treasure of Atlantis*. In this spiritual successor to the *Indiana Jones* movies, MacGyver discovers the secret of Atlantis: a giant, steam-powered computer that operates without electricity. MacGyver is incredulous — how could they have a computer without electricity? But the idea of a steam-powered computer is actually far from 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*. It was first defined by Alan Turing in 1936 in his seminal paper *On Computable Numbers, with an Application to the Entscheidungsproblem*. (To the select few readers whose German is a little rusty: *Entscheidungsproblem* means *decision problem*). If you understand how the Turing machine works, then you know why computation isn't tied to a specific hardware instantiation, be it electricity, steam, or the human brain. So let us take a closer look at the Turing machine.

Alan Turing (1912–1954)

Alan Turing was an English mathematician and polymath. His gamut of accomplishments have made him one of the most influential researchers of the 20th century. He laid large parts of the foundation of modern computer science, in particular artificial intelligence and the theory of computation. During World War 2, he took a leading role in cracking the Enigma, an encryption device used by the Nazis. It has been estimated that this breakthrough saved millions of lives. Turing was also an excellent long-distance runner, and was even a contender for a spot on the British Olympic team in 1948.

Turing's numerous accomplishments were not fully acknowledged during his lifetime, and as a homosexual he even faced criminal prosecution. In 1952, he was sentenced to undergo chemical castration treatment due to acts of "gross indecency" (i.e. homosexual relations). Two years later he died of cyanide poisoning, presumably a suicide.

Turing's dead body was found next to a half-eaten apple. For this reason, it has

been rumored that the original, rainbow-colored Apple logo is a tribute to Alan Turing and his tragic death due to the British prosecution of homosexuals. However, Apple co-founder Steve Wozniak has called this an urban legend.

The Turing machine is now widely considered to establish a firm upper bound on what can be computed in a mechanical fashion. This is astounding considering that a Turing machine consists of only three components:

1. a tape that acts as a rewritable and infinitely extendable data storage, and
2. a read/write head that can modify the tape, and
3. a state register that controls the behavior of the machine.

The tape is a linear arrangement of cells, and each cell stores at most one symbol. The read/write head can move to any cell on the tape, read the symbol in that cell, 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. 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).

A simple instruction of a Turing machine may read "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 0.1 Copying with a Turing machine

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

current state	*tape symbol*	*write action*	*move action*	*new state*
A	0	none	none	F
A	1	write(0)	⇐	B
B	0	none	⇐	C
B	1	none	⇐	B
C	0	write(1)	⇒	D
C	1	none	⇐	C
D	0	none	⇒	E
D	1	none	⇒	D
E	0	write(1)	⇐	A
E	1	none	⇒	E

This looks fairly cryptic, so let us tease apart what is going on here.

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 fills the current cell on the tape with a 1, whereas the arrows ⇐ and ⇒ specify that the machine moves one cell to the left or one cell 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's terrific, but what is that good for? What does the program do? 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 rightmost 1, with the state register in state A. This is visualized below.

0	0	0	0	1	1	0	0
					A		

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	0	0	0	1	**0**	0	0
				B			

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. No change is made to the tape.

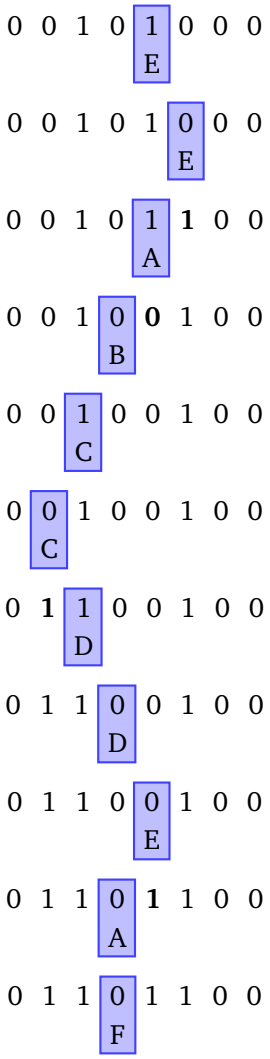
0	0	0	0	1	0	0	0
			B				

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

0	0	0	0	1	0	0	0
		C					

The rest of the computation then proceeds as depicted below, which changes to the tape once again highlighted in boldface:

0	0	**1**	0	1	0	0	0
			D				



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 final outcome of the individual steps, we can now make sense of the instructions at the beginning of this example. Put together, they 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. That's longer than the tape in our example, but remember that the tape of a Turing machine can always be extended as necessary.

The example above shows that a Turing machine can create copies of an input. 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. 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, the combination of you, your friends, and the shotglasses constitutes a Turing machine. The instructions you give each person are parts of the program that runs on this 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 that is randomly distributed around your ktichen. We then reinterpret "left" so that it means

"the largest bowl that is smaller than the current bowl". And "right" now means "the smallest bowl that is bigger than the current bowl". Even though we no longer have the bowls in a line, we can still move "left" and "right" based on the relative size of the bowls. 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. Shot glasses, bowls, M&Ms, humans, dogs, it does not matter, we always end up copying the input.

Silly as these examples may be, the central point stands: making changes to the physical make-up of the device that carries out the computation does not entail making 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 speed, 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. Remember, humans are the only known computing device with perfect command of natural language. In the spirit of learning from nature, we would do well to study language at a level that is compatible with these 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. Computation linguistics is about studying language as an instance of computation. Both NLP and computation linguistics belong to the larger field of computational linguistics, but they differ in their goals and in their methods.

While the terminology might be less vague now, the underlying concepts are still very abstract and intangible. To some extent things will only get clearer once we move on from the high-level perspective in this chapter and start looking at concrete issues. Still, if the road ahead is shrouded in mystery, it would at least be nice to why it is a road worth travelling. So let us next consider why one would want to study language from a computational perspective.

2 Why computation linguistics?

The focus on computation linguistics might seem peculiar to you. NLP has an easy sales pitch: "Make the world a better place while earning a six digit salary." Computation linguistics, on the other hand, has less tangible goals. Even if you are a heavily theory-minded researcher, it might not be immediately clear what computation linguistics has to offer. But there are good arguments for computation linguistics, and they range from real-world applications to scientific insights.

2.1 Practical arguments

The standard argument, and its standard counterargument

Let us first look at an argument that seems plausible, but ends up running into several issues. The argument starts with the reasonable 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 let's assume 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. Computation 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. Let's take a careful look at the counterargument such a person might put forward:

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 a few examples: relativity theory is not an integral part of calculating artillery ballistics, the smallpox vaccine did not need a theory of germs, and you don't need to understand convection to build a good chimney. 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 linguistic ideas for the sake of simplicity and efficiency. These tools do surprisingly well and often outperform competing models that draw from what linguists have learned about language. The state of affairs is summarized very succinctly by a hyperbolic quote that is commonly attributed to the computational linguist Frederick Jelinek:

> Every time I fire a linguist, the performance of the speech recognizer goes up.

Frederick Jelinek (1932–2010)

Frederick Jelinek played a key role in bringing information theory and probabilistic methods to computational linguistics, or rather, bringing them back from the dead.

Following the success of Chomsky's *Transformational grammar* in the 50s and 60s, computational linguists put their hope in rule-based approaches and largely stayed away from statistics and probabilities. Jelinek bucked this trend. After he joined IBM in the 70s, he worked tirelessly on designing speech recognition systems that were sufficiently robust for real-world application. The more theoretically minded, rule-based approaches had nothing comparable to offer. By the 1990s, the probabilistic models Jelinek pioneered had become a corner stone of NLP, and they remain important to this day. But perhaps the pendulum has swung a bit too far — the rule-based side may still come back with a revolution of its own.

There are numerous examples of simple (or downright simplistic) models matching or even outperforming more sophisticated models that are deeply steeped in theory. Consider the case of a 2013 study that evaluated various text-based models for predicting 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. As it turns out, successful novels have an unusually high frequency of thought-oriented verbs like *recognize* and *remember*. A model that only pays attention to such word frequency effects performs better than one that also analyzes the minute structural intricacies that linguists care about.

It seems, then, that there is a shockingly large gap between "language as a computational problem" and "computers solving natural language tasks". The latter is doing just fine without the former, and in some cases theory may even make things worse for practical applications.

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 have made major inroads in recent years, in particular in the form of neural networks. The approaches deliberately forego all linguistic knowledge and instead opt for treating language as an arbitrary data-crunching problem like any other. And once again they turned out to be surprisingly performant. If this trend continues to its logical extreme, NLP may be completely absorbed into the field of machine learning in the near future. 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" might be too specific an enterprise to make many worthwhile contributions. In sum, then, computation linguistics might have few actual contributions to make to NLP.

And thus we have arrived at the conclusion of the devil's advocate: while in principle computation linguists might be able to help with practical applications, recent history tells us otherwise. Other fields like physics and chemistry might show a trickle-down effect from theory to applications, but a look at the field right now

reveals no comparable pipeline from computation linguistics to NLP.

A counterargument to the counterargument

There is more than a grain of truth to the devil's advocate's reasoning, but at the same it is too narrow and overly reductionist. Things simply aren't that black-and-white, and a more nuanced analysis reveals interesting shades of gray.

Admittedly some areas of application do not stand to 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. The nature of this task is largely independent of the complexities 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 particularly 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.

But not all problems are simple or very restricted, and this leaves room for computation linguistics to make worthwhile contributions. Language technology is still very much about going for the low-hanging fruit. Many interesting problems remain untackled. 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 in 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 JPGs or PNGs to be renamed, whereas the PDF 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 in the picture": What is the intended interpretation here? People that appear in the picture should have their name listed in the filename? Each person should have their name listed in the picture? List every name that belongs to a group of people and appears in 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 they still struggle with language-related tasks that even a 5-year old would solve with ease. Once one broadens the horizon from what NLP is currently working on to what NLP could be working on, the importance of computation linguistics becomes much more apparent.

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 Swahili 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 proved thousands of times each day all over the world. Each day, children in linguistic communities of all shapes and sizes 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 our models will need, thereby shrinking the gap between resource-poor and resource-rich languages.

It is also worth noting that the performance of NLP models should not be taken at face value because it is based on a particular notion of performance. The NLP community has devised numerous quantitative metrics for benchmarking its models. But these benchmarks have several blind spots in their design. For one thing, they only measure how many errors a model makes, rather than how grave the errors are.

Suppose you have to evaluate how well two children — let's call them Shorty and Luna — have mastered basic addition. The table below shows your questions and their respective answers.

Question	Shorty	Luna
What is 3+4?	6	7
What is 6+3?	9	9
What is 7+3?	10	10
What is 3+4+3?	9	343

Looking purely at the number of correct answers, Luna does better than Shorty. But every teacher will tell you that Shorty has a better grasp of addition than Luna. Yes, Shorty is short by 1 when adding up 3 and 4. But once you take that mistake into account, all the other answers make sense. In particular, under Shorty's mistaken assumption that $3+4=6$, it is necessarily the case that $3+4+3=6+3=9$. Luna, on the other hand, missed this important generalization and gives a very puzzling answer to the last question. Instead of adding, Luna suddenly concatenates the numbers. That is extremely bewildering, and a teacher who only evaluates students based on how many answers they get wrong would miss this.

You might object that the problem here isn't with counting the number of mistakes, but rather with the small number of questions we asked. If we had asked more questions of the form $3+4+3$, Luna would have done worse than Shorty. You are correct, and this is also the answer you will hear in NLP. NLP models are tested on millions of sentences, which the example above obviously does not do justice to. But the size of a test sample isn't a solid indicator of its quality. What if Luna's problem only occurs with sequences of addition of the form $n+(n+1)+n$? Even in a test set with millions of questions, this pattern might be fairly rare, whereas Shorty's problem with adding 3 and 4 might crop over and over again. With addition it is fairly easy to design a balanced test set, but with language NLP engineers rely largely on the data that is freely available in the wild. This includes tweets, posts on message boards, news paper articles, and so on. Linguists have argued for a long time that these samples are not representative of the richness of language. The most intricate aspects of language tend to reveal themselves only rarely, but when they do they are of great importance.

As a result, NLP performance metrics should be taken with a grain of salt. Yes, model A might outperform model B by 10 points in your benchmark. But model B might still make a better impression on users because its errors aren't nearly as serious or puzzling as those made by model A. When interacting with humans, slightly wrong beats really wrong, and wrong beats weird.

Such qualitative performance metrics are still sorely missing in NLP. A good theoretical grounding — where "theory" subsumes both theoretical linguistics and the theory of computation — is a guide through those areas of language where data is insufficient. The more language-like a model, the less data it should need to display language-like behavior.

In sum, then, NLP approaches still have their fair share of shortcomings that insights from computation linguistics can help address. As NLP moves into increasingly complex domains of language, with increasingly impoverished data, the use of a strong theoretical foundation will become increasingly apparent.

An example from the history of computational linguistics

You might still be skeptical. All I have done is point out some shortcomings of the current NLP approaches that could potentially benefit from computation linguistics. I still owe you a concrete example of a fruitful transfer of knowledge. Fair enough, so here we go:

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 the framework of *finite-state 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. Nowadays, finite-state machinery is still used for modeling specific aspects of morphology, e.g. number systems. This is a domain where good data is scarce and mistakes can have severe consequences — there's a big difference between transferring $23.45 and $2,345. The linguistic insights have undergone a voyage from theoretical linguistics to computation linguists to NLP, where they have proven tremendously useful for over 30 years now. But insights have also traveled in the other direction. A few processes in morphology that involve copying pose serious challenges to finite-state morphology, highlighting the need for a better linguistic understanding of these phenomena. This way theoretical and computational linguists feed each other's research, with NLP as the happy consumer of the final results.

Sometimes the transmission of insights is almost imperceptibly indirect, and is easily lost in the fog of history. Formal language theory — which is an integral part of computation 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's work on tree transducers, and those are now used in machine translation and compiler design, the latter being indispensable for modern programming languages. The computational linguist Aravind Joshi relied on key insights about the nature of language when he developed *Tree Adjoining Grammars* (TAG), which then found applications in modeling biochemical structures such as messenger RNA. The formalism of *Combinatory Categorial Grammar*, which is equally driven by engineering concerns and pure linguistic theory, has been used in studying the structure of music. As you can see, there is no telling how fields will cross-fertilize each other as their ideas and insights slowly transgress disciplinary boundaries. But there are concrete cases of such exchange between computational linguistics and numerous other fields, including NLP.

Interim Summary

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. It also depends on the quantity and quality of available data, and the need for strong safeties — minor mistranslations are acceptable for a blog post, but not in a business contract. 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 benefits a lot from a good theoretical foundation. So, no, computation linguistics isn't the magic bullet for solving all of NLP's woes, but it has valuable contributions to make in key areas. These key areas may well become the central focus of attention soon. If the current staples of NLP end up being subsumed by general purpose machine learning techniques, we may see the field move towards harder problems that are ripe with language-specific issues.

2.2 Scientific arguments

The previous section defends computation linguistics in terms of its utility for NLP. These utilitarian arguments are needed to 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. If computation linguistics can somehow get NLP to incorporate more linguistic insights, that's nice. It probably won't tell us much about language, though. So it won't improve linguistics, psychology or cognitive science. Fortunately, the scientific merit of computation linguistics is a lot more clear-cut than the utilitarian argument: since language is intrinsically a computational problem, studying language from a computationally informed perspective is only natural.

Neurolinguistics and psycholinguistics have already uncovered a lot about the computational side of language, whereas linguists have a lot to say about the rules and representations that make up a speaker's knowledge of their language. Integrating the two, however, has proven very difficult for a variety of reasons. Computation linguists provides a bridge to sythesize these diverging perspectives on language into a unified picture.

Competence and performance

In order to make the argument for computation linguistics, we first have to put in place the distinction between **competence** and **performance**. Probably the most important development in 20th century linguistics is the *cognitive turn*, the shift from viewing language as a disembodied 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, its hardware is often called the *wetware* to emphasize the contrast to man-made machines. Specific languages are just specific externalizations of this abstract program "language" that runs on the wetware. Languages are but a window into a specific kind of computation that the human brain carries out with little effort. Since the cognitive turn, theoretical linguistics is no longer about languages, it is about language.

But if language is a computation carried out by wetware, this immediately raises the question how exactly 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. A native speaker of English, for instance, recognizes that in the two sentences below, only the first one is ambiguous.

(1) a. Who do you want to leave?
b. Who do you wanna leave?

The first question has two meanings, paraphrased as "Who is such that you want them to leave?" and "Who is such that you want to leave them?", respectively. The second question only allows the latter interpretation. Competence describes the implicit knowledge of a speaker that is needed for him or her to recognize such a contrast. But the actual process of bringing competence to bear on these questions is a much more involved matter. That's what performance is about.

Intuitively, we may think of a speaker's competence as a book that records the rules and laws of their language. Those rules might look slightly different from other speakers — even when two speakers share the same language, they often have very minor differences in what sentences they consider well-formed, which words they use, and so on. But that is not the key point here. What matters is that a book of law, by itself, does not do anything. We need a law enforcement mechanism, a system for analyzing things in the real world, putting them in relation to those laws, and acting accordingly. In the real world that's judges, police, but also the law-abiding citizen that modulates their own behavior according to the laws they are aware of. That is the metaphorical counterpart to linguistic performance.

With language, performance takes on many facets: analyzing the incoming sound waves, filtering out irrelevant background noise, breaking down the continuous waveforms into discrete chunks of vowels and consonants, assembling sounds into words, combining words into sentences, interpreting the sentence, marshalling sufficient working memory to store all this information, constructing a reply, meticulously controlling the articulatory apparatus to utter the intended reply without any mispronunciations or slips of the tongue, and much more. Performance is a huge chunk of language, and yet it can be very insightful to factor it out and focus only on competence.

Some linguists do not like the idea of factoring out performance, they consider competence too much of an abstraction. Among other things, a competence-view of language implies that we have to look at language modulo any processing restrictions of the human brain, e.g. limits on working memory and attention span. Of course nothing of this sort can be observed in nature. You can't have human language without the human brain, and that mushy, three pound lump is constantly subject to very tight limits on working memory and attention. So there is no way of directly observing competence, every piece of linguistic data is the compound result of both competence and performance.

But something like the competence-performance split is actually very common in many other fields, including computer science. Here is a concrete example. When UPS was looking for the most efficient system to ship a package from A to B, they did not care if their solution would still work well if they had to be able to ship to 50 trillion different cities. That is irrelevant because Earth doesn't even have that many cities, so

the issue simply would not arrive in practice. Their problem space is bounded in size. Yet there is a whole branch of computer science, called **complexity theory**, which studies how hard certain problems are to solve if one assumes that there is upper bound on the size of the problem. For instance, how hard is it to sort a list if there is no upper bound on how long that list can be. In practice, this does not matter because a computer's memory is limited, which puts a hard upper bound on the length of lists it can work with (that's in contrast to a Turing machine, where we assumed that we never run out of tape). So why would computer scientists study a problem that can never arise in the real world? The answer is that these unbounded problems are often more insightful than the bounded ones. The abstraction from a bounded problem to an unbounded problem makes it clearer what the challenging parts are and how they could be optimized. We will actually see this at various points throughout this book. As counter-intuitive as it may seem, you often have to abstract away from reality in order to get closer to reality.

Just like computer science has benefited from looking at problems of unbounded size, linguists have learned a lot about language by factoring out performance and focusing on competence. But the competence level is rarely viewed through a computational lens, even though language, by virtue of being a cognitive ability, is intrinsically tied to computation. Research into how language is computed usually limits itself to the realm of performance, approaching the issue from the perspective 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 (2) is grammatically correct, native speakers usually say no.

(2) The player tossed a frisbee smiled.

However, the sentence is actually well-formed, it has the same structure as the minimally different (3).

(3) The player thrown a frisbee smiled.

For some reason the algorithm native speakers of English use has no problem with (3) but is thrown off track in (2) which differs in only 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 computation linguistics begins.

Neuroscience

Neuroscience has made tremendous progress in the last few decades, and as a result there has also been increased interest in neurolinguistics. As in many other areas of neuroscience, the hope is that the "code" of the human brain will reveal itself through its wetware. Know the hardware, and you'll know the software; unfortunately, things are not that straight-forward.

It is certainly interesting to see how the human brain works on the hardware level. But it is far from obvious that this tells us much about the actual computations. Nothing we have learned about computation since Turing's pioneering work supports this notion, intuitive as it might be. Studying a computer's hardware tells us little about what it is computing. Suppose your computer still has an old rotary hard drive, rather than an SSD. When we hear your hard drive spin up, we can reasonably assume 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 this or that piece of hardware starts heating up. Increased heat output would suggest an increase of computational activity in that specific area. A hot graphics chip, for instance, would be indicative of some high graphical load. 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. This is why the acronym GPU, which stands for *Graphics Processing Unit*, has been extended to GPGPU, which is short for *General Purpose Computing on GPUs*. Among other things, GPUs are now commonly used in machine learning tasks. Content-wise, this has very little to do with graphics rendering, but the mathematics are actually fairly similar. This is one reason why it is difficult to draw conclusions from hardware to software — two very different domains A and B can instantiate very similar computational problems, so that a device that looks like it is computing A might actually be computing B.

A curious example of this principle is the first-person shooter *Doom*, well-known for its high-adrenaline gameplay and impressive music. When this classic video game was ported from PC to Atari's *Jaguar* console, buyers and reviewers alike lamented the lack of any in-game music. The reason for that is surprising and insightful at the same time. At the time, Doom was a very demanding game, too demanding for the Atari Jaguar. Its CPU simply wasn't beefy enough to handle the game. The developers, however, realized that some of the game functions could be off-loaded to another chip, freeing up more CPU cycles for the rest of the game. But this meant that this other chip could no longer serve its intended purpose as it was busy with other computations. Yes, you guessed it, that chip's standard role was to handle the music. There was no other chip with free computing cycles for the music, and thus the music had to be completely cut from the port. While ultimately unsatisfying for Atari Jaguar owners, the developers' decision to co-opt a music chip for general purpose computation was genius.

Something similar has actually been observed in humans, where the visual cortex of a blind person will be partially co-opted for auditory computations. Admittedly the human brain is not as content-agnostic as a modern computer, and overall specific tasks seem to be tied to specific areas of the brain. But there is at least some flexiblity, and it is too early to say how tight this link actually is from a computational perspective. Suppose task X is localized in brain area A. Then a computationally similar task Y may or may not be localized to area A, there is no way of telling in advance. 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 does not tell us much about the nature of computation.

There is also a severe granularity mismatch between hardware and computation that makes it very hard to make meaningful inferences from the former to the lat-

ter. This argument was made very forcefully by Jonas and Kording in 2017. They started with a plausible assumption: if current neuroscience techniques can tell us something about computation, then they should be able to tell us something about the computations carried out in a computer. If we take a computer, give it a specific task to run, and collect measurements that are comparable to what neuroscientists have collected for the human brain, we should be able to detect reflexes of the computations in the data. As you might imagine, this did not work well at all. Jonas & Kording (2017: 1) concluded that "current analytic approaches in neuroscience may fall short of producing meaningful understanding of neural systems". If the hardware level can't give us conclusive evidence about the computations in a device we have perfect knowledge of, it is doubtful that the human brain can be figured out by just studying its hardware.

This basic point become downright trivial if we look at a concrete example. Consider the simple task of searching through a list. Suppose, for instance, that 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. It can be accomplished in various ways, each with their own advantages and disadvantages, but the two most fundamental strategies are *linear search* and *binary search*.

Example 0.2 Linear search and binary search

Suppose that our list is A, C, D, F, G, X and we want to know if G is in the list.

Using the linear search algorithm, 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 give a negative reply. The table below shows what happens at each step of the computation when searching for G in the list A,C, D, F, G, X.

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 a specific property of the list. While in principle the symbols in a list can appear in any random order, our list A, C, D, F, G, X lists the symbols in alphabetical order. This fact can be used to avoid searching through irrelevant parts of the list.

Instead of 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. After all, if the middle symbol alphabetically precedes the symbol we are looking for, then so do all the symbols to the left of the middle symbol. Hence it makes no sense to search there. After the left half has been removed, we perform

binary search on the remainder, i.e. the right half. On the other hand, 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, i.e. the left half. We continue this way, throwing away one half in each step and looking at the middle point of the remaining half. If the list contains the item we are looking for, we will find it eventually. If at some point we have thrown away the whole list, the item wasn't in the list.

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. That's because binary search can skip large chunks of the list. But crucially this only works if we have a guarantee that the list is ordered in a specific way — otherwise, we can't tell which half of the list can be ignored at any given point.

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. Yet the behavior cannot be explained in terms of hardware, because we know for a fact that the two machines are exactly the same. This is why good algorithms and data structures are so important in computer science — progress on the hardware side cannot match the massive speed-ups of smart software. The ingenuity of the human brain probably does not lie in its hardware, at least not exclusively. It stands to reason that millions of years of evolution have resulted in algorithms that have been tweaked, tuned, and optimized to be as efficient as possible for their respective domain. Hardware alone is not enough, the true magic happens at the software level, which does not reveal itself transparently via hardware.

Keep in mind that in the case of computers, we have the additional advantage that we already know how their architecture works because we designed it. So if that still is not 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. A strict bottom-up approach

that figures out the human mind solely based on wetware is unlikely to succeed.

In the domain of language, we have the potential for a bidirectional approach that proceeds both top-down (from computation to hardware) and bottom-up (from hardware to computation). Computation linguistics can contribute towards such a linking theory by identifying computational core mechanisms of language that are sufficiently general that they may be detectable in the wetware with current day techniques. Computation linguistics is the glue that links fine-grained theories of competence to computational concepts that are sufficiently broad to possibly have some detectable reflexes at the hardware/wetware level.

Psychology

In contrast to neuroscientists, psychologists don't investigate the physical instantiation of cognition. They are perfectly happy to treat the brain's wetware 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 limitations that computation linguistics can help address.

One central issue is again a granularity mismatch between the explanandum — language as a cognitive ability — and the means of explanation. Psycholinguistic models can be highly specific in what cognitive parameters they presuppose.

Example 0.3 A psycholinguistic account of priming

A common assumption in the literature 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 some item in memory at a given address, all one needs is this address to immediately retrieve the item 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*).

A very extreme case of psychologists and psycholinguists making highly specific assumptions is the ACT-R framework. ACT-R provides an elaborate computer model of human cognition. The specificity of ACT-R allows researcher to run detailed simulations and tweak parameters to find the best fit for experimental data. ACT-R is an impressive achievement, but it is a machine with many knobs, cogs and gears. This makes it difficult to ascertain what each component contributes to the final result. This has two downsides. First, the findings aren't as insightful as one might hope. Simulations may

tell us that configuration X accounts for the facts, but what about X′ or X″? Do we need all the components interacting in this specific manner, or are most assumptions dispensable except for maybe one or two?

You may think that we can answer this by just running additional simulations with other parameter values. But this is often not practically feasible due to combinatorial explosion. If your model has 10 binary parameters, then there are $2^{10} = 1024$ different instantiations of the model. Even if your simulation takes only half an hour to complete, you're already looking at 512 hours, which is over three weeks. In practice, your model will take longer to run and have many more parameters to tweak, so exhaustive simulation of all options isn't feasible. Computation linguistics, on the other hand, has the advantage that its concepts, albeit abstract, are sufficiently simple to be explored from a mathematical perspective using little more than pen and paper. While this certainly poses its own challenges, it often takes much less time and effort than designing models and running simulations.

Putting aside the time investment factor, we are still left with the problem that one cannot generalize from modeling results in a monotonic fashion. That model M captures phenomenon P does not entail that it also accounts for phenomenon P′. Once we make tweaks to account for P′, we have to rerun our simulations for P to ensure that everything still works as desired. The mathematical approach favored by computation linguistics, on the other hand, provides crucial clues under which conditions key properties are preserved, and this makes it easier to make inferences from P to P′.

This is not to say that the modeling approach is inferior. It has the major advantage that it is close to the empirical realities. The models are complex, but their behavior and predictions can be directly tested against the observed data. The highly abstracted and idealized concepts of computation linguistics require more effort to bring to bear on the data. Some problems are so complex that it is far from obvious how they should be idealized in order to make them amenable to mathematical investigation. Just as with neuroscience, this isn't a case of one approach supplanting the other, but rather of natural complementation.

Some psycholinguists may object, though, that the previous paragraphs only engage with a particular subpart of their field, namely the cognitive modeling approach. Even if one grants that all the postulated shortcomings are on the mark, that would only apply to a small part of psycholinguistics. The overwhelming majority of psycholinguists operates with less specific models and theories. These models may be too coarse for computational simulations, but they work just fine as a generator of new research questions that can be explored through experiments. While correct, these models face the opposite problem of lacking the fine-grained details that would make them sensitive to issues of linguistic competence and theoretical linguistics. They cannot distinguish between competing analyses or illuminate whether language is rule-based or constraint-based. Computation linguistics provides the scaffolding to explore these issues and identity properties that might be detectable in psycholinguistic experiments.

Interim summary: The promise of computation linguistics

The limitations of neuroscience and psychology pointed out above can be traced back to a comparatively low degree 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 computation linguistics, the one advantage that sets it apart from neuroscience and psychology, is one of *productive abstraction*. Computation linguistics can completely abstract away from all extraneous detail and performance aspects without losing the connection to cognition. The methods used by computation linguists allow us to connect language and computation (and hence cognition) at the competence level. We can tackle 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 The virtue of abstractness

3.1 Marr's three levels of analysis

The preceding observations regarding abstractness are far from new. In 1976, David Marr and Tomaso Poggio formulated the *Tri-Level Hypothesis* in their paper *From Understanding Computation to Understanding Neural Circuitry*. They proposed that computational processes, including cognition, should 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; many aspects of programming deal with issues at this level

computational what problem space does the system operate on, how are the solutions specified (but not necessarily carried out)

Example 0.4 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 ($A \cap B$ in mathematical notation).

- 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. In your laptop, 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 probably hopeless to reverse engineer the program from the physical evidence.

David Marr (1945-1980)

David Marr was a British neuroscientist who did revolutionary work on visual processing. Marr took a very pragmatic approach to science. He maintained that working on concrete empirical problems is more fertile than sterile debates about methodology and theoretical primitives (so he probably wouldn't have liked this chapter). History presumably proved him right, as it is very doubtful that his Tri-Level Hypothesis would have caught on if it weren't for the strengths of his model of vision, which combines the physical, algorithmic, and computational level for maximum effect.

The tri-level hypothesis highlights 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 phone 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 computation linguistics is aiming for.

3.2 Abstraction necessitates formal rigor

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 ultimately notation will make reading easier and faster. 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 a scientist is the ability to enjoy feeling stupid:

> The more comfortable we become with being stupid, the deeper we will wade into the unknown and the more likely we are to make big discoveries. (Schwartz 2008)

4 Summary

4.1 Key insights for Unit 0

1. **Computation** and computers are not the same. Computation is an abstract concept that can be instantiated by various objects, including non-existing ones. Computers are one such object, but so is the human brain, a cell, or a line of water buckets when manipulated appropriately.

2. **Turing machines** show how little is needed to carry out very complex computations. A movable read-write head, a state register, and an unlimited tape are sufficient to solve even the most complex computations. But figuring out the correct set of instructions may be hard.

3. **Natural language processing** (NLP) is focused on solving language-related tasks with computers. A different aspect of computational linguistics (which I call **computation linguistics** for lack of a better term) instead seeks to study language as a computational problem. This is the approach presented in this textbook.

4. A cognitive system like language can be described at various levels of abstraction, in the spirit of **Marr's three levels of description**. These are, in increasing order of detail: *computational*, *algorithmic*, and *physical*. The computational level allows us to abstract away from many details that are irrelevant for the issues linguists care about.

5. Abstraction is a virtue. It allows for profound insights and succinct generalizations that could not be stated at more fine-grained levels.

6. Neurolinguistics and psycholinguistics operate at less abstracted levels, as do approaches based on computational modeling. Each one has valuable contributions to make, but integrating insights from the different levels remains one of the hardest problems in cognitive science.

4.2 Relevant literature for Unit 0

Computation linguistics and related fields One of the central themes of this chapter is the difference between NLP on the one hand and computation linguistics on the other. This contrast is very salient when comparing Wilks's (2006) history of NLP to Penn's (2006) overview of symbolic computational linguistics (which is more closely related to the computation linguistics presented here). It is particularly striking how differently the two papers view the historical role and contribution of linguistics. The papers might be a difficult read at this point, but even if the jargon and concepts are still opaque, the spiritual difference between the two approaches emerges clearly.

Pullum & Kornai (2003) and chapters 1 and 10 of Kornai (2007) present an overview of *mathematical linguistics*. As the name suggests, mathematical linguistics studies language with the help of mathematics. This may sound very similar to computation linguistics, and the two do have significant overlap. They are not the same, but there is no agreed upon list of criteria to distinguish them. One answer is that mathematical linguistics draws largely from traditional mathematics, in particular logic and algebra, whereas computation linguists builds on methods from theoretical computer science. I prefer a different split, with mathematical linguistics as a methdological classification whereas computation linguistics is about the subject matter. Mathematical linguistics is then defined by its mathematical methodology and can be brought to bear on any linguistic issue, irrespective of whether it has a direct link to speakers' mental computations. Computation linguistics, on the other hand, is a subfield of linguistics that explores the computational aspects of language. A mathematical linguist may, for instance, propose a rigorous model of how loanwords spread through a linguistic community. This phenomenon occurs at a higher level than that of individual language users and thus is not directly connected to specific mental computations. At the same time, computation linguistics can include work that has very little mathematics in it, e.g. the analysis of some empirical phenomenon using a computationally informed formalism.

Krahmer (2010) discusses the interplay of computational linguistics and psycholinguistics. Crocker (2010) is a more extensive survey of the subfield of computational psycholinguistics. Hale (2014) carefully develops a powerful framework for studying sentence processing from a computational perspective. There are numerous publications on ACT-R, but a computationally minded reader is best served by Whitehill (2013).

Alan Turing The Turing Machine was first discussed in Turing (1936, 1938). Hodges (1983) is a gripping biography of Alan Turing; it was out of print for a long time, but has been reprinted after the release of Hollywood's biopic *The Imitation Game*, with Benedict Cumberbatch as Alan Turing. Turing has also been tremendously influential

in the domain of artificial intelligence. For a technically light-weight but sprawling pop-science introduction to this topic, see Hofstader (1979).

Linguistics beyond language Section 2.1 lists several examples where approaches from theoretical and/or computational linguistics made their ways into seemingly unrelated applications. All these approaches will be discussed more extensively in later chapters, but the curious reader may already take a gander at the following references: Chomsky (1957, 1959) and Chomsky & Schützenberger (1963) are seminal papers from the very beginning of what is now called formal language theory. One of the first papers on tree transducers is Rounds (1970), who explicitly names Chomsky's Transformational grammar as a major motivation for his work. Tree Adjoining Grammar (Joshi 1985) is a different, computationally minded grammar formalism. For applications of TAG in molecular biology see Uemura et al. (1999) and Matsui, Sato & Sakakibara (2005).

Other references The study on predicting the success of novels is Ashok, Feng & Choi (2013). The tri-level hypothesis is formulated in Marr & Poggio (1976) and Marr (1982). For recent findings on how how specific brain areas may be co-opted for different task, see Bola et al. (2017) and references therein. Jonas & Kording (2017) apply current techniques in neuroscience to the analysis of microchips to show that these methods provide insufficient insight into computation.

P Programming

P.1 Linear search and binary search

In Sec. 2.2, example 0.2, we saw that looking at computation in terms of hardware fails to illuminate even basic points such as the difference between linear search and binary search. This point becomes even more pronounced if we look at how these search algorithms can be implemented in a specific programming language.

Linear search is one of the simplest search algorithms and can be implemented in 4 lines. The code here implements it as a function that returns either `False` or an integer. `False` is returned iff the search item is not in the list. When an integer is returned, it indicates the leftmost position of the search item.

The docstring under the function name specifies the intended type of each argument. The type of the search item is set to `any`, which means that there are no restrictions on its type.

```
def linear_search(search_list, item):
    """Left-to-right search for position of item in search_list.

    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
    """
    # iterate over all positions in the list
    for i in range(len(search_list)):
        if item == search_list[i]:
            return i
    # if we made it this far,
    # then we haven't found anything
    return False
```

Here is a concrete example:

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

Now let's contrast linear search against binary search. As you can see, the code is slightly different from the intuitive description in example 0.2. There we said that binary search picks the middle item of the list. If the middle it is not the search item, one discards one half of the list and runs binary search on the remainder. This is a recursive definition: an instance of binary search can spawn another instance of binary search. Recursive definitions are elegant and mathematically pleasing, but they are not always the best way of implementation.

Python in particular is not well-optimized for recursion and probably will never be because it would require some fundamental design changes with major sacrifices. Fortunately, binary search does not require recursion, that is just one of many different ways of describing the procedure. The implementation below uses `while`-loop instead, which avoids Python's issues with recursion but looks very different from the intuitive description of binary search. With another programming language, e.g. Haskell, the latter might be the more natural and efficient implementation.

```
def binary_search(search_list, item):
    """Use binary search to find position of 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;
        # we use int() for rounding down
        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
    # Case 4: item not in list, while-loop aborted
    return False
```

```
>>> 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
```

This shows very clearly that one and the same idea can be implemented in many different ways. Which way is best often depends on very subtle and arcane details that have very little to do with the problem itself. In the case at hand, an idiosyncrasy of Python makes a `while`-loop preferable to recursion, but that has no bearing on binary search in general and how it compares to linear search. Such implementation details tend to obscure what really matters, and this is why we should always strive for a level of description that abstracts away from implementation-dependent differences.

P.2 Set intersection

Example 0.4 briefly raised the question how exactly set intersection could be implemented at the algorithmic level. Python actually provides an out-of-the-box solution in the form of a set data type that provides the method `intersection`.

```
>>> A = {1, 2, 3, 4}
>>> B = {2, 4}
>>> A.intersection(B)
{2, 4}
```

While useful and very fast, this solution hides all the interesting parts under the hood. Let's first look at a piece of code that only uses the most basic programming concepts and is also very intuitive. Unfortunately, it is also very inefficient.

```
def list_intersection(listA, listB):
    """Build a list that only contains elements of both lists.

    This algorithm is highly ineffecient
    because it loops over the second list multiple times!

    Parameters
    ----------
    listA : list
        first list of elements
    listB : list
        second list of elements

    Returns
    -------
    list

    Examples
    --------
    >>> list_intersection([3, 1, 2], [4, 7, 2, 1])
    [1, 2]

    >>> list_intersection([3, 1, 2], [])
    []

    >>> list_intersection([3, 1, 2], [1, 2, 3])
    [3, 1, 2]
    """
    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is any b of listB the same as a?
        for b in listB:
            if a == b:
                # found a, add it to intersection
                intersection.append(a)

    # all loops done, return intersection
    return intersection
```

Here we create an empty list and only add an item to it if said item occurs in both lists. Doing this requires two `for`-loops. Notice how we do a full iteration over the second list for each element in the first list. If the first list contains m items and the second one n, we perform a total of $m + (m \times n)$ lookups. So the number of lookups grows much faster than the combined number of elements in the lists.

The obvious problem is that we look at the second list over and over again, rather than just processing it once and memorizing what elements occur in it. In Python, such memorizing takes the form of converting a list to a dictionary (also known as a *hash map* or *hash table* in other programming languages). A dictionary uses keys as an addressing system for quickly looking up items.

```
>>> A = {'some_key': 'the_value',
         'key_2': [5, 3],
         10: 'integers can be keys, too'}
>>> A.get('some_key')
'the_value'
>>> A.get('key_2')
[5, 3]
>>> A.get(10)
'integers can be keys, too'
```

Instead of searching through the second list over and over again, we first convert it to a dictionary and then use that for quick lookup.

```
def list_intersection(listA, listB):
    """Build a list that only contains elements of all lists.

    This algorithm uses dictionaries for speed,
    but requires more memory.
    """
    # convert second list to dictionary
    dictB = {item: item for item in listB}

    # create empty intersection
    intersection = []

    # for each item a of listA
    for a in listA:
        # is a in listB?
        if dictB.get(a):
            # found a, add it to intersection
            intersection.append(a)

    # return intersection
    return intersection
```

With larger lists, this implementation will be much faster. And it is essentially what the very first solution with sets is doing, because Python's `set` data type is just a special case of Python dictionaries. For real-world applications, seemingly minor

differences like this can have major repercussions. But if the lists are always very small, memorizing them with dictionaries might not be worth it. There are no simple answers here, it all depends on the specific use case.

You already saw in the case of linear search versus binary search how implementation details can obscure the bigger picture. But here the issue wasn't even choosing between largely different algorithms, but just how exactly one wants to implement a (very simple) operation. The two pieces of code we came up with are almost exactly the same, yet they differ greatly in their runtime behavior.

E Exercises for Unit 0

E.1 Theory

Exercise 0.1 Consider once more the Turing machine from example 0.1. Show how this Turing machine operates on an input of the form 011110, with the read-write head starting on the rightmost 1 in state A. Since the tape of a Turing machine can be expanded without limits, you may pad it out with 0s as necessary. This does not require any special instruction for the machine.

Exercise 0.2 Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: 0, 1, and ⋄ as a special symbol for cells that do not contain one of the two digits.
2. The Turing machine takes as its input a single string of 0s and 1s (e.g. 001101 or 1010110).
3. All other cells of the tape are filled with ⋄.
4. The machine starts on the leftmost symbol of the input.
5. The machine outputs a string that is almost exactly the same as the input, except that the first symbol is replaced by $1-n$, where n is the value of the last symbol. For instance, 11101 becomes 01101, whereas 01101 and 11100 stay the same.

Exercise 0.3 Continuing the previous exercise, modify your machine so that it works even if the read-write head can start on any random symbol of the input string.

Exercise 0.4* Specify a Turing machine that satisfies all the following requirements:

1. The machine uses only the following symbols on the tape: digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and ⋄ as a special symbol for cells that do not contain a digit.
2. The Turing machine takes as its input a single number n between (and including) 0 and 999.
3. The machine starts on the leftmost digit of the input.
4. The machine outputs $1000-n$ (leading 0s are allowed, e.g. 0001 instead of 1).

To save yourself some writing, you may conflate multiple lines by using variables. So if 7 is rewritten as 3 and 8 as 2, you can just have a line rewriting n as $10 - n$ and adding "n is 7 or 8". As long as there are only finitely many choices for n, this does not change anything about the machine, it is just a way to represent multiple instructions with a single line.

Exercise 0.5 Continuing the previous exercise, look at the specification of your Turing machine. Is there any individual part that captures the "essence" of substraction? Or does the whole come about only through the interaction of the individual parts?

Exercise 0.6 The practical argument for computation linguistics presupposes that man-made engineering solutions can be improved by learning from nature — in the case of NLP, human language use. But at the same time, airplanes aren't built like bumblebees, and submarines aren't fish. Doesn't this suggest that technology carves out its own way, with its own solutions? Formulate an extended argument for this position, and one against it. Which one do you find more convincing?

Exercise 0.7 Consider once more the sorted list A, C, D, F, G, X. Using the tabular format from example 0.2, show how linear search and binary search move through the list when looking for item C.

Exercise 0.8* Binary search is only guaranteed to work correctly with ordered lists. But this does not imply that binary search necessarily fails on unordered lists. Write down an unordered list with 5 elements A, B, C, D, E such that binary search will find A but not E.

E.2 Programming

Exercise 0.9 Given the Python definition of `linear_search` in this chapter, what is the output of `linear_search([1,1,1,1], 1)`?

Exercise 0.10 In Python, `0` is sometimes treated the same as `False`. This can result in some unexpected bugs, for instance with the piece of code below. What exactly is the problem? How would you fix it? Is this an issue that a high-level definition of linear search should address?

```
if linear_search([1,2,3,4], 1):
    print("Item in list!")
else:
    print("Item not in list!")
```

Exercise 0.11 Write a recursive implementation of binary search.

Exercise 0.12 What would a recursive implementation of linear search look like? Compared to binary search, do you find the recursive implementation of linear search simpler, more convoluted, or about the same in complexity?

Exercise 0.13* Consider the first version of the list intersection function in this chapter, which does not use dictionaries. Instead of the second `for`-loop, we could have used Python's built-in membership test for lists:

```
if a in listB:
    intersection.append(a)
```

Would this have solved the speed issues? Justify your answer. In order to do so, you might have to do some research on how exactly Python's `in`-operator works.

Exercise 0.14* The problem with multiple `for`-loops gets even worse when computing the intersection of multiple lists. For these cases, then, it's even more important to have an efficient solution.

Generalize the dictionary-based intersection function so that it can take an arbitrary number of lists as arguments. All lists except the first one should be converted to a dictionary.