Lecture 1

Big Picture

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. Syntactic processing refers to the act by which a speaker assigns a natural language sentence (conceptualized as a string of words, rather than, say, a continuous signal of sound waves) some syntactic structure. However, it is not always clear that the structures assigned during syntactic processing are actually well-formed with respect to a native speaker's grammar. So syntactic processing is not exactly a standard case of parsing. It is a lot murkier than parsing, and whenever one ventures into murky territory, it's prudent to make out some landmarks as points of orientation.

1.1 Parsing in Computer Science

1.1.1 Rewrite Grammars

In computer science, things are straight-forward. The formal specification the parser has to adherer to 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 $\alpha \to \beta$ 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:

$$\begin{array}{rr} A \rightarrow a & A \rightarrow AA \\ B \rightarrow b & A \rightarrow Bc \\ AB \rightarrow a & aB \rightarrow d \end{array}$$

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

$$A \rightarrow AA \rightarrow ABc \rightarrow ac$$

$$A \rightarrow AA \rightarrow ABc \rightarrow aBc \rightarrow dc$$

$$a \rightarrow AA \rightarrow AAA \rightarrow AAAA \rightarrow ABcAA \rightarrow acAA \rightarrow acABc \rightarrow acaBc \rightarrow acdc$$

By default, rewriting 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 s 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 s, it could also determine for every s whether it is actually generated by the grammar (since an "ungrammatical" string would have an illicit structure).

Instead, computer scientists commonly use a subclass of rewrite grammar known as *context-free grammars* (CFGs). In syntax, those are commonly known as *phrase structure grammars*. 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. 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 $aaabbb$:

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

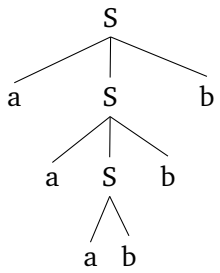
Can you give a description of the language generated by G?

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).

Exercise. 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 are actually derivation trees? ◯

Example 1.3 CFG Derivation Trees

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



1.2 Recognizers and Parsers

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. In general, recognizers are designed in a modular fashion such that they are not restricted to a single grammar but instead work for an entire class of grammars. So a recognizer for CFGs takes a CFG G as a parameter and then determines for a given string whether it is generated by G (strictly speaking, we should call such a parameterized recognizer a meta-recognizer since it is actually a program that computes a recognizer for the supplied grammar).

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. In computer science, this usually means assigning the string its derivation tree. Often it is 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. In fact, parsing often boils down to applying the grammar's rewrite rules in a very smart fashion.

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 handling this scenario, and we will encounter some later in the semester.

Remark. This scenario rarely occurs 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 a way such that every well-formed string has one unique derivation. ◯

Computer scientists require their parsers to be sound and complete.

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

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

Both properties are clearly desirable in a parser. After all, what's the point of meticulously writing a grammar if your parser messes things up later down the road? And the parser should also be efficient — you don't want your OS to crash because the parser swallowed up all your memory while reading in your program. Unfortunately, the human parser didn't quite get the memo.

1.3 Syntactic Processing

1.3.1 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.

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?

1.4 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 (Chomsky 1986; Fodor 1983:cf.). 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 as the grammar and the parser as distinct entities, we can think of 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. 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? ◯

References and Further Reading

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.

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

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

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

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

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.