Lecture 1

The Big Picture: Why Parsing isn't Sentence Processing

On a purely technical level, parsing is the process of assigning a string of symbols a structural description according to some formal specification, usually a grammar. As such, parsing is not specific to language, any kind of problem where hidden structure has to be inferred from linearly arranged items may be considered an instance of parsing. This covers a very diverse range of processes such as discerning the structure of source code, segmenting a movie into its three acts, and even protein folding (the mechanism by which sequences of amino acids combine into these complex three-dimensional objects we call proteins).

This general view of parsing is possible thanks to the semantic agnosticism of formal languages: pretty much everything can be conceptualized as a formal language. A program is a sentence of some programming language, and the programming language is defined via an alphabet (variables, functors, data structures, etc.) and a grammar that defines how the elements of the alphabet can be concatenated. The set of well-formed proteins is also a language that is defined by an (unfortunately still unknown) grammar with amino acids as its alphabet. And of course natural languages can be viewed as formal languages, as is commonly done in computational linguistics. In this case, parsing is about assigning structures to natural language utterances.

Linguists might interject that this description is overly general: natural language parsing must assign *tree-like* objects to natural language utterances that match, in some suitably abstract sense, the structures subconsciously employed by native speakers. This is indeed the standard view of parsing in linguistics, but it is a very specific notion that is tailored to an equally specific set of goals. The linguistic definition treats parsing as a model of how humans process sentences. That is certainly an interesting enterprise, but as the non-linguistic examples above demonstrate it is a particular subproblem of the full spectrum of parsing work. In fact, human parsing has various quirks and peculiarities that distinguish it from pretty much every other parsing problem. So we should be careful to distinguish the formal process of *parsing* from the cognitive mechanisms driving *sentence processing*.

As we will see, though, the two have a fair share of overlap, so that results about the former can inform the latter. That is highly welcome; sentence processing is a lot murkier than parsing, and whenever one ventures into murky territory, it is advisable to have some landmarks as points of orientation.

1 Parsing in Computer Science

1.1 Goals and Applications

Computer scientists were interested in parsing from an early date on, with Yngve (1955) usually cited as the first worked-out parsing algorithm. To put this into perspective, research in formal language theory did not take off until Chomsky (1956) and Chomsky and Schützenberger (1963). So parsing research was being done before its formal foundation was even in place yet (as is almost always the case in the history of science).

Quite typical for computer science, the interest in parsing wasn't driven by intellectual curiosity alone but had a strong applied component to it. The early 50s had seen the arrival of high-level programming languages like Autocode, which was soon followed by Fortran and COBOL. This made it possible to write programs in a language that abstracts away from the machine code actually executed by the computer. It is hard to imagine nowadays just how revolutionary high-level programming were at the time. Their most obvious advantage is that they are much easier to understand for humans, which enables them to write and maintain much more complicated programs with less time and effort. But by abstracting away from the hardware executing the code, high-level programming languages also made it possible to write programs that can run on very different hardware. Not all computers are the same — your laptop's x86 architecture shares little with your tablet's ARM-based hardware, yet you can write some Python code that will immediately run on both without any system-specific modifications because Python, just like any other high-level programming language, is deliberately designed to be hardware agnostic. These were undeniable advantages of high-level programming languages, but as so often in life they also created new problems.

While high-level programming languages do away with the need for humans to write code "close to the metal", that does not change the fact that the actual hardware can only understand instructions in its own machine code. Somehow, the human-friendly source code has to be translated into hardware-friendly machine code, which is commonly referred to as *compilation* or *compiling*. Compiling is anything but straightforward as even the most elementary programming concepts like variables pose serious challenges. Consider the following piece of Python code.

```
1  a = 1
2
3
4  def increment_by_two(n):
5      """increment n by 2"""
6      a = 2
7      return n + a
8
9  print(increment_by_two(a))
```

This code involves two assignments of the variable *a*. One occurrence of *a* is initialized as the integer 1, the other one is set to 2 in the definition of the `increment_by_two` function. A careless translation of this program might overwrite the first instantiation of *a* by the second one and would thus compute 2+2 rather than the intended 1+2. The actual translation for Python, on the other hand, is smart enough to recognize that the second variable appears within the scope of a function and thus constitutes a different object that just happens to have the same name. But scope is a structural notion, it

is not readily apparent from the linear order of symbols. For example, if we added another assignment for a immediately after the function, that should overwrite the first occurrence of a. Structural considerations like this are indispensable for a correct compilation procedure from high-level source code to hardware-suitable machine code, and as a result, an efficient method for parsing source code is a prerequisite for compiler design.

Since parsers main purpose in computer science is to facilitate the compilation of source code into machine code, it is of utmost importance that parsers are not ill-behaved. On a practical level, parsers must be efficient — you don't want your OS to crash because the parser swallowed up all your memory while reading in your program, and compiling even large programs with millions of line of code like the Linux kernel should not take unfeasibly long. In addition, a parser should not assign a program an illicit structure or fail to find a structure for a well-formed program. The technical term for this is *correctness*, which is formally defined as the conjunction of soundness and completeness.

Soundness If the parser assigns string s structure σ, then σ is a licit structure for s. In plain English: The parser says only correct things.

Completeness If σ is a licit structure for string s, then the parser assigns σ to s. In plain English: The parsers says all correct things.

Note that soundness and completeness presuppose that we can tell for a given string what its licit structures are. That means we need a definition of the class of well-formed structural descriptions. In other words, we need a grammar.

1.2 Rewrite Grammars

The class of structural descriptions the parser has to operate in is supplied by a *rewrite grammar* $G := \langle N, T, S, R \rangle$, where

- N is a finite, non-empty set of *non-terminal* symbols, and
- T is a finite, non-empty set of *terminal* symbols, and
- $S \in N$ is the designated *start* symbol,
- $R \subset (N \cup T)^* \cdot N \cdot (N \cup T)^* \times (N \cup T)^*$ is a finite set of rewriting rules.

In linguistic parlance: the grammar has lexical items ($= N$), parts of speech ($= T$), a special part of speech ($= S$), and a number of rules for rewriting (the set of which is called R). The rules are usually written in the form $\alpha \rightarrow \beta$ to indicate that α is rewritten as β. The definition above puts no restrictions on α and β except that they are strings of symbols drawn from N and T, and α must contain at least one non-terminal. A grammar generates all those strings that can be obtained from the start symbol S by applying rewrite rules until the output consists only of terminal strings. If string s is generated by grammar G, we also say that G *derives* s. The *language generated by* G is the set L of strings that G derives.

Example 1.1 A Rewrite Grammar in Action

Suppose we have a grammar G with $T := \{a, b, c, d\}$, $N := \{A, B\}$, and start symbol A. Furthermore, R consists of the following rules:

1)	A	→	a		4)	A	→	AA
2)	B	→	b		5)	A	→	Bc
3)	AB	→	a		6)	aB	→	d

This grammar generates an infinite set of strings (why?). Here's three of them and how the grammar derives them:

$$A \xrightarrow{4} AA \xrightarrow{5} ABc \xrightarrow{3} ac$$

$$A \xrightarrow{4} AA \xrightarrow{5} ABc \xrightarrow{1} aBc \xrightarrow{6} dc$$

$$a \xrightarrow{4} AA \xrightarrow{4} AAA \xrightarrow{4} AAAA \xrightarrow{5} ABcAA \xrightarrow{3} acAA \xrightarrow{5} acABc \xrightarrow{1} acaBc \xrightarrow{6} acdc$$

By default, rewrite grammars are not particularly well-behaved on a computational level — for instance, some of them are so complex that one cannot tell for all strings whether they are generated by the grammar. This immediately entails that they do not have efficient parsers, either. After all, the parser has to assign a structure to every string according to the rewrite grammar, and if it could do that for every string, it could also determine for every string whether it is actually generated by the grammar (since an "ungrammatical" string would have an illicit structure).

Proposition 1.1. Parsing is impossible for unrestricted rewrite grammars. □

For this reason, computer scientists prefer a subclass of rewrite grammars known as *context-free grammars* (CFGs). In syntax, those are commonly known as *phrase structure grammars* (PSGs). For CFGs, R is a finite subset of $N \times (N \cup T)^+$, that is to say, only a single non-terminal symbol can appear to the left of a rewrite arrow. That's why these grammars are called context-free: whether a rule can apply to a non-terminal symbol never depends on what appears to the left or the right of the symbol.

Exercise 1.1. The grammar in the previous example has four context-free rewrite rules. What are they? ⊙

Example 1.2 A Context-Free Grammar

Let $G := \langle \{S\}, \{a, b\}, S, R \rangle$, where R contains the following rewrite rules:

$$S \rightarrow aSb$$
$$S \rightarrow ab$$

This grammar generates an infinite set of strings, including *aaabbb*:

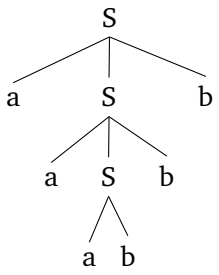
$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb$$

Can you give a description of the language generated by G?

The nice thing about CFGs is that their derivations can be represented in terms of trees. So originally the phrase structure trees used in syntax were actually the derivation trees of CFGs (we will come back to this point much later in the semester during our discussion of Minimalist grammars).

Example 1.3 CFG Derivation Trees

The context-free derivation above corresponds to the tree below:



Exercise 1.2. In linguistics there are many debates about the difference between *representational* and *derivational theories*. Representational theories do not care about how structures are assembled and instead use constraints to separate the well-formed structures from the ill-formed ones. Derivational theories describe how the structures built via certain operations and restrict when and where these operations may apply. Phrase structure trees are usually thought of as representational in nature, they encode the structure of the sentence but not necessarily how it was built (e.g. via Merge and Move in Minimalism). What should we make of this divide given that phrase structure trees were originally derivation trees? ⊙

1.3 Recognizers and Parsers

In many applications it isn't important to know the structure of an input string but just whether it is well-formed or not. A *recognizer* is a formal device (e.g. a piece of software, but we could also build an analog machine) that can tell for every string whether it is generated by a specific grammar G. A *parser* is an extension of a recognizer in that it not only determine for a given string s whether it is generated by G, but if s is generated by G, then the parser also assigns s a structural description according to the grammar. To linguists the distinction may seem pedantic since intuitively there is no way to determine the well-formedness of a sentence without analyzing its structure; when we ask speakers for grammaticality judgments, we ask them whether there is a well-formed structure for the sentence. This is also true of recognizers to some extent as they do reason based on the structural regularities inherent to the supplied grammar, but a recognizer does not necessarily keep track of the structure it builds. Think of it this way: if I ask you whether $1 + 2 + 3 + 4 = 10$, you can confirm this without keeping track whether you first added 1 and 2, or 2 and 3, or maybe something completely different. Similarly, a recognizer only needs to determine whether a string is generated, not how it is generated — that is the parser's job.

As you might have guessed already, it is often very easy to extend a recognizer into a parser by simply keeping track of the steps the recognizer uses to determine

whether a string is in the grammar's language since these steps are closely related to the grammar's rewrite rules. Things get tricky, though, when a string has more than one derivation tree. Should the parser assign all derivations or only one, and if the latter, which one? There's different solutions for this scenario, and we will encounter some later in the semester.

Remark. Strings with multiple derivations are rare in computer science because most parsing is concerned with parsing computer programs, i.e. taking a piece of source code and determining its underlying structure to facilitate the translation of the program into machine code. And programming languages are deliberately designed in such a way that every well-formed string has one unique derivation (although there is sometimes local ambiguity which can only be resolved after reading in a few more symbols). ⊙

2 Syntactic Processing

2.1 The Human Parser and its Quirks

From the perspective of a computer scientist, whatever controls syntactic processing is a very unruly piece of machinery. First of all, it does not seem to be complete. *Garden path sentences*, for example, are well-formed yet so hard to process that most native speakers can't do it without help from a friendly linguist.

(1) a. The horse raced past the barn fell.
    b. The old man the boat.
    c. The government plans to raise taxes were defeated.

Soundness doesn't hold, either, since speakers commonly fall prey to *grammatical illusions*.

(2) a. The key to the cabinets are on the table.
    b. The candidates that no republicans nominated have ever won.
    c. More people have been to Russia than I have.

And excessive memory load is frequent, but seems to depend more on the structure of a sentence rather than its length. In each one of the following pairs, the first sentence is a lot easier than the second even though they have the same length (cf. Gibson 1998; Resnik 1992).

(3) a. The cheese was rotten that the mouse ate that the cat chased.
    b. The cheese that the mouse that the cat chased ate was rotten.

(4) a. The fact that the employee who the manager hired stole office supplies worried the executive.
    b. The executive who the fact that the employee stole office supplies worried hired the manager.

Furthermore, processing effects can vary across languages. For instance, modification of nested phrases is often ambiguous.

(5) I fixed the door of the car with a scratch.

Here the PP *with a dent* can modify *window* or *car*. Some languages like English prefer modification of the structurally more embedded noun *car*, while German and Spanish speakers are more likely to interpret the sentence as *scratch* modifying the higher noun *door*. These two readings are referred to as low and high PP attachment, respectively. So not only is the human parser a rather ill-behaved creature from a computational perspective, it isn't even a uniform creature across languages.

But hold on a second. Soundness and completeness are properties that hold with respect to a specific grammar. That's straight-forward in computer science, where the roles of grammar and parser are well-defined. With natural language, on the other hand, we have no idea what the real thing looks like. So does it make any sense to carry over the computer scientists' strict division between grammar and parser?

2.2 Three Cognitive Oppositions

The distinction between grammar and parser is commonly conflated with the competence-performance dichotomy that was introduced in chapter 1 of *Aspects* (Chomsky 1965).

Competence A speaker's linguistic ability and knowledge, abstracted away from all cognitive, anatomic and physical limitations (e.g. memory limitations, a brain aneurysm, or the finite time span afforded by a universe that's doomed to eventually collapse in on itself)

Performance A speaker's usage of his linguistic knowledge.

Equating competence with grammar and performance with parsing is indeed tempting, seeing how the quirks of the human parser can limit a speaker's ability to understand a sentence in real time even if its structure quickly emerges upon careful inspection. The parser is indeed a factor that one needs to abstract away from if one wishes to describe a speaker's knowledge of language.

But the distinctions are not exactly the same. This is easy to see once one considers the fact that the competence performance distinction can be applied to a parser itself. The competence theory of the parser is a specification of the parser and its behavior under ideal conditions — some kind of algorithm or program, not too different from how one specifies a parser for a programming language. The performance theory, on the other hand, is about how the parser behaves once it is run in the cognitive environment of the human brain. The leading idea of this course is that we can import competence theories of parsing from computer science and turn them into performance theories of parsing via some linking hypothesis, some metric that relates the parser's behavior to processing difficulty.

Depending on how much of a scientific realist you are (personally I'm closer to a constructive empiricist in the vein of van Fraassen 1980), you might wonder what this view implies about language as a cognitive module (cf. Chomsky 1986; Fodor 1983). Are the grammar and the parser two distinct cognitive systems? Should we distinguish the competence parser from the performance parser? The short answer is that this doesn't matter for this course. The rude answer is that this is pointless philosophizing and you're better off spending your time on real research. And then there's a long answer in terms of *Marr's three levels of analysis* (Marr and Poggio 1976).

Marr proposes that any aspect of cognition can be described on three levels of increasing abstraction:

physical the "hardware" instantiation, e.g. neural circuitry for vision

algorithmic what kind of computations does the system perform, what are its data structures and how are they manipulated

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

Example 1.4 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 ($A \cap B$).
- On the algorithmic level, things get trickier. For instance, what kind of data structure do you want to use for the input (sets, lists, arrays?), and just how does one actually construct an object that's the intersection of two sets?
- On the physical level, finally, things are so complicated that it's basically impossible to tell what exactly is being computed by the machine. Voltages increase or decrease in various transistors spread over the CPU, memory and mainboard, and that's about all you can make out. Unless you already have a good idea of the higher levels and the computational process being carried out, it's pretty much hopeless to reverse engineer the program from the electrical signals.

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

What does this mean for language? Instead of thinking of the grammar and the parser as distinct entities, we can view them as different descriptions of the human faculty of language, with the grammar close towards the top in the computational section, while the parser is closer to the algorithmic level (cf. Neeleman and van de Koot 2010). The distinction between competence parser and performance parser would also fit into this paradigm, with the former closer to the computational level but not as close as what we call the grammar.

Exercise 1.3. Can you think of an experiment or case study that would argue against the idea of parser and grammar as one and the same object? ⊙

2.3 Why Processing Matters

Given what we have seen so far, the waters of syntactic processing are indeed very murky. Why, then, should we bother with syntactic processing at all rather than just

focusing on computational parsing models and their usage in real-world applications? As so often in science, the most natural and immediate answer is that syntactic processing is an interesting and fun puzzle. And just like other aspects of human psychology studying it comes with the satisfaction of feeding mankind's innate vice of anthropocentric narcissism. But there are more tangible advantages.

For those that mostly care about the engineering aspects of language technology, the human parser is both a challenge and an opportunity. It is a challenge because the limitations of the human parser are limitations of what is comprehensible to humans. A natural language system with perfect command of English will be considered broken by its user's if it expresses itself in garden path sentences and with multiple levels of center embedding. These constructions might not be particularly hard for a formal parsing model, but they nonetheless must be avoided in interactions with humans. Studying syntactic processing is also an opportunity because history has shown that many a great engineering insight can be gleamed from nature, and the human parser is an impressive machine — warts and quirks notwithstanding. The parser is incredibly robust for most sentences encountered in the wild, and it operates much faster than any known parsing algorithm. It is essentially real-time, which means that if the input is presented to the parser in an ongoing stream, the parser finishes within some fixed constant c of steps after reading in the last symbol. Even with very strong restrictions on the grammar no state-of-the-art parsing model can match this speed.

To linguists, syntactic processing holds the promise of resolving issues that cannot be decided based on grammaticality judgments. This is in fact a very old idea: the *Derivational Theory of Complexity* (DTC; Miller and Chomsky 1963; Miller and McKean 1964) posits that the more grammatical operations are required to build a sentence, the harder it is to process. The DTC quickly fell out of favor for empirical (Slobin 1966) and conceptual reasons (cf. Garnham 1983) but has seen a revival in recent years. Colin (1996:Ch.5) argues convincingly that the experimental results may discredit a specific version of the DTC rooted in early Transformational Grammar but do not challenge the assumption that grammatical complexity affects processing difficulty. Hale (2011) gives an outline of what a computationally grounded reinterpretation could look like, and recent applications for Minimalist syntax are developed in Kobele et al. (2012); Graf and Marcinek (2014); Graf et al. (2015). If this revival of the DTC turns out to be more successful, processing difficulties will offer a window into the mechanisms of the grammar and thus allow syntacticians to distinguish between competing analyses.

Syntactic processing may also be essential in explaining certain typological universals, in particular why certain patterns are never instantiated cross-linguistically — that is to say, why there are typological gaps. At this point, no grammar formalism can account for all typological facts purely in terms of *formal universals*, i.e. restrictions on its operations and mechanisms. They all have to invoke *substantive universals* such as a fixed set of categories, feature geometries and a very elaborate system of functional projections. Substantive universals are much less attractive than formal universals because they are hard to unify or reduce to more basic mechanisms. However, if it could be shown that some typological gaps exist because the corresponding patterns are much harder to process than the attested ones, then we would have a direct explanation of this typological facts in terms of processing. In other words, some typological universals no longer need to be accounted for in the grammar and can instead be factored out into a processing-based account.

References and Further Reading

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2:113–124.

Chomsky, Noam. 1965. *Aspects of the theory of syntax*. Cambridge, Mass.: MIT Press.

Chomsky, Noam. 1986. *Knowledge of language: Its nature, origin, and use*. New York: Praeger.

Chomsky, Noam, and M. P. Schützenberger. 1963. The algebraic theory of context-free languages. In *Computer programming and formal systems*, ed. P. Braffort and D. Hirschberg, Studies in Logic and the Foundations of Mathematics, 118–161. Amsterdam: North-Holland.

Colin, Phillips. 1996. *Order and structure*. Doctoral Dissertation, MIT.

Fodor, Jerry. 1983. *The modularity of mind*. Cambridge, Mass.: MIT Press.

van Fraassen, Bas. 1980. *The scientific image*. Oxford: Oxford University Press.

Garnham, Alan. 1983. Why psycholinguists don't care about DTC: A reply to Berwick and Weinberg. *Cognition* 15:263–269.

Gibson, Edward. 1998. Linguistic complexity: Locality of syntactic dependencies. *Cognition* 68:1–76.

Graf, Thomas, Brigitta Fodor, James Monette, Gianpaul Rachiele, Aunika Warren, and Chong Zhang. 2015. A refined notion of memory usage for minimalist parsing. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, 1–14. Chicago, USA: Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W15-2301.

Graf, Thomas, and Bradley Marcinek. 2014. Evaluating evaluation metrics for minimalist parsing. In *Proceedings of the 2014 ACL Workshop on Cognitive Modeling and Computational Linguistics*, 28–36.

Hale, John. 2011. What a rational parser would do. *Cognitive Science* 35:399–443.

Kobele, Gregory M., Sabrina Gerth, and John T. Hale. 2012. Memory resource allocation in top-down minimalist parsing. In *Proceedings of Formal Grammar 2012*.

Marr, David, and Tomaso Poggio. 1976. From understanding computation to understanding neural circuitry. Technical report, Artifical Intelligence Laboratory, MIT, AIM-357.

Miller, George A., and Noam Chomsky. 1963. Finitary models of language users. In *Handbook of mathematical psychology*, ed. R. Luce, R. Bush, and E. Galanter, volume 2. New York: John Wiley.

Miller, George A., and Kathryn Ojemann McKean. 1964. A chronometric study of some relations between sentences. *Quarterly Journal of Experimental Psychology* 16:297–308.

Neeleman, Ad, and Hans van de Koot. 2010. Theoretical validity and psychological reality of grammatical code. In *The linguistic enterprise: From knowledge of language to knowledge of linguistics*, ed. Martin Everaert, Tom Lentz, Hannah de Mulder, Oystein Nilsen, and Arjen Zondervan, 183–212. Amsterdam: John Benjamins.

Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of COLING-92*, 191–197.

Slobin, Dan. 1966. Grammatical transformations and sentence comprehension in childhood and adulthood. *Journal of Verbal Learning and Verbal Behavior* 5:219–227.

Yngve, Victor H. 1955. Syntax and the problem of multiple meaning. In *Machine translation of languages*, ed. William N. Locke and A. Donald Booth, 208–226. Cambrdige, MA: MIT Press.