Lin630
Parsing and Syntactic Processing

Thomas Graf

Contents

Lecture 0

Syllabus

Course: Parsing and Syntactic Processing
Course#: Lin630 (officially still Lin651)
Time: tbd
Location: CompLab SBS N250
Course Website: lin630.thomasgraf.net

Name: Thomas Graf
Email: lin630@thomasgraf.net
Office hours: tba
Office: SBS N249
Personal Website: thomasgraf.net

1 Overview

- **Big Questions**
  - What is the relation between competence and performance, grammar and parser?
  - Are syntactic processing effects conditioned by the grammar?
  - What qualifies as a parser as opposed to a recognizer or a parsing schema?
  - Can we use insights from syntactic processing research to speed up current parsing technology?

  The first two are common questions for any processing course. The third and fourth one hint at the special twist of this course: we approach these issues from a computational perspective! Parsing theory is a big (albeit messy) area of computer science, there's tons of parsing models on the market. So let's bring all these insights to bear on how humans parse natural language.

- **Teaching Goals**
  At the end of this course you will
  - be familiar with a variety of common parsing models (top-down, bottom-up, left-corner, Earley, CYK)
  - know the most common syntactic processing effects (in particular those related to memory usage)
  - be able evaluate claims in the psycholinguistic literature from a computational perspective

- **Prerequisites**
  None beyond basic syntax skills — you should be able to draw a reasonable

tree for a sentence like *The fact that the employee who the manager hired stole office supplies did not go unnoticed by the janitor*. Knowledge of theoretical computational linguistics (e.g. as covered in Lin637) is helpful, but not necessary.

2 Course Requirements

- **Homeworks**
  There will be weekly homeworks, the solutions of which will be discussed during review sessions run by students retaking this course. Homeworks are essential if you want to learn anything in this course — you don't truly understand a parsing algorithm until you can carry it out yourself.

- **Review Paper**
  Write a critical review of a particular topic in parsing or syntactic processing that will be shared with and commented on by your class mates. You have to get my approval for your topic and your list of readings by the end of week 8 (see the website for topic suggestions). Your paper should be about 15 pages (letter paper, 1in margins, double spaced, 12pt), must be written in LaTeX, and has to be completed by the end of week 12. After you have gotten feedback from your colleagues, you must hand in a revised version by the end of finals week.

  This assignment serves two purposes: you get experience writing review articles, and your colleagues get to learn about topics that weren't covered in class.

- **Paper Evaluations**
  For each review paper, you have to write a 1-page evaluation (it can be longer if necessary, but do not go beyond 3 pages). Ideally, you should engage with the paper on an intellectual level, which means critiquing parts you disagree with, suggesting follow-up readings, or asking questions that were left unanswered. But presentational issues such as grammar, style, or overall structure can also be brought up.

  Think of this as you peer-reviewing a squib for a journal like *Language and Linguistics Compass*, which is something you will have to do quite frequently as an academic. It also hones your skills for giving detailed but polite and structured feedback in a corporate environment, e.g. as part of writing and maintaining documentation for a specialized piece of software.

- **Workload per Credits**
  - *1 credit*: regular attendance, commenting on all review papers
  - *2 credits*: the above, plus doing all the homeworks
  - *3 credits*: the above, plus writing a review paper

  Students who are retaking this course for credit can instead volunteer to run some of the review sessions.

3 Outline

This outline assumes that we meet twice a week for 90 minutes with a 10 minute break, plus a weekly recitation of 60 minutes. After week 9, we meet once a week for 60 minutes to brainstorm ideas for connecting parsing to protein folding.

Wk	Chap	Topic	Assignments
1	1,2	Parsing across disciplines, modular view of parsing	
2	3,4	Top-down parsing	
3	5,6	Bottom-up parsing, chart parsing intro	
4	6	CKY and Earley	
5	7	(Generalized) Left-corner parsing	
6	8	Semiring parsing	
7	9	Moving beyond context-free grammars	
8	10,11	CFG-parsing of mildly-context sensitive formalisms	get topic approved
9	12,13	LR parsing, partial parsing	
10		protein folding	
11		protein folding	
12		protein folding	review paper draft
13		protein folding	
14		protein folding	
15		protein folding	paper evaluations
finals1		Q&A session	
finals2			paper due

The whole course has a time investment of 3 credits for 15 weeks, which totals $3 * 53 * 15 = 2385$ minutes. The distribution is shown below.

type	number	minutes	total
lecture	18	80	1440
recitation	9	60	540
research meeting	6	60	360
Q&A session	1		45
			2385

4 Policies

4.1 Contacting me

- Emails should be sent to lin630@thomasgraf.net to make sure they go to my high priority inbox. Disregarding this policy means late replies and is a sure-fire way to get on my bad side.
- Reply time $<$ 24h in simple cases, possibly more if meddling with bureaucracy is involved.
- If you want to come to my office hours and anticipate a longer meeting, please email me so that we can set aside enough time and avoid collisions with other students.

4.2 Special Needs

If you have any special needs that might impact your class performance (learning disabilities, impaired sight or hearing, etc.), please come to my office hours or contact me via mail so we can make suitable arrangements.

5 Online Component

Rather than Blackboard, I use github to distribute lecture notes and readings (yes, you have to print them yourself). The readings are in a private repository, so you need a github account to access them. If you don't have one already, you can create one for free (github does not collect any user data). Make sure to email me your username asap so that I can give you access to the repository. If you do not want to use github for some reason, you can drop by my office to make an offline copy of the readings.

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*:

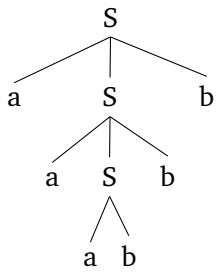
$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb$$

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 are actually derivation trees? ⊙

1.3 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 andling 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. ⊙

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.

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 (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 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? ⊙

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.

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.

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.

Lecture 2

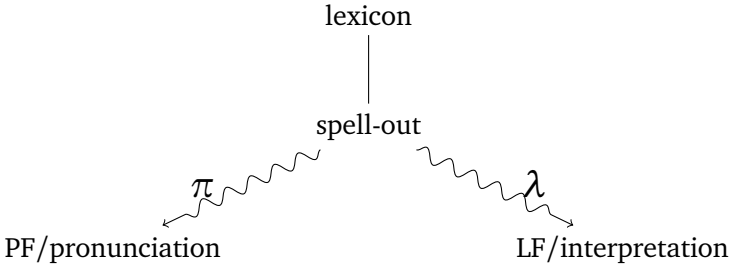
A Modular View of Parsing

Last time we saw that Marr's three levels of analysis are a useful guideline with respect to how one might think about the distinction between grammar and parser in linguistics. We also extended this idea idea that one and the same device can be described at distinct levels of abstraction to the parser itself, differentiating between an idealized competence parser — the full specification of the human parser — and the performance parser, i.e. how the parser actually behaves when executed by the neural hardware of the human brain. But even if we disregard cognition for a moment and think about parsing in purely computational terms, it quickly becomes evident that there's many different ways of defining a parser, some more specific than others. In computer science, this is mostly a matter of convenience and generality, where abstraction pays off for studying the mathematics of parsing whereas the actual implementation in some programming language requires a lot more detail.

For this course, however, abstraction is a matter of explanation: eventually, we want to relate parsing techniques to human sentence processing. If our description of the parser abstracts away from, say, how intermediate information is stored and retrieved, we're making the empirical assumption that these aspects of the parser are irrelevant for the psycholinguistic phenomena we observe. So before we even start any kind of empirically minded work, we have to get a better understanding of the different levels of abstraction, and in particular, what exactly we are abstracting away from.

1 The Most General View of Parsing

One of the cornerstone's of transformational grammar is the inverted T-model, a metaphor for the place of syntax in the language faculty.



The standard interpretation of this diagram is procedural in nature:

1. syntax builds structures from the items in the lexicon (or the numeration in earlier versions of Minimalist syntax),
2. at some point spell-out takes place and creates two copies of the built tree (Chomsky 1995:229),
3. one copy is turned into a pronounceable structure by a variety of processes summarily referred to as π,
4. the other copy is turned into an interpretable structure by processes jointly referred to as λ.

There is, however, a more parsimonious interpretation of this model:

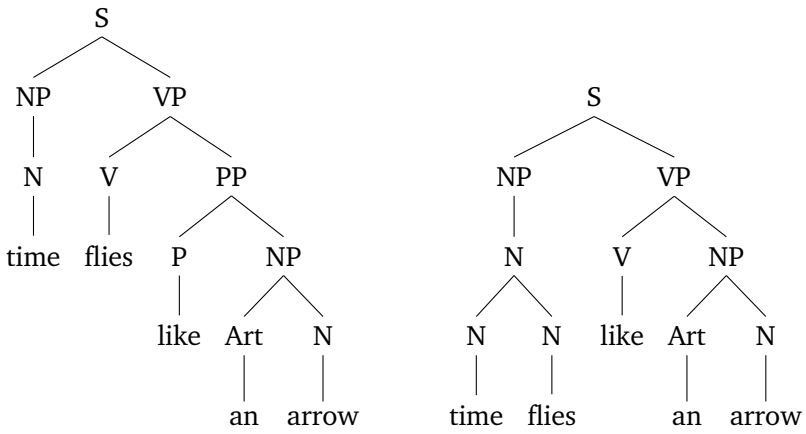
- the grammar specifies a set of well-formed representations (e.g. phrase structure trees),
- we can define various functions for mapping these representations to other structures; in the case at hand, a function π from trees to strings and a function λ from trees to semantic interpretations

Notice that this more agnostic perspective generalizes the T-model from natural language to all kinds of rule-based systems. Just consider programming: a program is a highly structure object that has a linear realization in the form of its source code (its image under π) and a realization in terms of the instructions carried out by the computer it is run on (its image under λ). Why does this matter? Because it gives us the most general and abstract description of a parser.

Parsing as inverted pronunciation Given a grammar G with mapping π from trees to strings, a parser for G is a device that computes π^{-1} (the inverse of π).

Example 2.1 The π^{-1} parser

Suppose that our grammar has exactly two trees s and t whose string yield is *Time flies like an arrow*.



Like any other function, π can be viewed as a set of pairs $\langle a, b\rangle$, which means that a is mapped to b by π. So for our example, π is some set that contains at least $\langle s, \text{time flies like an arrow}\rangle$ and $\langle t, \text{time flies like an arrow}\rangle$. The inverse of π is simply the set that contains the pair $\langle b, a\rangle$ iff $\langle a, b\rangle$ is a member of π. Hence π^{-1} contains both $\langle \text{time flies like an arrow}, s\rangle$ and $\langle \text{time flies like an arrow}, t\rangle$. Given our interpretation of these pairs, this just means that this string can be mapped to s and t. And from this it follows that any device that correctly computes π^{-1} picks out s and t as the only licit trees for *time flies like an arrow* — which is exactly what we want a parser to do.

Exercise 2.1. If π^{-1} is parsing, what is λ^{-1}? ⊙

2 One Step Down: Parsing as Grammar Intersection

The obvious problem with the previous view of parsing is that it tells us nothing about how the actual process is accomplished: if we don't know how to compute π^{-1}, there is little we can do. Now for some choices of π there are some very general algorithms for computing π^{-1} (for example if π can be defined in terms of monadic second-order logic, the extension of first-order logic with quantification over sets). But for the purpose of relating parsing to human sentence processing, this perspective is still too general. It can provide profound insights into the overall computational difficulty of the parsing problem, but it is too coarse to have any bearing on specific processing phenomena or assumptions about the human parser.

A slightly more concrete view is offered by *intersection parsing*. Intersection parsing builds on the insight that all common grammar formalisms are closed under intersection with finite languages.[1] That is to say, if G is a grammar of formalism $\mathscr{F}$, and one takes the intersection of the language L generated by G and some arbitrary finite language L_F, then this language $L(G) \cap L_F$ can be generated by some grammar G' belonging to formalism $\mathscr{F}$. Crucially, the proofs for this theorem are constructive in nature, which means that they actually provide a procedure for constructing G' from G and L_F.

This opens up the following strategy:

1. Suppose i is our input sentence that needs to be parsed with respect to grammar G. Let $L := \{i\}$ be the language whose only well-formed string is i.
2. Construct the grammar G' that generates $L(G) \cap L$.
3. Use G' to infer the valid trees for i.

But how exactly does one handle the third step? Isn't that exactly the parsing problem we started out with, except that we're now operating with a different grammar?

As so often, the answer is both yes and no. First, let's make two assumptions about our initial grammar G.

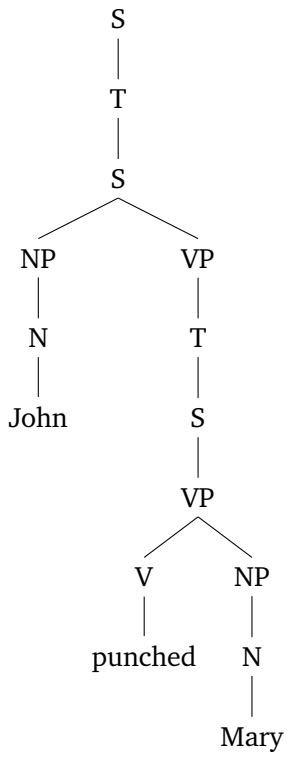
No loops The grammar does not generate any trees with a sequence of unary branches where some non-terminal symbol occurs more than once.

[1] The closure property actually extends to all regular string languages. However, the class of regular languages includes all finite languages.

No empty heads The empty string ε is not a valid terminal symbol.

Example 2.2 A tree with two loops

The tree below contains two loops.



One can prove that this two assumptions jointly imply the property below.

Bounded tree size A grammar G has the *bounded tree size property* iff it holds for every sentence s generated by G that the size of each tree that G assigns to s is finitely bounded by the length of s.

The intuitive reasoning is as follows: In order to increase the size of a tree, one has to apply a rewrite rule. Every rewrite rule increases the number of symbols that are expanded into one or more symbols. If we can't have loops, that puts an upper bound on the length of unary branches. If we can't have empty heads, then every symbol that is rewritten by at least two symbols will eventually bring about the addition of at least two terminal symbols to the generated string. So the length of the string limits how many rewrite rules can be applied, which in turn bounds the size of the trees that can be assigned to the string.

The bounded tree size property, in turn, guarantees that G' in step 3 above assigns only finitely many trees to the input sentence.

Lemma 2.1. If grammar G has the bounded tree size property, it assigns only finitely many trees to every string in the language it generates. □

This result may seem obvious, but it is the very reason why it is dead easy to use G' to infer the valid trees for i: the only trees well-formed with respect to G' are those that G assigns to i. That is to say, in order to determine the licit trees for i we just have list the trees generated by G'. And since the latter by assumption has the bounded tree size property, there's only finitely many trees that G' generates. We can find them all by simply applying all the rewrite rules of G' in all possible ways (this step will finish after a finite amount of time since there's only a finite number of trees that need to be generated).

Parsing as intersection Suppose i is our input sentence that needs to be parsed with respect to grammar G. Let $L := \{i\}$ be the language whose only well-formed string is i.

- Construct the grammar G' that generates $L(G) \cap L$.
- Apply all rewrite rules of G' in all possible ways to obtain the set of trees generated by G'. This is also the set of trees that G assigns to i.

Exercise 2.2. As linguists we are very fond of our empty heads, so the requirement that the empty string may not appear on the right hand side of any rule seems overly strong. Can you think of a relaxed version of the ban against empty heads that is compatible with linguistic practice but also preserves the bounded tree size property? ⊙

Exercise 2.3. Would the procedure above also work for grammars that do not enjoy the bounded tree size property? What exactly would we lose? ⊙

While intersection parsing is very elegant, it still abstracts away from too many issues that are relevant to us. In particular, the *incrementality of parsing* no longer plays a role. Syntactic processing operates in an incremental fashion; the parser doesn't wait for the entire sentence to be uttered before it starts working, it kicks off as soon as the first word has been heard. The parsing as intersection approach, on the other hand, needs to know the entire input sentence in order to construct the grammar that will be used for inferring the tree structures. Since pretty much all interesting phenomena in syntactic processing are related to incrementality in some form or another, intersection parsing is also too coarse for our purposes. We have to get closer to the algorithmic level of description.

3 Towards the Algorithmic Level

3.1 The Three Modules of a Parser

Modern parsing theory treats parsers as the combination of three distinct modules.

Parsing schema A rule system that specifies

- the general form of *parsing items*,
- the possible initial items,
- the desired final items,
- a finite number of *inference rules* for constructing new items from old ones

Control structure A system for choosing

- which inference rules to apply at any given point during the parse, and
- in which order they should be applied.

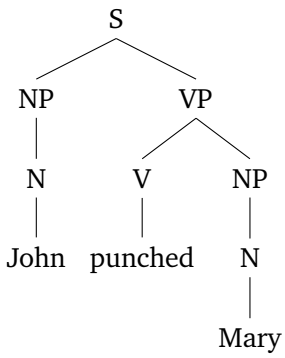
Data structure A system for storing and retrieving parsing items.

Example 2.3 A naive parser for CFGs

Suppose we want to parse the sentence *John punched Mary* with the CFG given below.

S	→ NP VP	N	→ John
NP	→ N	N	→ Mary
VP	→ V NP	V	→ punched

Obviously there is only one valid tree for this sentence.



But how do we get this tree? In your syntax class, you might have learned to draw a table like the one below.

item	**rule**
S	start symbol
NP VP	S → NP VP
N VP	NP → N
John VP	N → John
John V NP	VP → V NP
John punched NP	V → punched
John punched N	NP → N
John punched Mary	N → Mary

What we have done is to identify an initial item, S, followed by a list of the rules that we used to rewrite items. Note that the order we applied the rules in is mostly irrelevant.

item	**rule**
S	start symbol
NP VP	S → NP VP
NP V NP	VP → V NP
N V NP	NP → N
N V N	NP → N
John V N	N → John
John V Mary	N → Mary
John punched Mary	V → punched

The rewrite rules act as our parsing schema, they tell us how to start and where to go from there. A control structure would also tell us in which order we have to apply the rules, e.g. by forcing us to always rewrite the left-most non-terminal in the current item, as we did in the first table. As a data structure, we may use a table like the one above, but if there's multiple derivations for a sentence we will need something more sophisticated to keep track of all of them.

While the parsing literature is huge, parsers differ only in a small number of ways when it comes to the parsing schema and the control structure. The culprit for the large number of competing parsing models is the similarly large number of data structures. Optimizing data structures can have a huge effect on a parser's speed and memory efficiency, so it's no wonder that people keep tinkering with them. But this is mostly a matter of algorithm design with little relevance for the empirical phenomena we'll be looking at. For this reason we will mostly ignore data structures throughout the course and focus on parsing schemata and control structures instead. Fortunately this will also make the technical parts a lot less cumbersome (compared to, say, the treatment in the otherwise excellent Grune and Jacobs 2008).

3.2 How Parsers Differ

Modulo data structures, parsers differ only in a handful of ways.

- **Parsing Schema Parameters**
  - orientation
    - top-down ("from S to NP VP")
    - bottom-up ("from NP VP to S")
    - mixture of the two
  - underlying grammar formalism
    - CFGs,
    - TAGs,
    - Minimalist grammars
- **Control Structure Parameters**
  - directionality
    - directional: parse input strictly left-to-right or right-to-left (= incremental)
    - non-directional: input can be read in arbitrary order (= non-incremental)
  - search method
    - depth-first: build a full branch until you encounter a terminal, then move on to next branch
    - breadth-first: build the tree in "layers" like a house of cards
  - rule selection
    - exhaustive: apply all inference rules
    - probabilistic: apply most likely rules

4 A Few Remarks on Parser Performance

As mentioned last time, parsers need to be efficient. In computer science this is due to practical considerations (parsing a program with thousands of lines of code), whereas in psycholinguistics it is an empirical fact — the human parser is extremely fast. Nonetheless efficiency will play a rather small role in this course, mostly because it is very unclear how to relate the results from computer science to the empirical task.

Computer scientists measure a parser's performance in terms of *asymptotic worst-case complexity*. That is to say, how long does it take to parse a sentence if everything goes wrong that could go wrong within the specified parameters of the problem (e.g. high structural complexity of the input sentence) if there are no restrictions on the length of the sentence.

Asymptotic worst-case complexity is expressed via the "Big O" notation. If a parser has time complexity $O(n^3)$, this means that it takes at most n^3 steps for the parser to finish, where n is the length of the sentence measured in words. The Big O notation ignores parameters whose impact on complexity diminishes as the length of sentences approaches infinity. The size $|G|$ of some grammar G, for instance, has a huge effect on parsing performance for short sentences, but for very long sentences the combinatorial explosion is such a major challenge for the parser that the impact of grammar size is minuscule in comparison. Consequently it holds that $O(|G|n^3) = O(n^3)$.

Without going into too much detail, we can rank parser efficiency according to which of the following classes they belong to:

Real-time The sentence is parsed as fast as it is read in, up to some constant c (which is factored out by the "Big O" notation).

Linear Parsing time is linearly bounded by sentence length, e.g. $O(2n + 5) = O(n)$

Squared $O(n^2)$

Cubic $O(n^3)$

Polynomial $O(n^i)$, for some $i \geq 1$

Exponential $O(i^n)$, for some $i \geq 1$

The best known parsers for CFGs run in cubic time, which seems much worse than humans' real-time processing performance. There is no proof that there aren't any faster CFG parsers, but after 50 years of research it seems unlikely that real-time CFG parsers do exist but have not been discovered yet. But one has to be careful:

- **Notions of complexity**
  The cubic time result for CFGs is about asymptotic worst-case complexity, whereas claims about the human parser are necessarily about average case complexity due to said parser's nasty habit to simply crash when things get difficult.

- **Generality of parser**
  The cubic time result is for parsers that work for very large classes of CFGs. If a parser can be optimized for a specific CFG, it can run much faster. The fewer grammars a parser needs to be sound and complete for, the more shortcuts and speedhacks can be employed. Maybe natural language grammars are just very specific objects that allow for tons of optimization (island effects anyone?).

- **Determinism**
  The cubic time result is due to the massive non-determinism CFG parsers have to deal with. One sentence can have hundreds of distinct derivations. The same is also true of natural language grammars, but semantics and discourse completely disambiguate sentences in the majority of cases. This makes natural language processing mostly a deterministic process, and parsing deterministic CFGs is also very fast.

References and Further Reading

Chomsky, Noam. 1995. *The minimalist program*. Cambridge, Mass.: MIT Press.

Grune, Dick, and Ceriel J.H. Jacobs. 2008. *Parsing techniques. A practical guide*. New York: Springer, second edition.

Lecture 3

Top-Down Parsing

Top-down parsing is arguably the simplest parsing model. In fact, it is so intuitive that syntax students, for example, have a tendency to automatically use this algorithm when asked to determine if a grammar generates a given sentence. That's because top-down parsing is very close to the idea of phrase structure grammars as top-down generators.

1 Intuition

Suppose you are given the following CFG.

1)	S	→ NP VP	6)	Det	→ a | the
2)	NP	→ PN	7)	N	→ car | truck | anvil
3)	NP	→ Det N	8)	PN	→ Bugs | Daffy
4)	VP	→ Vi	9)	Vi	→ fell over
5)	VP	→ Vt NP	10)	Vt	→ hit

When asked to show that this grammar generates the sentence *The anvil hit Daffy*, you might draw a tree. But a different method is to provide a tabular depiction of the rewrite process.

string	**rule**
S	start
NP VP	S → NP VP
Det N VP	NP → Det N
the N VP	Det → the
the anvil VP	N → anvil
the anvil Vt NP	VP → Vt NP
the anvil hit NP	Vt → hit
the anvil hit PN	NP → PN
the anvil hit Daffy	PN →Daffy

Of course the rewrite rules could also be applied in other orders.

string	rule
S	start
NP VP	S → NP VP
NP Vt NP	VP → Vt NP
NP Vt PN	NP → PN
NP Vt Daffy	PN →Daffy
NP hit Daffy	Vt → hit
Det N hit Daffy	NP → Det N
Det anvil hit Daffy	N → anvil
the anvil hit Daffy	Det → the

string	rule
S	start
NP VP	S → NP VP
Det N VP	NP → Det N
Det N Vt NP	VP → Vt NP
the N Vt NP	Det → the
the anvil Vt NP	N → anvil
the anvil hit NP	Vt → hit
the anvil hit PN	NP → PN
the anvil hit Daffy	PN →Daffy

In all three cases we proceed top-down: non-terminals are replaced by a string of terminals and/or non-terminals. From the perspective of phrase structure trees, the trees are growing from the root towards the leafs. The difference between the three tables is the order in which non-terminals are rewritten.

- Table 1: depth-first, left-to-right
- Table 2: depth-first, right-to-left
- Table 3: breadth-first, left-to-right

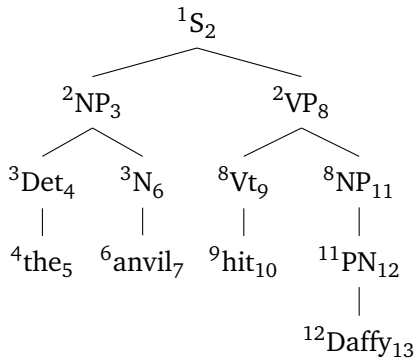
depth-first rewrite some symbol that was introduced during the previous rewriting step

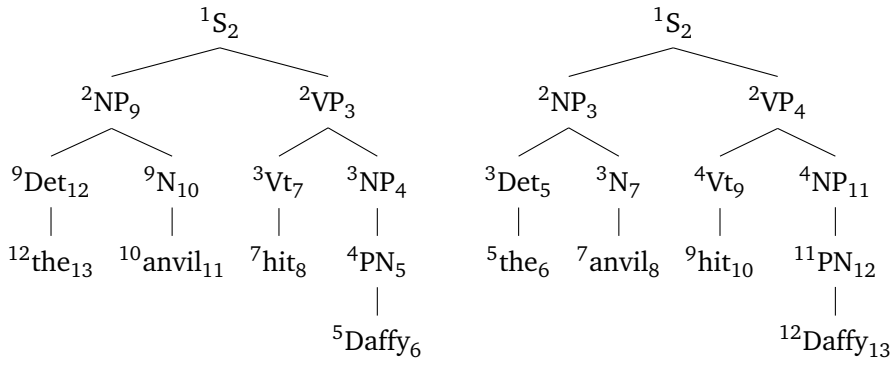
breadth-first before rewriting a symbol introduced during rewriting step j, all symbols that were introduced at rewriting step i must have been rewritten, for every $i < j$

left-to-right if several symbols are eligible to be rewritten, rewrite the leftmost one

right-to-left if several symbols are eligible to be rewritten, rewrite the rightmost one

We can visualize the differences between these strategies by annotating phrase structure trees with indices to indicate when a symbol is first introduced (prefix) and when it is rewritten (suffix). For terminal symbols, which are never rewritten, we stipulate that the suffix is one higher than the prefix.





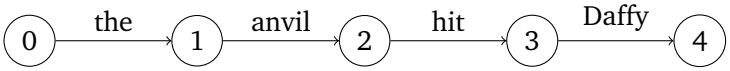
Exercise 3.1. Draw the table for a breadth-first, right-to-left parser, and the corresponding annotated phrase structure tree. ⊙

Exercise 3.2. Consider the sentence *Bugs fell over*. Draw the tables for a left-to-right depth-first strategy and left-to-right breadth-first strategy, and draw the corresponding annotated phrase structure trees. Do the two strategies differ at all over such a short sentence? If so, how, and at which nodes? ⊙

2 Formal Specification

2.1 Sentences as Indexed Strings

The input to the parser is just a sequence of words, i.e. a string. We adopt the standard convention for numbering positions in a string, i.e. the i-th word in the string occurs between positions $i-1$ and i.



Given a string w that consists of n words, each word can be referenced by the number to its left. If $w = \textit{the anvil hit Daffy}$, $w_0 = \textit{the}$ and $w_3 = \textit{Daffy}$, for instance. Notice that if the grammar allows for empty heads, an input sentence can be mapped to infinitely many such strings depending on where one posits empty heads.

2.2 Parsing Schema

The parsing schema can be easily stated in terms of logical inference rules. This idea was originally called *parsing as deduction* (Pereira and Warren 1983; Shieber et al. 1995) and was later refined by Sikkel (1997). Parsing as deduction makes it very easy to see what distinguishes different parsing schema and put various control structures on top of them. It's also the foundation for *semiring parsing* (Goodman 1999), which provides an abstract perspective that unifies recognizers and parsers, among other things.

Given a CFG $G := \langle N, T, S, R\rangle$ and input string $w = w_1 \cdots w_n$, our *items* take the form $[i, \beta, j]$, where

- $\beta \in (N \cup T)^*$ is a string of terminal and/or non-terminal symbols, and
- i and j are positions in the string such that $0 \leq i < j \leq n+1$.

An item encodes the conjecture that the substring spanning from i to j can be generated from β via G. Our only *initial item* (also called *axiom*) is $[0, S, n]$, which denotes that the grammar generates the string spanning from 0 to n. As our *final item* (also called *goal*) we require $[n, , n]$.

The parser uses two inference rules; one for scanning, the other one for top-down prediction.

$$\textbf{Scan} \qquad \frac{[i, a\beta, j]}{[i+1, \beta, j]}\ a = w_i$$

$$\textbf{Predict} \qquad \frac{[i, \alpha N \beta, j]}{[i, \alpha\gamma\beta, j]}\ N \rightarrow \gamma \in R$$

The scanning rule tells us that the starting position of an item can be moved one to the right if the item starts with a terminal symbol that matches the word in the input string at the corresponding position. The predict rule produces a new item from an old by rewriting one the one non-terminal symbols using a production of G.

Example 3.1 Top-down parse of *The anvil hit Daffy*

Let G be the grammar at the beginning of the handout. Then a depth-first, left-to-right parse of *The anvil hit Daffy* will proceed as follows, where predict(n) denotes a prediction step using rule n.

parse item	**inference rule**
[0,S,4]	axiom
[0,NP VP,4]	predict(1)
[0,Det N VP,4]	predict(3)
[0,the N VP,4]	predict(6)
[1,N VP,4]	scan
[1,truck VP,4]	predict(7)
[2,VP,4]	scan
[2,Vt NP,4]	predict(5)
[2,hit NP,4]	predict(10)
[3,NP,4]	scan
[3,PN,4]	predict(2)
[3,Daffy,4]	predict(8)
[4,,4]	scan

Note that the table above only shows the prediction steps leading to a successful parse. The parser, on the other hand, applies every possible prediction step. So from [0,NP VP,4], for instance, the parser not only predicts [0,Det N VP,4] via rule 3 but also [0,PN VP,4] via rule 2 since both can be applied to NP.

As was discussed last time, the parsing schema only describes the starting point of the parser, the desired outcome, and which inference steps the parser may take to get from the former to the latter. It does not specify in which order these steps should be taken, nor how the parser handles multiple parses in parallel. This is left to the control structure and the data structures.

2.3 Control Structure

We can add a control structure to the parser to describe both the parsers directionality (left-to-right VS right-to-left) and its search method (depth-first VS breadth-first). There's many ways of specifying the control structure, but one simple option is to encode it in the parse schema. More precisely, we add a dot • to the items to highlight which symbol should be expanded. The different strategies then correspond to different ways of moving the dot through the parse items.

Left-to-right, depth-first The only axiom is $[0, \bullet S, n]$, and the only goal is $[n, \bullet, n]$.

$$\textbf{Scan} \qquad \frac{[i, \bullet a\beta, j]}{[i+1, \bullet\beta, j]}\ a = w_i$$

$$\textbf{Predict} \qquad \frac{[i, \bullet N\beta, j]}{[i, \bullet\gamma\beta, j]}\ N \to \gamma \in R$$

This type of top-down parser is also called a *recursive descent parser*.

Left-to-right, breadth-first As before the axiom and goal are $[0, \bullet S, n]$ and $[n, \bullet, n]$, respectively.

$$\textbf{Scan} \qquad \frac{[i, \bullet a\beta, j]}{[i+1, \bullet\beta, j]}\ a = w_i$$

$$\textbf{Predict} \qquad \frac{[i, \alpha \bullet N\beta, j]}{[i, \alpha\gamma \bullet \beta, j]}\ N \to \gamma \in R$$

$$\textbf{Return} \qquad \frac{[i, \beta\bullet, j]}{[i, \bullet\beta, j]}\ \beta \in (N \cup T)^+$$

Example 3.2 Breadth-first parse of *The anvil hit Daffy*

Once again we assume that G is the grammar at the beginning of the handout.

parse item	inference rule
[0,•S,4]	axiom
[0,NP VP•,4]	predict(1)
[0,•NP VP,4]	return
[0,Det N •VP,4]	predict(3)
[0,Det N Vt NP •,4]	predict(5)
[0,•Det N Vt NP,4]	return
[0,the •N Vt NP,4]	predict(6)
[0,the truck •Vt NP,4]	predict(7)
[0,the truck hit •NP,4]	predict(10)
[0,the truck hit PN •,4]	predict(2)
[0,•the truck hit PN,4]	return
[1,•truck hit PN,4]	scan
[2,•hit PN,4]	scan
[3,•PN,4]	scan
[3,Daffy •,4]	predict(8)
[3,•Daffy,4]	return
[4,•,4]	scan

And as before the parser actually predicts a lot more items, the table only lists those that are used in the successful parse.

Exercise 3.3. Specify the right-to-left versions of the parsers above. ⊙

Exercise 3.4. Parse the sentence *The truck fell over* will all four different types of top-down parsers. Use tables such as the ones in the previous two examples. ⊙

Exercise 3.5. The left-to-right breadth-first parser differs slightly from the intuitive one we saw in Sec. 1. In what respect? And which one of the two versions would be more efficient for syntactic processing? (Hint: draw a phrase structure tree as before and annotate the nodes according to when they are introduced by the parser, and when they are scanned or rewritten. Then compare this tree to the one in Sec. 1.) ⊙

Exercise 3.6. The parse items for all the parsers above include both the beginning and the end of the substring they span. This seems a bad fit for incremental processing, where the length of the input string isn't known in advance. Can we remove the index of the end position from the parse items and still have a functional parser? What changes must be made to the inference rules, axioms, and goals? ⊙

2.4 Data Structure

There's a myriad of different data structures a top-down parser may employ. We won't discuss them in great detail, but let's quickly summarize what the parser needs to be able to do.

- **Store history of successful parses**
  Recall that a parser differs from a recognizer in that the latter only groups sentences into grammatical and ungrammatical whereas the former also assigns tree structures to well-formed sentences. This structure can be easily inferred from the parse history, but this means that the parser needs to keep track of this history.

- **Dealing with non-determinism**
  We have implicitly assumed in our examples that the parser always makes the right decision with respect to which rewrite rule should be applied to which non-terminal symbol. But this is of course not the case in practice. Instead, the parser needs a way to deal with multiple parses, which means being able to store multiple parse histories.

A single parse history can be stored as a table, for example. Multiple parses, then, are a table of such parse history tables.

Exercise 3.7. Formulate a strategy for converting a given parse history, stored as a table, into the corresponding phrase structure tree. ⊙

Smarter data structures can greatly improve the parser's efficiency. For instance, identical items can be shared across parse histories. That way, if the parser currently

has three histories containing the item [4,•NP,20], the possible inferences from this item only need to be computed once rather than three times, which would be a waste of resources. Similarly, some of the inferences from this item can also be reused for a parse with item [2, •NP, 18]. These techniques for sharing structure and memorizing the results of computation so that they can be reused somewhere else later on fall under the umbrella of *dynamic programming* and *tabulation*. They are very powerful ideas, but their addition does not alter the parsing schema or the control structure.

However, the control structure needs to include some strategy for how distinct parses should be prioritized. For instance, one may finish one parse table before moving on to the next, or interleave the different tables. Often parse tables are put in a *priority queue*. At each step, the first table in the queue is opened and one inference step is applied. Depending on the outcome of this inference, the table may be removed from the top spot and inserted somewhere else in the queue. Similarly, the inference step may have created new parse tables that also need to be inserted somewhere in the queue.

References and Further Reading

Goodman, Joshua. 1999. Semiring parsing. *Computational Linguistics* 25:573–605.

Pereira, Fernando C. N., and David Warren. 1983. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, 137–144.

Shieber, Stuart M., Yves Schabes, and Fernando C. Pereira. 1995. Principles and implementations of deductive parsing. *Journal of Logic Programming* 24:3–36.

Sikkel, Klaas. 1997. *Parsing schemata*. Texts in Theoretical Computer Science. Berlin: Springer.

Lecture 4

Top-Down Parsing: Psycholinguistic Adequacy

1 Two Basic Advantages

1.1 Incrementality

The most basic requirement for a psychologically plausible parser is that it works in an incremental fashion, that is to say, parsing can take place as soon as the first word of the sentence has been uttered, rather than delaying parsing until the full sentence has been heard. You showed in an earlier exercise that the top-down parser only needs to keep track of the starting position of an item, which entails that it can be run in an incremental fashion.

1.2 Simplicity

Top-down parsers are conceptually simple. The top-down orientation of the parser mirrors the view of CFGs as top-down generators. In addition, the inference rules are extremely simple (compared to, say, *left-corner parsing*, which we will discuss at a later point), seeing how the inference rule of the parser directly mirrors the rewrite rules of the grammar. This is particularly appealing for proponents of a *transparent parser*. A transparent parser must use the grammar in as direct a fashion as possible. That precludes processing-based explanations of syntactic phenomena — e.g. island effects — and prioritizes a maximally simple parser.

2 Garden Path Effects

2.1 Garden Paths as Backtracking

One of the best known phenomena in syntactic processing are *garden path effects*, which arise with sentences that are grammatically well-formed but nonetheless difficult to parse (Frazier 1979; Frazier and Rayner 1982). More precisely, a garden path sentence is a sentence w that up to some w_i has a strongly preferred analysis that must be discarded at w_{i+1}. Here's a list of examples that includes more than the usual suspects:

(1) *Structural ambiguity*

a. The horse raced past the barn fell.

b. The raft floated down the river sank.
c. The player tossed a Frisbee smiled.
d. The doctor sent for the patient arrived.
e. The cotton clothing is made of grows in Mississippi.
f. Fat people eat accumulates.
g. I convinced her children are noisy.

(2) *Lexical ambiguity*
a. The old train the young.
b. The old man the boat.
c. Until the police arrest the drug dealers control the street.
d. The dog that I had really loved bones.
e. The man who hunts ducks out on weekends.

Frazier (1979) proposes to treat garden path effects as a result of reanalysis: the parser is forced to abandon its current set of hypotheses, backtrack to an earlier point, and build a new structure from there. If for some reason this process proves too difficult, the parser gets stuck and assigns no structure at all.

This kind of analysis is compatible with top-down parsing, depending on the data structure we use. Remember that the data structure keeps track of the items in the current parse, but also of alternative parses. The control structure includes a set of instructions for which parses should be prioritized. Let's try to make Frazier's account precise using a top-down parser with priority queues as data structure and a serial strategy for expanding multiple parses.

2.2 Priority Queues and Prefix Trees

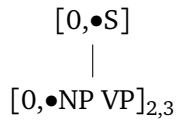
Let's assume that our data structure is a *priority queue*, i.e. a list of parse tables that can be reordered, to which we can add new entries, and from which we can delete old entries (note: even though I call it a list, that doesn't mean queues are best implemented as lists; in Python, for example, you're better off using arrays).

As so often, a more abstract perspective is helpful here. Instead of a list of parse tables, we can use a *prefix tree* as a unified way of representing multiple parses at once. The root of the prefix tree is always our start axiom [0,S] (since the final position is redundant for top-down parsing, it can be safely omitted). If parse item q is obtained from item p via a prediction or scan rule, q is added to the tree as a daughter of p. Furthermore, each node that is not a leaf is also subscripted with the list of unused predict rules. We call a node in this tree a *parse leaf* iff it is a leaf or has at least one subscript, and a (strictly descending) path spanning from the root to a leaf or a parse leaf is called a *parse path*. The set of parse tables, then, corresponds exactly to the set of parse paths. Putting all of this together, we can implement a priority queue as a ranking of parse paths through the prefix tree.

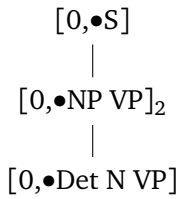
Example 4.1

Parse Tables as Trees Suppose we are using the grammar from the beginning of Lecture 3 to parse *the anvil hit Daffy* with a recursive descent parser. Our parse table

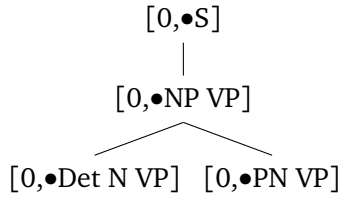
starts with [0,•S] as usual, from which we can only predict [0,•NP VP]. This can be represented as the tree below.



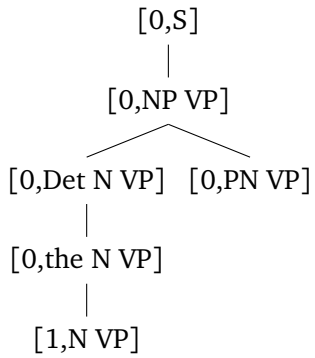
Note that [0,•NP VP] has subscripts 2 and 3 to indicate that we can use rules 2 and 3 of our CFG to predict new parse items. Suppose the parser uses predict(3) to create the item [0,•Det N VP]. Then this item is added as a daughter of [0,•NP VP] to the previous tree, and the subscript 3 is also removed from [0,•NP VP]. We do not need to add a subscript to [0,•Det N VP] since it is a leaf.



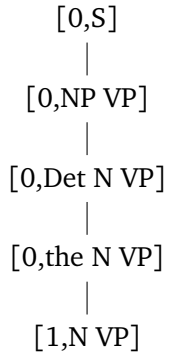
The next step of the parser now depends on how the control structure prioritizes distinct parses. We can either expand the table that ends in [0,•Det N VP], or go back to the one ending in [0,•NP VP] and apply predict(2). Suppose we do the latter, so that [0,•PN VP] is added as the second daughter of [0,•NP VP], which thus loses its last subscript.



Now let's expand [0,•Det N VP] again to obtain [0, •the N VP] and then apply a scan step, yielding [1, •N VP].



If our parser is really smart, it will be able to infer at this point that the scanned word *the* can never be obtained from [0,PN VP] (technically this is achieved by associating every parse item p with a regular expression that describes the possible left edges of the strings that can be derived from p). So the parser can remove [0,PN VP] from the tree, which is tantamount to discarding the parse table where NP was rewritten as PN.



What makes the tree representation of parse tables appealing is that the construction and prioritization of parse tables can be reduced to strategies for tree building. In particular, the familiar notions of depth-first and breadth-first carry over in a natural fashion.

depth first/serial expand a parse item p that was introduced during the previous parse step; if the parse item p cannot be expanded, expand the lowest parse leaf l that dominates p

breadth first/parallel before expanding a parse item introduced during parse step j, all parse items that were introduced at parsing step i must have been rewritten, for every $i < j$

The depth first strategy leads to a parser that always builds a single complete parse history rather than multiple partial ones. If the parse history cannot be expanded anymore, the parser either stops (successful parse) or backtracks to the last choice point in the parse history and tries a different choice instead. This corresponds exactly to the notion of *serial parsing* in the psycholinguistic literature.

The intuitive counterpart of serial parsing is *fully parallel parsing*, where all parse tables are built up at the same time. This is exactly the behavior of the breadth-first strategy. However, even proponents of parallel parsing usually do not assume that the human parser computes and stores all options all the time. Instead, only a subset of very likely parses is claimed to be worked on in parallel, with all others either discarded or at least not expanded on. On a technical level we may formalize this in terms of a probabilistic procedure for which parse items may be expanded, and in which order. The details are not important here, the basic insights is simply that just as we can add probabilities to the control schema to regulate which inference rules the parser may apply, we can also use probabilities to prioritize the expansion of certain parse tables over others.

2.3 Formal Account of Garden Paths

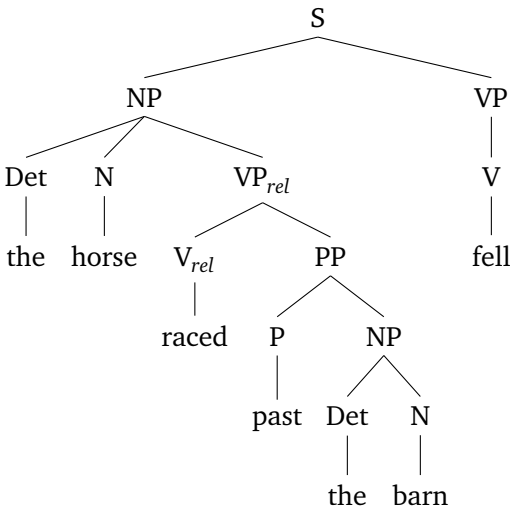
A recursive descent parser with a depth-first expansion strategy for prefix trees, coupled with a preference ranking over prediction steps, can easily account for garden path effects. In the case of *the horse raced past the barn fell*, for example, the parse builds a single parse table, and due to how certain rewrite rules are preferred over others, this parse table encodes the structure for *the horse raced past the barn*. When encountering *fell*, the parser has to backtrack. A successful parse requires backtracking to the point where *raced* is analyzed as part of the VP rather than the NP — that's quite a distance.

Example 4.2 Backtracking in *the horse raced past the barn fell*

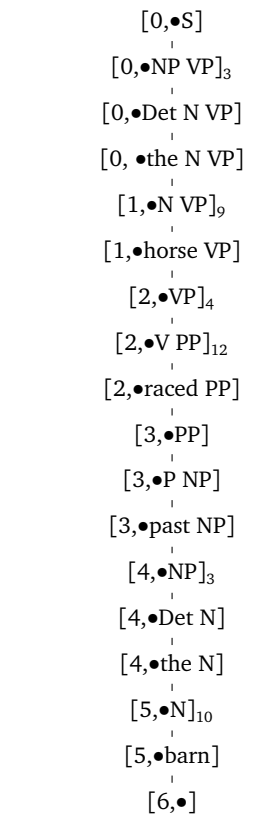
Assume we have the following (massively simplified) grammar:

1)	S	→NP VP	8)	Det	→the
2)	NP	→Det N	9)	N	→barn
3)	NP	→Det N VP_{rel}	10)	N	→horse
4)	VP	→V	11)	P	→past
5)	VP	→V PP	12)	V	→fell
6)	VP_{rel}	→V_{rel} PP	13)	V	→raced
7)	PP	→P NP	14)	V_{rel}	→raced

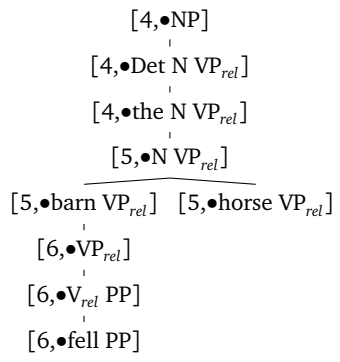
Here's the resulting tree for our garden path sentence.



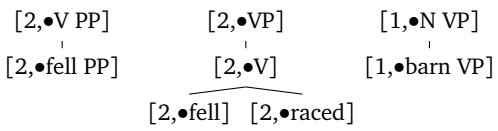
If the parser prefers NP → Det N over NP →Det N CP and operates in a recursive descent fashion, the construction of the first parse table results in the prefix tree below.



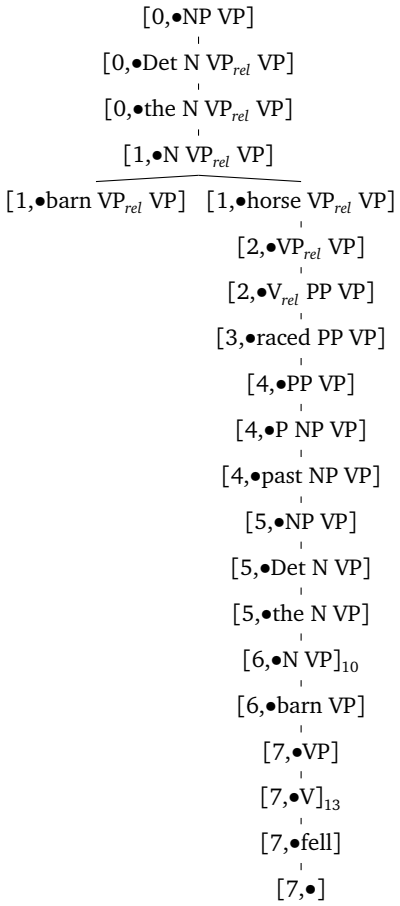
Since this parse does not succeed, the parser needs to backtrack. The closest choice point is $[5,\bullet N]_{10}$, which obviously does not fix the problem of integrating *fell* into the structure, as can be verified after a single scan step. The next choice point is $[4,\bullet NP]_3$. Here the parser still has the option of replacing NP by Det N CP, which won't help much either, but it takes quite a while to realize this because the tree for the parse table obtained by following this option involves a choice point, too.



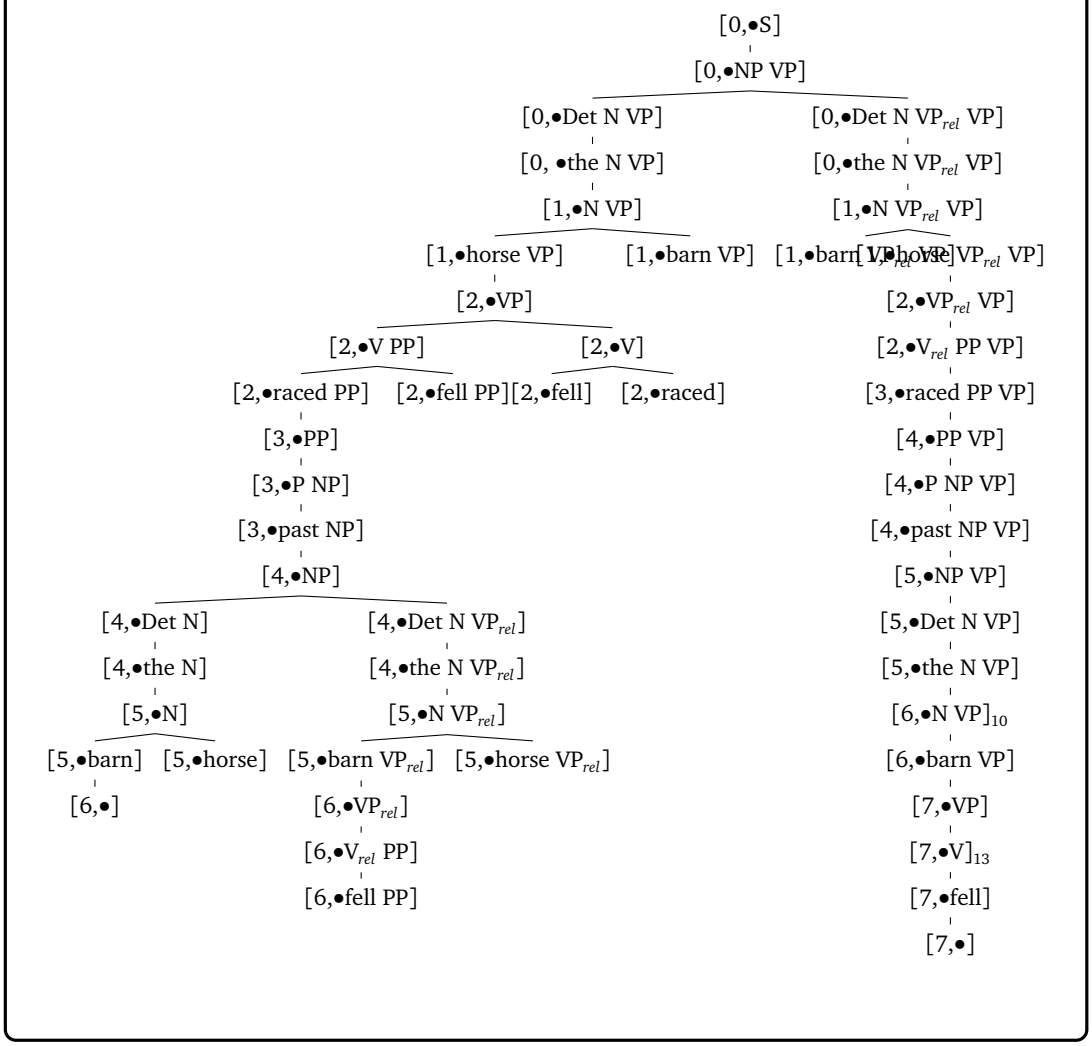
Since this venue didn't yield a successful parse either, the parse backtracks to $[2,\bullet V\ PP]_{12}$, and after this fails, to $[2,\bullet VP]_4$. Once again it is not successful, and the same holds once it expands [1,•N VP].



Only if the parser backtracks all the way to $[0,\bullet\text{NP VP}]_3$, essentially undoing all its work so far, can it find a working parse.



The prefix tree for all the parse histories built by the parser before it encounters a successful parse it much bigger than the simple phrase structure tree for the sentence.



2.4 Center Embedding

It is a well-known fact that center embedding constructions are harder to parse than right embedding constructions, even if one controls for sentence length, discourse and vocabulary.

(3) a. I bought the cheese (that) the mouse (that) the cat (that) the dog chased ate wanted.

b. I bought the cheese (that) the mouse wanted (that) the cat ate (that) the dog wanted.

A common idea is that the difficulty of center embedding has to do with an increased memory burden, either because more items have to be kept in memory for a non-trivial amount of time, or because specific items have to be stored for a longer time, or a combination of the two. The specifics vary between accounts.

Kimball (1973), for instance, proposes that the parser cannot work on more than two sentences at once, i.e. it is possible to keep up to two CPs in memory, but not more than that. Right embedding of CPs is fine because opening a new CP is the last step

needed to close the containing CP, which can subsequently be removed from memory. Center embedding, on the other hand, requires that both CPs be stored in memory, so it is natural limited to one level of embedding.

Gibson's (1998) *Syntactic Prediction Locality Theory* (SPLT) contends that memory burden increases the more dependencies need to be stored at the same time, and since center embedding of the type NP-S-VP necessarily involves starting a new dependency before the old one between NP and VP has been finished, center embedding is difficult.

This argument can be ported into our more formal setting via a *linking hypothesis* between control structure and processing difficulty. Remember that we can annotate phrase structure trees to indicate the order in which nodes are introduced and rewritten by the parser. Obviously any item that is not rewritten immediately after its introduction — i.e. any item whose prefix and suffix differ by more than 1 — needs to be stored in memory. Let's put these items in boxes so that they are easy to pick out at a glance. Parsing difficulty, then, can be measured in a variety of ways.

Tenure the *tenure of node n* is the difference between its subscript and its superscript

Payload the *payload of tree t* is the number of nodes whose tenure is strictly greater than 1

These two measures can be combined to create a variety of difficulty metrics such as the two below (cf. Kobele et al. 2012 and Graf and Marcinek 2014).

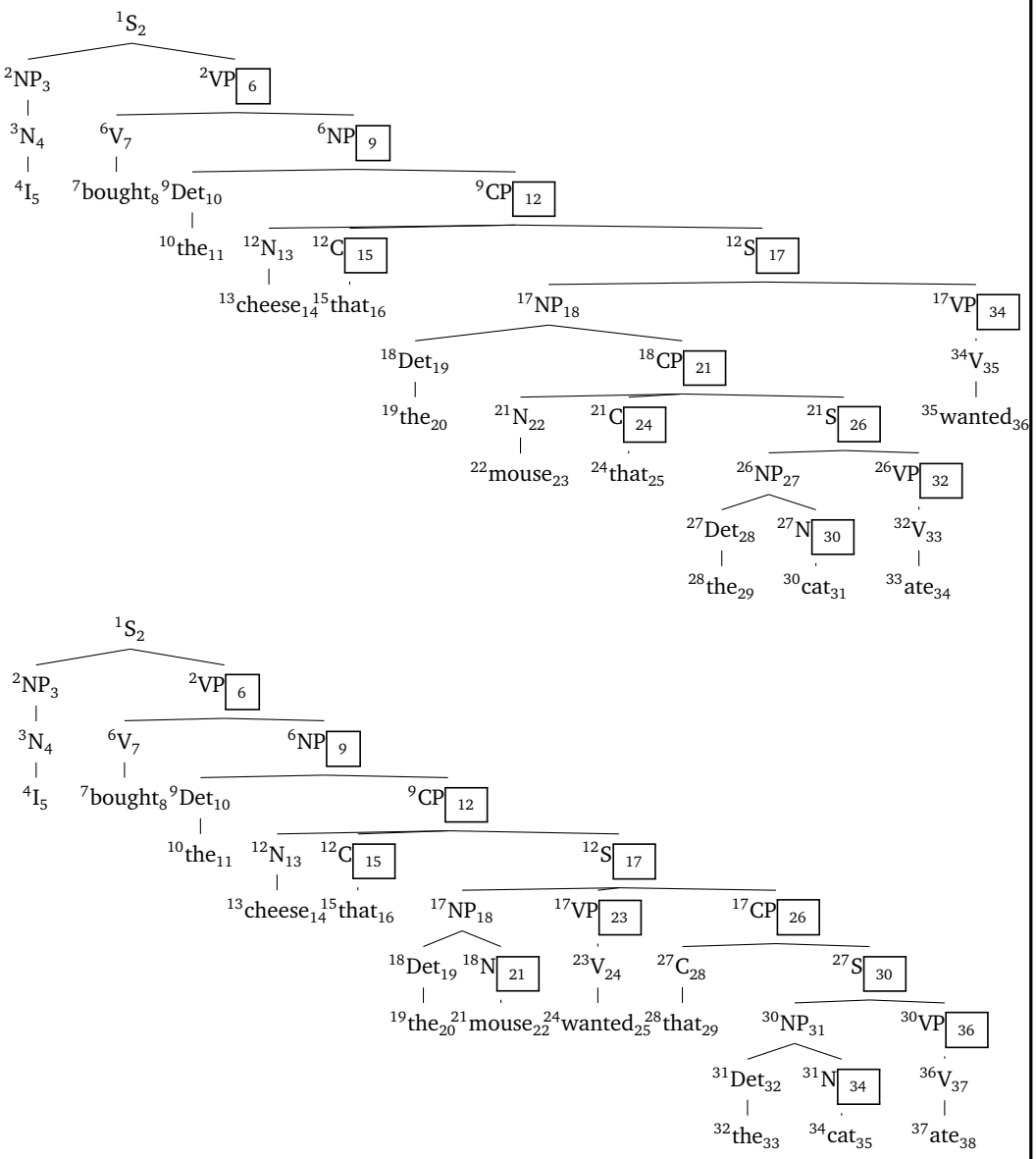
MaxTen greatest tenure among all the nodes; $\max(\{\text{tenure}(n)\})$

SumTen total number of steps that items must be memorized; $\sum(\{\text{tenure}(n) > 1\})$

All of these metrics capture the fact that right embedding isn't harder than center embedding. **MaxTen** and **SumTen** also predict right embedding to be easier.

Example 4.3 Center embedding and maximum tenure

Consider the center embedding and right embedding trees below, which have been annotated according to the behavior of a recursive descent parser. For center embedding we are using a promotion-style analysis of relative clauses (Vergnaud 1974; Kayne 1994), which posits that the relative clause is an argument of the determiner, with the head noun residing in a CP specifier. For right embedding the extraposed relative clause is considered a daughter of S.



The difficult center embedding sentence has a payload of 11, a maximum tenure of 17, and a sum tenure of 55. The easier right embedding sentence has a payload of 11, a maximum tenure of 9, and a sum tenure of 48. Irrespective of how we weigh or rank these three metrics, then, right embedding is never predicted to be harder than center embedding. As long as we do not take payload as the only difficulty metric, right embedding is correctly predicted to be easier than center embedding.

Notice that our explanation for the difficulty of center embedding is very different from the one we used for garden path effects. Garden path effects were explained in terms of the difficulty of finding a right parse among the many possible ones. We used prefix trees as a way of representing the parser's route through this search space, and since a prefix tree encodes many parsing tables at ones, our account is ultimately based

on the processing challenges of coordinating multiple parse tables — if all parse tables could be easily stored in memory and expanded in a breadth-first manner, garden path effects would not arise.

Center embedding, on the other hand, is explained purely in terms of how the parser constructs the correct parse. How the parser actually finds this parse does not factor into the explanation. The claim is that even if the parser had a perfect *oracle*, a machine that could tell it at every step which inference step must be taken to get the correct structure, the difference between center embedding and right embedding would not disappear because the former still puts a higher load on working memory than the latter.

3 Problems

3.1 Memory Usage in Left Embedding

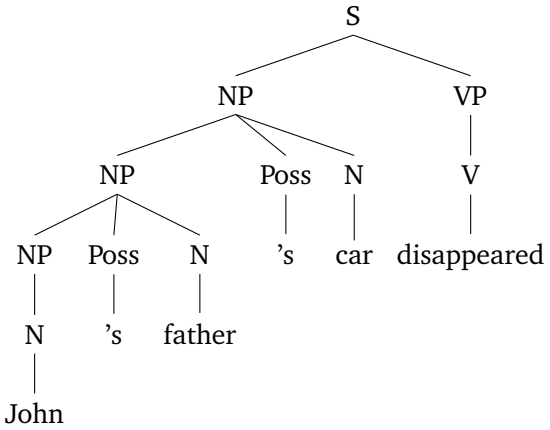
A careful examination of the top-down parsers behavior over center embedding constructions reveals that these configurations aren't the only ones that should cause an increased memory burden. Coupled with our metrics of processing difficulty, a recursive descent parser predicts that left embedding should be hard to parse, too — which is not borne out empirically.

(4) a. The exhaust pipe of the car of the father of the mechanic is broken.
b. The mechanic's father's car's exhaust pipe is broken.

The cause for the predicted difficulty spike with left embedding is straight-forward. Suppose the parser conjectures at step i that node n in the tree has daughters d_1 and d_2. Steps $i+1$ to j are spent by the parser expanding the subtree rooted in d_1. The bigger the subtree, the higher the value of $j+1$, the point at which the parser can move on to expanding d_2. Since d_2 must be kept in memory from step i until step $j+1$, and the value of j increases with the size of the subtree rooted in d_1, every construction that increases the size of a left sibling — including left embedding — should increase parsing difficulty.

3.2 Looping in Left Recursion

Left embedding constructions are problematic for top-down parsers in more respects than just psycholinguistic adequacy. Consider a grammar that licenses possessive structures such as the one below.



In order to arrive at this structure, the parser must first infer [0,•NP VP], and then expand the NP to get [0,•NP Poss N VP]. But now the parser once again has to expand the NP, possibly creating [0,•NP Poss N Poss N VP]. It is easy to see that the parser can keep rewriting NP *ad infinitum*, essentially looping the NP rewriting step and producing longer and longer parse items. Without a smart control structure that detects this kind of looping, the parser will soon run out of working memory, or if there is no limited on the amount of memory, keep looping forever without ever constructing a parse.

A standard solution is to use a probabilistic control structure that prunes away all parse tables that fall below a certain probability threshold. As the parser keeps conjecturing bigger and bigger structures, the probability of these parses decreases until they are eventually pruned away. This also called a *beam parser*. Beam parsers require a method for automatically inferring the necessary probabilities from a corpus, and their overall behavior can be difficult to figure out. So the appealing simplicity of top-down parsers is lost to some extent.

3.3 Merely Local Syntactic Coherence Effects

Merely local syntactic coherence effects is a term coined by Tabor et al. (2004) to refer to cases where an analysis that is locally well-formed but incompatible with the structure built so far nonetheless induces a temporary increase in parsing difficulty (see also Konieczny 2005, Konieczny et al. 2009, and Bicknell et al. 2009). This is exemplified by the contrast in (5). During self-paced reading experiments, subjects' reading speed of (5a) decreases at *tossed*, indicating an increase in processing difficulty. The effect disappears, however, when *tossed* is replaced by *thrown*.

(5) a. The coach smiled at the player tossed a frisbee.
b. The coach smiled at the player thrown a frisbee.

It seems as if the fact that *the player* can be locally analyzed as the subject of *tossed* confuses the parser. Since this interpretation is not available with *thrown* due to the unambiguous past participle morphology, no slowdown occurs.

Merely local syntactic coherence effects are completely unexpected with a top-down parser because there is no way the parser could infer an item that treats *tossed* as a finite verb with *the player* as its subject. The preceding parts of the sentence have already disambiguated the structure of the sentence to an extent where this is no longer a viable hypothesis. These effects are expected, however, if the parser proceeds bottom-up instead of top-down.

References and Further Reading

Bicknell, Klinton, Roger Levy, and Vera Demberg. 2009. Correcting the incorrect: Local coherence effects modeled with prior belief update. In *Proceedings of the 35th Annual Meeting of the Berkeley Linguistics Society*.

Frazier, Lyn. 1979. *On comprehending sentences: Syntactic parsing strategies*. Doctoral Dissertation, University of Connecticut.

Frazier, Lyn, and Keith Rayner. 1982. Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology* 14:178–210.

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

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.

Kayne, Richard S. 1994. *The antisymmetry of syntax*. Cambridge, Mass.: MIT Press.

Kimball, John. 1973. Seven principles of surface structure parsing in natural language. *Cognition* 2:15–47.

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

Konieczny, Lars. 2005. The psychological reality of local coherences in sentence processing. In *Proceedings of the 27th Annual Conference of the Cognitive Science Society*.

Konieczny, Lars, Daniel Müller, Wibke Hachmann, Sarah Schwarzkopf, and Sascha Wolfer. 2009. Local syntactic coherence interpretation. Evidence from a visual world study. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, 1133–1138.

Tabor, Whitney, Bruno Galantucci, and Daniel Richardson. 2004. Effects of merely local syntactic coherence on sentence processing. *Journal of Memory and Language* 50:355–370.

Vergnaud, Jean-Roger. 1974. *French relative clauses*. Doctoral Dissertation, MIT.

Lecture 5

Bottom-Up Parsing

The natural counterpart to top-down parsing is bottom-up parsing, where trees are built starting at the leaves and moving towards the root. Like top-down parsing, bottom-up parsing is fairly simple and builds the foundation of a variety of parsing algorithms. But just like top-down parsing, it has certain shortcomings regarding psycholinguistic adequacy.

1 Intuition

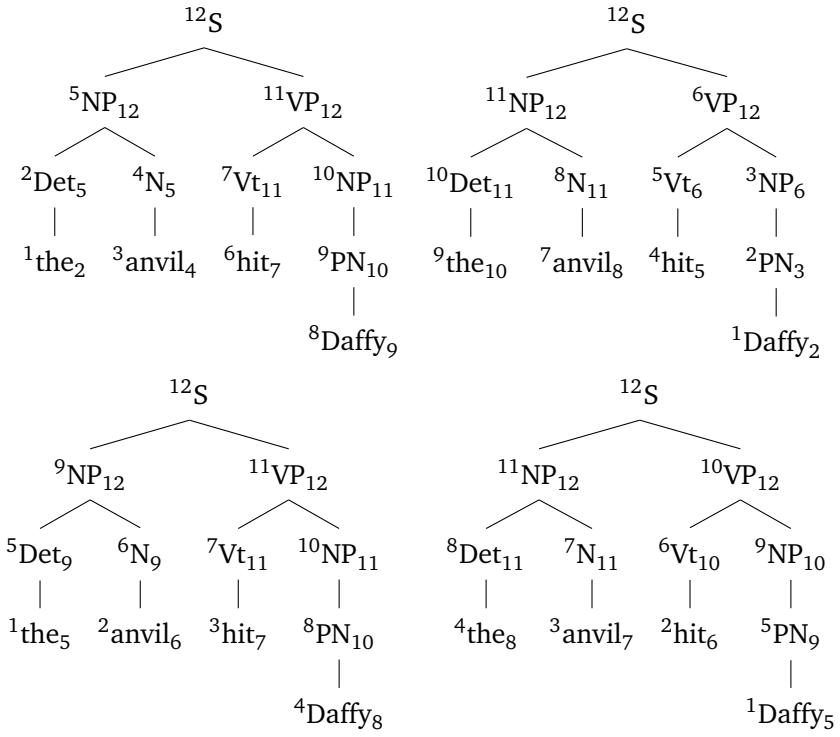
For illustration we use the same grammar as for the top-down parser in Lecture 3.

1)	S	→ NP VP	6)	Det	→	a | the
2)	NP	→ PN	7)	N	→	car | truck | anvil
3)	NP	→ Det N	8)	PN	→	Bugs | Daffy
4)	VP	→ Vi	9)	Vi	→	fell over
5)	VP	→ Vt NP	10)	Vt	→	hit

A bottom-up parser essentially applies the rewrite rules in reverse. If the current input i appears on the right-hand side of a rewrite rule for N, replace i by N.

string	**rule**
the	read input
det	Det →the
det anvil	read input
det N	N →anvil
NP	NP →Det N
NP hit	read input
NP Vt	Vt →hit
NP Vt Daffy	read input
NP Vt PN	PN →Daffy
NP Vt NP	NP →PN
NP VP	VP →Vt NP
S	S →NP VP

The order in which rules are applied once again gives rise to at least four different types of parsers, a helpful first approximation of which can be gleaned from the examples below. The four trees below correspond to (in clockwise order) left-to-right depth first, right-to-left depth first, right-to-left breadth first, and left-to-right breadth first.



2 Formal Specification

2.1 Parsing Schema

Since bottom-up parsers are essentially the dual of top-down parsers, the former's deductive definition closely resembles that of the latter. Whereas a top-down parser starts with $[0, S, n]$ and seeks to derive $[n, , n]$, the bottom-up parser has axiom $[0, , 0]$ and goal $[0, S, n]$. So axioms and goals are simply switched (and $[n, , n]$ is replaced by $[0, , 0]$). Similarly, the top-down scan and predict rules have bottom-up counterparts Shift and Reduce. The reduce rule is exactly the predict rule of a left-to-right top-down parser with top and bottom switched. The shift rule is the scan rule of a right-to-left top-down parser with top and bottom switched.

$$\textbf{Shift} \qquad \frac{[i, \beta, j]}{[i, \beta a, j+1]}\, a = w_j$$

$$\textbf{Reduce} \qquad \frac{[i, \alpha\gamma\beta, j]}{[i, \alpha N\beta, j]}\, N \to \gamma \in R$$

Example 5.1 Bottom-up parse of *The anvil hit Daffy*

Here's an example for our standard example sentence *The anvil hit Daffy* using the parsing schema above. Note that rules are applied in arbitrary order to reflect the fact that the parsing schema still lacks a control structure and thus imposes no specific rule order.

parse item	inference rule
[0,,0]	axiom
[0,the,1]	shift
[0,the anvil,2]	shift
[0,the N,2]	reduce(7)
[0,Det N,2]	reduce(6)
[0,Det N hit,3]	shift
[0,NP hit,3]	reduce(3)
[0,NP hit Daffy,4]	shift
[0,NP hit PN,4]	reduce(8)
[0,NP Vt PN,4]	reduce(10)
[0,NP Vt NP,4]	reduce(2)
[0,NP VP,4]	reduce(5)
[0,S,4]	reduce(1)

2.2 Control Structure

The control structure can be encoded by adding the familiar • to the rules.

Left-to-right, depth-first The only axiom is $[0, \bullet, 0]$, and the only goal is $[0, S\bullet, n]$.

$$\textbf{Shift} \qquad \frac{[i, \beta\bullet, j]}{[i, \beta a\bullet, j+1]}\ a = w_j$$

$$\textbf{Reduce} \qquad \frac{[i, \alpha\gamma\bullet, j]}{[i, \alpha N\bullet, j]}\ N \rightarrow \gamma \in R$$

This kind of parser is also called a *shift reduce parser*. Notice that even though the parser reads the input from left-to-right, the structure building process is partially right-to-left since the reduction rule reduces elements to the left of • from right to left.

Left-to-right, breadth first It is surprisingly difficult to specifier a breadth first bottom-up parser in a deductive fashion (more on that below), and consequently the rules are a lot more complicated. The axioms and goals are the same as for the depth-first parser, though.

$$\textbf{Shift} \qquad \frac{[i, \beta\bullet, j]}{[i, \beta a\bullet, j+1]}\ a = w_j$$

$$\textbf{Reduce} \qquad \frac{[i, \alpha \bullet \beta\gamma\delta, j]}{[i, \alpha\beta N \bullet \delta, j]}\ N \rightarrow \gamma \in R$$

$$\textbf{Return} \qquad \frac{[i, \alpha \bullet \beta, j]}{[i, \bullet\alpha\beta, j]}\ \alpha \in (N \cup T)^+,\ \neg\exists N[N \rightarrow \beta \in R]$$

Example 5.2 Depth-first parse of *The anvil hit Daffy*

parse item	inference rule
[0,•,0]	axiom
[0,the •,1]	shift
[0,Det •,1]	reduce(6)
[0,Det anvil •,2]	shift
[0,Det N •,2]	reduce(7)
[0,NP •,2]	reduce(3)
[0,NP hit •,3]	shift
[0,NP Vt •,3]	reduce(10)
[0,NP Vt Daffy •,4]	shift
[0,NP Vt PN •,4]	reduce(8)
[0,NP Vt NP •,4]	reduce(2)
[0,NP VP •,4]	reduce(5)
[0,S •,4]	reduce(1)

Example 5.3 Breadth-first parse of *The anvil hit Daffy*

parse item	inference rule
[0,•,0]	axiom
[0,the •,1]	shift
[0,the anvil •,2]	shift
[0,the anvil hit •,3]	shift
[0,the anvil hit Daffy •,4]	shift
[0,•the anvil hit Daffy,4]	return
[0,Det •anvil hit Daffy,4]	reduce(6)
[0,Det N •hit Daffy,4]	reduce(7)
[0,Det N Vt •Daffy,4]	reduce(10)
[0,Det N Vt PN •,4]	reduce(8)
[0,•Det N Vt PN,4]	return
[0,NP •Vt PN,4]	reduce(3)
[0,NP Vt NP •,4]	reduce(2)
[0,•NP Vt NP,4]	return
[0,NP VP •,4]	reduce(5)
[0,•NP VP,4]	return
[0,S •,4]	reduce(1)

Exercise 5.1. Mirroring the redundancy of the index j for top-down parsers, index i can safely be eliminated from the parse items of a bottom-up parser. Explain why. ⊙

Exercise 5.2. Our breadth-first parser is actually too permissive. Show that the rules as given can also be used to construct a depth-first parse of *The anvil hit Daffy*. ⊙

Exercise 5.3. Is there a way to restrict the breadth-first parser so that it can no longer construct depth-first parses? ⊙

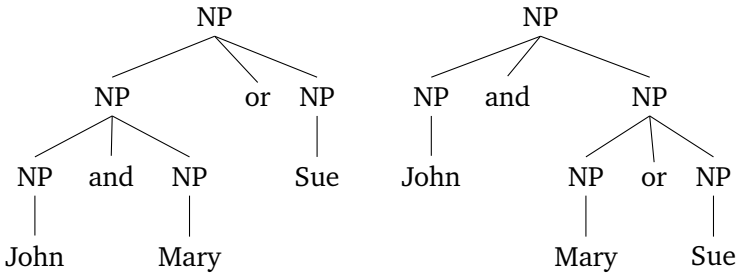
Notice that both parser have a certain overlap between the rules. In the case of the depth-first parser, the domains of the shift and reduce rules are not disjoint. That is to say, for some parse items both shift and reduce are valid continuations of the parse. This is a necessary consequence of non-determinism in the grammar. A top-down parser faces non-determinism with respect to which rewrite rule to apply to a given non-terminal N. For a bottom-up parser, this step is usually deterministic because distinct non-terminals can be assumed to also have distinct right-hand sides in rewrite rules (rules assigning parts of speech to words are one notable exception). But the bottom-up parser must decide whether to reduce right away or read another input symbol first, which might allow for a different reduction step.

Example 5.4 A depth-first parse with delayed shift

Consider the grammar below, which generates conjunctions and disjunctions of proper names.

1) NP → NP and NP
2) NP → NP or NP
3) NP → John | Mary | Sue

Now consider the phrase *John and Mary or Sue*, which has two semantically distinct structures.



If the bottom-up parser reduces as early as possible, we get the tree to the left. This also shows that reduction proceeds from the right edge of the parse item.

parse item	**inference rule**
[0,•,0]	axiom
[0,John •,1]	shift
[0,NP •,1]	reduce(3)
[0,NP and •,2]	shift
[0,NP and Mary •,3]	shift
[0,NP and NP •,3]	reduce(3)
[0,NP •,3]	reduce(1)
[0,NP or •,4]	shift
[0,NP or Sue •,5]	shift
[0,NP or NP •,5]	reduce(3)
[0,NP •,5]	reduce(2)

In order to obtain the other tree, the parser must delay the application of reduce(1) until reduce(2).

parse item	inference rule
[0,•,0]	axiom
[0,John •,1]	shift
[0,NP •,1]	reduce(3)
[0,NP and •,2]	shift
[0,NP and Mary •,3]	shift
[0,NP and NP •,3]	reduce(3)
[0,NP and NP or •,4]	shift
[0,NP and NP or Sue •,5]	shift
[0,NP and NP or NP •,5]	reduce(3)
[0,NP and NP •,5]	reduce(2)
[0,NP •,5]	reduce(1)

The breadth-first parser has a comparable overlap between shift and return. Once again this corresponds to the distinction between waiting for more input or building structure on top of the input read so far. The focal point of non-determinism, however, lies in the definition of the reduction rule, which allows the parser to non-deterministically partition the string to the right of • into three segments β, γ and δ to reduce β. This is essential to allow for cases where the symbol immediately after • cannot be reduced, or where reduction would be possible but would prevent the parser from assigning an alternative structure.

Example 5.5 A breadth-first parse with structural ambiguity

Consider now the two available breadth-first parses for the NP *John and Mary or Sue*. The first few steps are the same.

parse item	inference rule
[0,•,0]	axiom
[0,John •,1]	shift
[0,John and •,2]	shift
[0,John and Mary •,3]	shift
[0,John and Mary or •,4]	shift
[0,•John and Mary or Sue,5]	return
[0,NP •and Mary or Sue,5]	reduce(3)
[0,NP and NP •or Sue,5]	reduce(3)
[0,NP and NP or NP •,5]	reduce(3)
[0,•NP and NP or NP,5]	return

The structural difference now depends on which rule the parser applies first, reduce(1) or reduce(2). Keep in mind that both are valid. For reduce(1), we have $\beta = \varepsilon$, $\gamma =$ NP and NP, and $\delta =$ or NP. For reduce(2), we have $\beta =$ NP and, $\gamma =$ NP or NP, and $\delta = \varepsilon$. After each rule, the parser has to use the return rule to move the dot into a position from where it can apply the other reduce rule.

parse item	inference rule
[0,NP •or NP,5]	reduce(1)
[0,•NP or NP,5]	return
[0,NP,5]	reduce(2)

parse item	inference rule
[0,NP and NP •,5]	reduce(2)
[0,•NP and NP,5]	return
[0,NP,5]	reduce(1)

3 Psycholinguistic Adequacy of Shift Reduce Parser

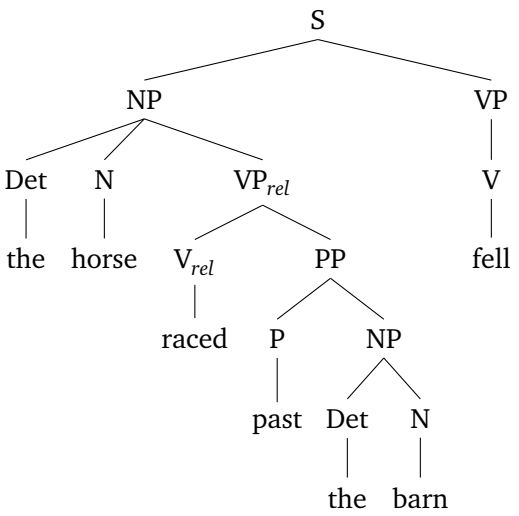
3.1 Garden Paths

If the parser prioritizes reduction over shifting, a serial shift reduce parser with backtracking makes similar predictions to a recursive descent parser.

Example 5.6 Shift-reduce parse for *The horse raced past the barn fell*

We operate with the same grammar as in 2.3.

1)	S	→NP VP	8)	Det	→the
2)	NP	→Det N	9)	N	→barn
3)	NP	→Det N VP_{rel}	10)	N	→horse
4)	VP	→V	11)	P	→past
5)	VP	→V PP	12)	V	→fell
6)	VP_{rel}	→V_{rel} PP	13)	V	→raced
7)	PP	→P NP	14)	V_{rel}	→raced



The parse history is given in Fig. 5.1.

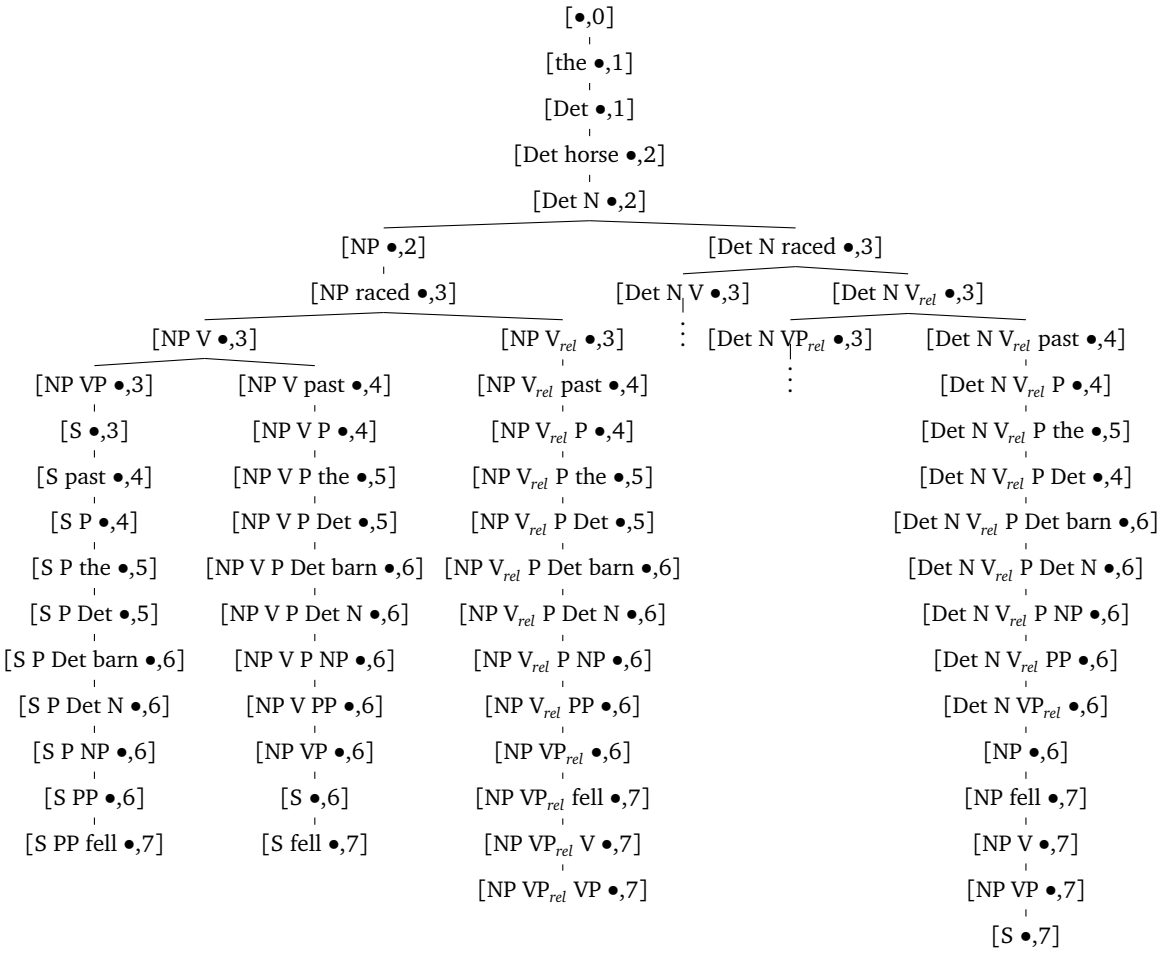


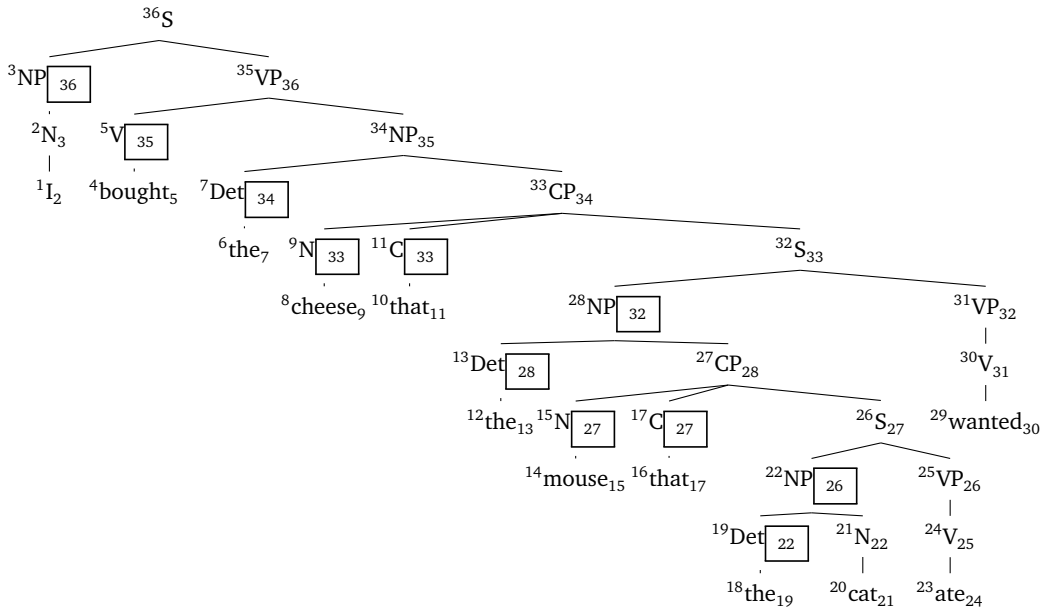
Figure 5.1: Parse history for serial shift reduce parse of *the horse raced past the barn fell*; the parser moves through the history in a recursive descent fashion

3.2 Embeddings

Just like the top-down parser, the bottom-up parser predicts center embedding constructions to be fairly difficult.

Example 5.7 Run of shift reduce parser over center embedding sentence

We use the same promotion-style analysis of relative clauses as in Sec. 2.4.



The parse shows a payload of 11, just as in the case of the recursive descent parser. The maximum tenure is 33, which is a lot more than the recursive descent parser's 17. The sum tenure is 184 (over three times the recursive descent parser's 55).

In contrast to the top-down parser, however, the bottom-up parser doesn't fare any better with the right embedding construction. So just like the top-down parser predicts center embedding to be difficult simply because center embedding involves left embedding dependencies, which top-down parsers struggle with, the bottom-up parser apparently struggles with center embedding because it also struggles with right embedding. The reason is simple:

1. Reduction is of A and B to C is possible only if both A and B have already been recognized.

2. The shift reduce parser fully assembles A before getting started on B. Hence the time A must be stored in memory is directly proportional to the size of B.

3. Right embedding increases the size of B.

Metric	Center	Right
Payload	11	11
MaxTen	17	9
SumTen	55	48

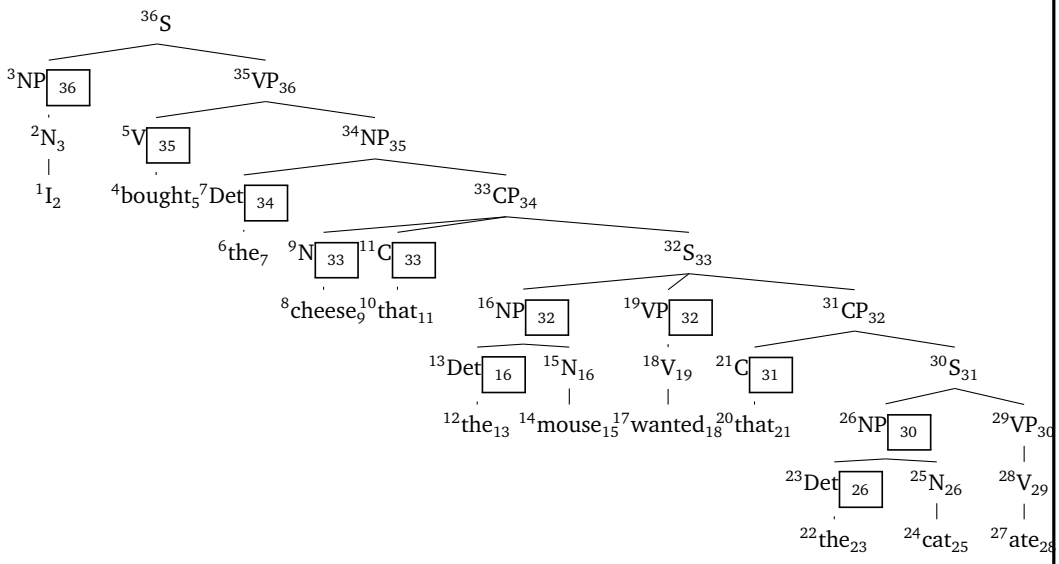
(a) Recursive Descent

Metric	Center	Right
Payload	11	11
MaxTen	33	33
SumTen	184	185

(b) Shift Reduce

Table 5.1: Overview of parser performance for right and center embedding constructions

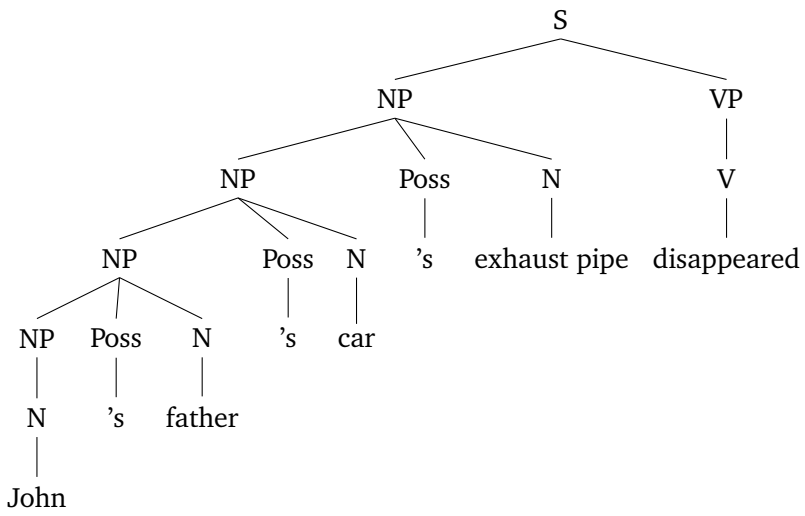
Example 5.8 Run of shift reduce parser over right embedding sentence



The payload is 11, exactly the same as for the center embedding sentence. Maximum tenure is also unchanged at 33. Summed tenure even increased from 184 to 185.

Table 5.1 compares the performance of the recursive descent parser and the shift reduce parser over the two examples sentences.

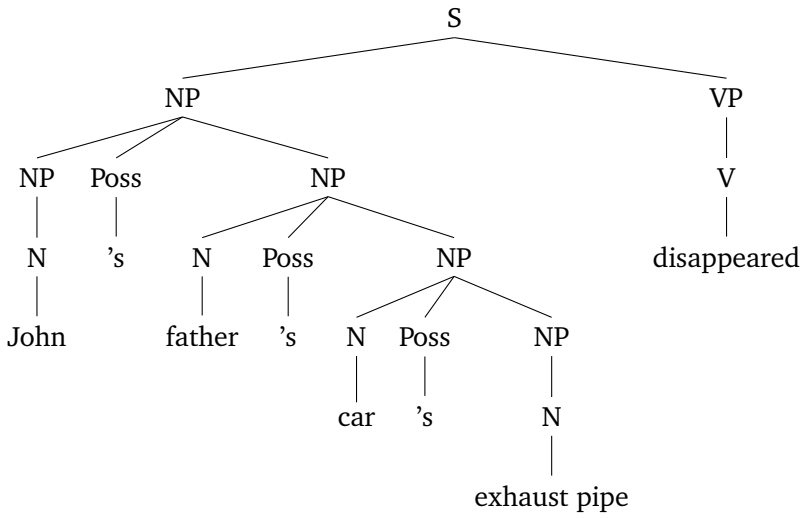
Exercise 5.4. Assume that the sentence *John's father's car's exhaust pipe disappeared* has the structure below.



⊙

Annotate the tree with indices according to the order in which it would be built by I) a recursive descent parser and II) a shift reduce parser. For each parser, calculate payload, maximum tenure and summed tenure. Do you see a difference between the two parsers regarding these difficulty metrics? If so, explain in intuitive terms what causes the performance gap.

Exercise 5.5. Repeat the previous exercise, but now assume the following structure instead:



⊙

Exercise 5.6. Last time we saw that left-to-right top-down parsers cannot account for merely local syntactic coherence effects, irrespective of whether one proceeds depth-first or breadth-first. Can a bottom-up parser account for this phenomenon? Does it have to use a specific search method (depth-first VS breadth-first)? ⊙

3.3 General Remarks

A bottom-up parser that always prefers shift isn't truly incremental and thus a bad model of human sentence processing. A shift reduce parser, on the other hand, is

incremental but not *predictive*. While a specific sequence of shift and reduce steps can block certain analysis (see the coordination example above), a bottom-up parser does not actively restrict its hypothesis. Given an input string $\alpha\beta$ where α has been fully analyzed — i.e. the parser has assigned a single connected subtree to α — the conjectured structure of α has no effect on which structures the parser may entertain for β. In particular, the parser will entertain analyses of β that are incompatible with its analysis of α.

This noncommittal attitude does not seem to be shared by the human parser. For one thing, α can prime the parser towards certain structures. Consider the following contrast.

(1) a. The grave robber buried in the sand all the treasures he had stolen.
b. The mine buried in the sand exploded.

Even though *buried* can be a finite verb as well as a participle, the former is strongly preferred in example (1a). In (1b), on the other hand, the participle interpretation is favored.

Similarly, processing can be shown in self-paced reading experiments to slow down in ungrammatical sentences as soon as it becomes evident that the sentence cannot be salvaged anymore.

(2) * The grave robber in the sand all the treasures he had stolen.

This sentence can be recognized as ungrammatical once *all* is encountered. This inference is not made by the shift-reduce parser, however, which will continue to parse this sentence until all symbols have been read and all possible reductions have been carried out. A top-down parser, on the other hand, will always crash at *all* since none of its conjectured structures are compatible with *all* at this position in the sentence.

Exercise 5.7. Draw the parse history (i.e. the prefix tree of parse tables) for the unsuccessful shift-reduce parse of the sentence *the grave robber in the sand all the treasures*, given the grammar below.

1)	S	→	NP VP	6)	Det	→	the | all
2)	NP	→	N	7)	N	→	grave | grave robber | sand | treasures
3)	NP	→	Det N	8)	P	→	in
4)	NP	→	NP PP	9)	V	→	stole | sand
5)	VP	→	V				

How many steps does the parser spend on parses that a more predictive parser could have already identified as unsalvageable? ⊙

Lecture 6

Chart Parsing

Lecture 7

Left-Corner Parsing

Top-down parsers and bottom-up parsers each turned out to have both their advantages and disadvantages. Top-down parsers are purely predictive, the input string is only checked against fully built branches — those that end in a terminal symbol — but does not guide the prediction process itself. Bottom-up parsers are purely driven by the input string and lack any kind of predictiveness. In particular, a bottom-up parser may entertain analyses for the substring spanning from position i to j that are incompatible with the analysis for the substring from 0 to $i-1$. Neither behavior seems to be followed by the human parser. The parser is predictive, since ungrammatical sentences are recognized as such as soon as the structure becomes unsalvageable. At the same time, though, the prediction process is guided by the input seen so far. What we need, then, is a formal parsing model that integrates top-down prediction and bottom-up reduction. Left-corner parsing does exactly that.

1 Intuition

The ingenious idea of left-corner parsing is to restrict the prediction step such that the parser conjectures X only if there is already some bottom-up evidence for the existence of X. More precisely, the parser conjectures an XP only if a *possible left corner of X* has already been identified. The *left corner of a rewrite rule* is the leftmost symbol on the righthand side of the rewrite arrow (intuitively, the symbol the arrow points at). For instance, the left corner of NP → Det N is Det. Thus Y is a possible left corner of X only if the grammar contains a rewrite rule $X \rightarrow Y\ \gamma$. In this case, the parser may conjecture the existence of X and γ once it has reached Y in a bottom-up fashion.

Consider our familiar toy grammar.

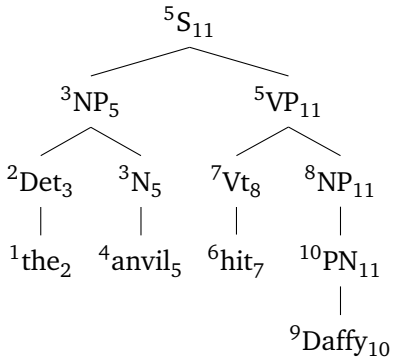
1)	S	→ NP VP	6)	Det	→ a | the
2)	NP	→ PN	7)	N	→ car | truck | anvil
3)	NP	→ Det N	8)	PN	→ Bugs | Daffy
4)	VP	→ Vi	9)	Vi	→ fell over
5)	VP	→ Vt NP	10)	Vt	→ hit

Rather than conjecturing S → NP VP right away, the parser has to wait until it has identified an NP before it can try to build an S. The NP, in turn, must be found bottom-up. This may involve a sequence of bottom-up reductions: read *Daffy*, reduce *Daffy* to PN, reduce PN to NP. Alternatively, it may involve a mixture of bottom-up reduction and left-corner condition prediction: read *the*, reduce to Det, use the rewrite rule NP

→ Det N in your top-down prediction, read *anvil*, reduce to N, reduce Det N to NP. Now that the NP has been identified, the parser may use S → NP VP in a prediction step.

string	**rule**	**predictions**
the	read input	
Det	Det → the	
	left-corner prediction	N to yield NP
anvil	read input	N to yield NP
N	N → anvil	N to yield NP
NP	complete prediction	VP to yield S
hit	read input	VP to yield S
Vt	Vt → hit	VP to yield S
	left-corner prediction	VP to yield S, NP to yield VP
Daffy	read input	VP to yield S, NP to yield VP
PN	PN → Daffy	VP to yield S, NP to yield VP
NP	NP → PN	VP to yield S, NP to yield VP
VP	complete prediction	VP to yield S
S	complete prediction	

The usual four way split between depth-first or breadth-first on the one hand and left-to-right versus right-to-left on the other makes little sense for left-corner parsers. The standard left-corner parser is depth-first left-to-right. A breadth-first left-corner parser behaves like a bottom-up parser if left-corner predictions are delayed, or like a depth-first left-corner parser if they apply as usual. And a right-to-left depth-first left-corner parser has no use for left-corner predictions since the predicted material has already been inferred in a bottom-up fashion anyways.



Exercise 7.1. What would the annotated trees look like for

- a left-to-right breadth-first left-corner parser where
  - reading a word can immediately be followed by a single reduction step,
  - reducing X to Y cannot be immediately followed by a left-corner prediction using Y.
- a left-to-right breadth-first left-corner parser where
  - reading a word can be immediately followed by a single reduction step,

  - reducing X to Y is immediately followed by a left-corner prediction using Y.

- a right-to-left depth-first left-corner parser. ⊙

2 Formal Specification

2.1 Standard Left-Corner Parser

Since the usual parameters make little sense for a left-corner parser, no control structure is needed and it suffices to define the parsing schema. The parser has to keep track of four distinct pieces of information:

- the current position in the string,
- any identified nodes l_i that have not been used up by any inference rules yet,
- which phrases $p_1, \ldots, p_n$ need to be built according to the left-corner prediction using some l_i, and
- which phrase is built from $l_i, p_i, \ldots, p_n$

Our items take the form $[i, \alpha \bullet \beta]$, where

- i is the current position in the string,
- α is the list of identified unused nodes, and
- β is a list of labeled lists of phrases to be built.

For instance, the item $[1, \bullet[_{\text{NP}}\ \text{N}]]$ encodes that if position 1 is followed by an N, we can build an NP.

The parser has a single axiom $[0, \bullet]$, and its goal is $[n, S\bullet]$. So the parser has to move from the initial to the last position of the string and end up identifying S. The parser uses five rules, four of which are generalizations of the familiar top-down and bottom-up rules.

$$\textbf{Shift} \qquad \frac{[i, \alpha \bullet \beta]}{[i+1, \alpha a \bullet \beta]}\ a = w_i$$

$$\textbf{Reduce} \qquad \frac{[i, \alpha\gamma \bullet \beta]}{[i, \alpha N \bullet \beta]}\ N \to \gamma \in R$$

$$\textbf{Scan} \qquad \frac{[i, \alpha N \bullet [_M\ N\gamma]\ \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}$$

$$\textbf{Predict} \qquad \frac{[i, \alpha N \bullet \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}\ M \to N\gamma \in R$$

$$\textbf{Complete} \qquad \frac{[i, \alpha \bullet [_M\ ]\ \beta]}{[i, \alpha M \bullet \beta]}$$

The shift rule reads in new input, and the reduce rule replaces the right-hand side of a rewrite rule by its left-hand side, thereby building structure in the usual bottom-up fashion. The scan rule eliminates a predicted symbol against an existing one, just like the top-down scan rule eliminates a predicted terminal symbol if a matching symbol can be found in the input at this position.[1]

The predict rule necessarily extends the prediction mechanism of a standard top-down parser since left-corner prediction proceeds both bottom-up, inferring the symbol to the left of the rewrite arrow, and top-down, inferring the sister nodes to the right. An existing left-corner N is removed, and instead we add to β a list that is labeled with the conjectured mother of N and contains the conjectured sisters of N. The completion rule, finally, states that once we've completely exhausted a list — i.e. all the conjectured siblings have been identified — the phrase that can be built from the elements in this list is promoted from a mere conjecture to a certainty, which technically amounts to pushing it to the left side of •.

Example 7.1 Left-corner parse of *The anvil hit Daffy*

parse item	**inference rule**
[0,•,]	axiom
[1,the •,]	shift
[1,Det •,]	reduce(6)
[1,•[$_{NP}$ N]]	predict(3)
[2,anvil •[$_{NP}$ N]]	shift
[2,N •[$_{NP}$ N]]	reduce(7)
[2,•[$_{NP}$]]	scan
[2,NP •]	complete
[2,•[$_{S}$ VP]]	predict(1)
[3,hit •[$_{S}$ VP]]	shift
[3,V •[$_{S}$ VP]]	reduce(10)
[3,•[$_{VP}$ NP] [$_{S}$ VP]]	predict(5)
[4,Daffy •[$_{VP}$ NP] [$_{S}$ VP]]	shift
[4,PN •[$_{VP}$ NP] [$_{S}$ VP]]	reduce(8)
[4,NP •[$_{VP}$ NP] [$_{S}$ VP]]	reduce(2)
[4,•[$_{VP}$] [$_{S}$ VP]]	scan
[4, VP •[$_{S}$ VP]]	complete
[4,•[$_{S}$]]	scan
[4,S •]	complete

[1] The scan rule of the recursive descent parser can decomposed into a shift rule and a second rule that closely mirrors the scan rule above:

$$\textbf{Shift} \quad \frac{[i, \bullet \beta]}{[i+1, a \bullet \beta]} \; a = w_i$$

$$\textbf{Scan} \quad \frac{[i, a \bullet a\beta]}{[i, \bullet \beta]}$$

The choice of • as a separator with identified material to the left and predicted material to the right is not accidental. Recall that the recursive descent parser is a purely predictive parser, and in all its parse items • occurred to the very left. So the predicted material was trivially to the right of •. Similarly, the shift reduce parser is completely free of any predictions, and the material built via shift and reduce was always to the left of •. Viewed from this perspective, the inference rules of the left-corner parser highlight its connections to top-down and bottom-up parsing (cf. Tab. 7.1).

	Top-Down	**Bottom-Up**	**Left-Corner**
Axiom	$[0, \bullet S]$	$[, 0]$	$[0, \bullet]$
Goal	$[n, \bullet]$	$[S\bullet, n]$	$[n, S\bullet]$
Scan	$\dfrac{[i, \alpha \bullet a\beta]}{[i+1, \alpha \bullet \beta]}$		$\dfrac{[i, \alpha N \bullet [_M\ N\gamma]\ \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}$
Shift		$\dfrac{[\alpha \bullet \beta, j]}{[\alpha a \bullet \beta, j+1]}$	$\dfrac{[i, \alpha \bullet \beta]}{[i+1, \alpha a \bullet \beta]}$
Predict	$\dfrac{[i, \alpha \bullet N\beta]}{[i, \alpha \bullet \gamma\beta]}$		$\dfrac{[i, \alpha N \bullet \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}$
Reduce		$\dfrac{[\alpha\gamma \bullet \beta, j]}{[\alpha N \bullet \beta, j]}$	$\dfrac{[i, \alpha\gamma \bullet \beta]}{[i, \alpha N \bullet \beta]}$
Complete			$\dfrac{[i, \alpha \bullet [_M\ ]\ \beta]}{[i, \alpha M \bullet \beta]}$

Table 7.1: Comparison of recursive descent, shift reduce, and left-corner parser

2.2 Generalized Left-Corner Parsing

The left-corner parser combines top-down and bottom-up in a specific manner: one symbol needs to be found bottom-up before a top-down prediction can take place. This weighting of bottom-up and top-down can be altered by changing the number of symbols that need to be present. That is to say, the left-corner of a rule is no longer just the leftmost symbol of its right side, but rather a prefix of the right side. For instance, if the number is increased to 2, then NP → Det A N could be used to predict N and NP only after Det and A have been identified. A left-corner parser where left corners are string of length 2 or more is called a *generalized left-corner parser*. It uses the same rules as a standard left-corner parser, except that the prediction rule is slightly modified.

$$\textbf{Predict} \qquad \frac{[i, \alpha\delta \bullet \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}\ M \to \delta\gamma \in R$$

Notice the close connection to bottom-up and top-down parsing. A bottom-up parser is a generalized left-corner parser that requires $\delta\gamma = \delta$, so M is predicted only if all its daughters have already been identified. In this case the prediction rule turns $[i, \alpha\delta \bullet \beta]$ into $[i, \alpha \bullet [_M]\beta]$, which the completion rule turns into $[i, \alpha M \bullet]$. The reduce rule is just a shorthand for running these two rules immediately one after another.

A top-down parser is similar to a generalized left-corner parser where δ is the empty string, so the prediction rule is never restricted by a left corner. This analogy is not completely right, however, because such a generalized left-corner parser can predict any rule at any given point, whereas the top-down parser must make predictions that are licit rewritings of non-terminal symbols in the parse items.

Still, generalized left-corner parsing is a straight-forward extension of left-corner parsing that allows altering how much weight is put on top-down prediction versus bottom-up confirmation.

3 Relation to Earley and Bottom-Up Parsing

4 Psycholinguistic Adequacy

Exercise 7.2. Show that just like top-down and bottom-up parsers, left-corner parsers struggle with garden path sentences. ⊙

Exercise 7.3. Recall the two structures that were proposed for *John's father's car's exhaust pipe disappeared* in exercises 5.4 and 5.5. Write down the left-corner parse tables for both structures. Based on these parse tables, annotate the trees with subscripts and superscripts in the usual fashion. Determine the payload as well as MaxTen and MaxSum. How does the left-corner parser fare in comparison to the recursive descent and shift reduce parsers? ⊙

Exercise 7.4. Are merely local syntactic coherence effects expected with a left-corner parser? ⊙

Lecture 8

Generalizing Parsers via Monoids and Semirings

Lecture 9

Parsing Beyond Context-Free Grammars

Lecture 10

A Context-Free Top-Down Parser for Minimalist Grammars

Minimalist grammars are more powerful than CFGs, they generate mildly context-sensitive string languages, which are taken to be a good approximation of natural languages. They are also very malleable and can be used as a formal basis for the overwhelming majority of analyses in the generative literature. The question, however, is how MGs can be parsed. In general, more powerful grammar formalisms require more sophisticated parsing strategies — the higher expressivity comes at the cost of increased complexity. As we will see today, however, MGs can actually be represented in terms of CFGs, which makes it a lot easier to design MG parsers based on the CFG parsers we already know.

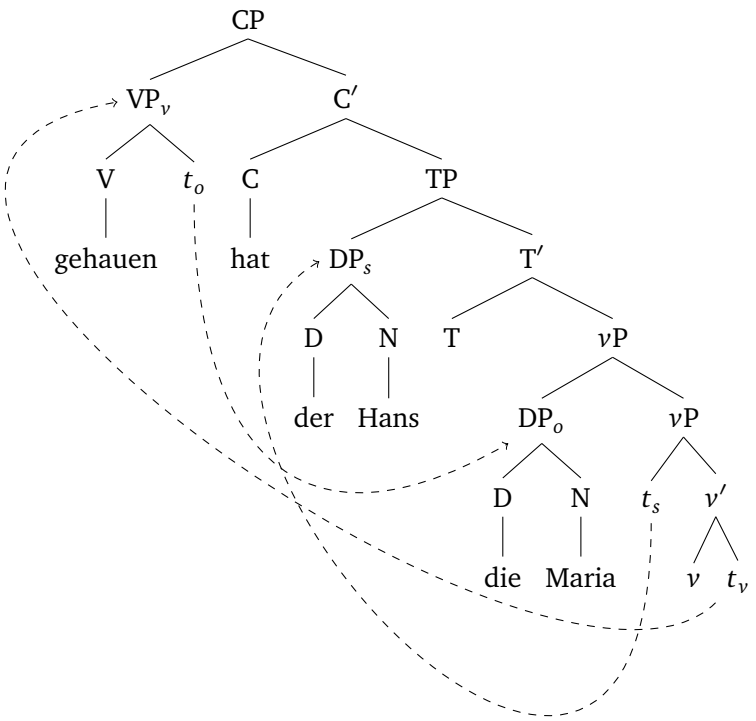
1 MG Derivations are Context-Free

The main difference between MG phrase structure trees and their corresponding derivation trees is that the latter only indicate when movement takes place, but no subtrees are actually being displaced. Surprisingly, this minor difference has a big effect on the complexity of these two data structures. Not all phrase structure trees can be generated by a context-free grammar — if this were the case, MGs would not be any more powerful than CFGs. MG derivation trees, however, are context-free.

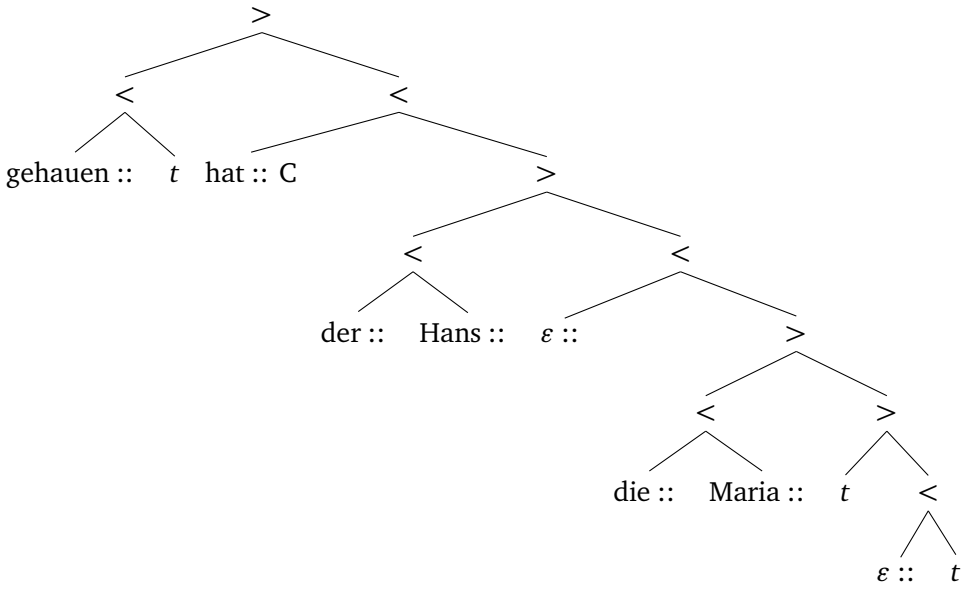
Consider the following sentence from Bavarian German, which displays topicalization of the V-head.

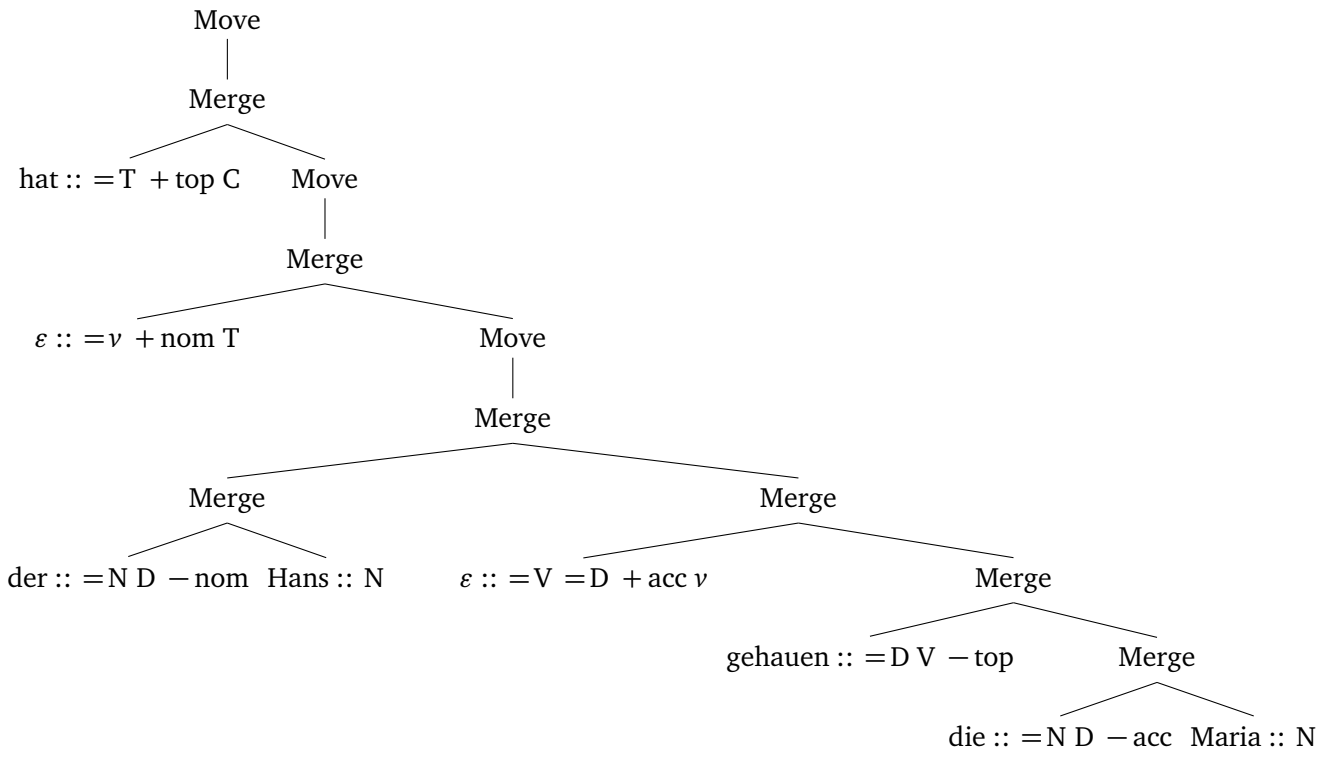
(1) Gehauen hat der Hans die Maria (, nicht geküsst).
beaten has the Hans the Maria (, not kissed).
'Hans beat Mary (he didn't kiss her).'

Since there are independent reasons to believe that the position before the finite verb can only be filled by phrases, the entire VP must have moved rather than the V-head itself. But since the object DP *die Maria* is part of the VP, it must have moved to a higher position outside the VP before the VP moved into the topic position. Given standard Minimalist assumptions — DP analysis of noun phrases, Larsonian shells, and the C-T-*v*-V clause spine — the phrase structure tree thus should be similar to the one below (for the sake of simplicity we ignore head movement of the auxiliary verb from *v* to T and C).

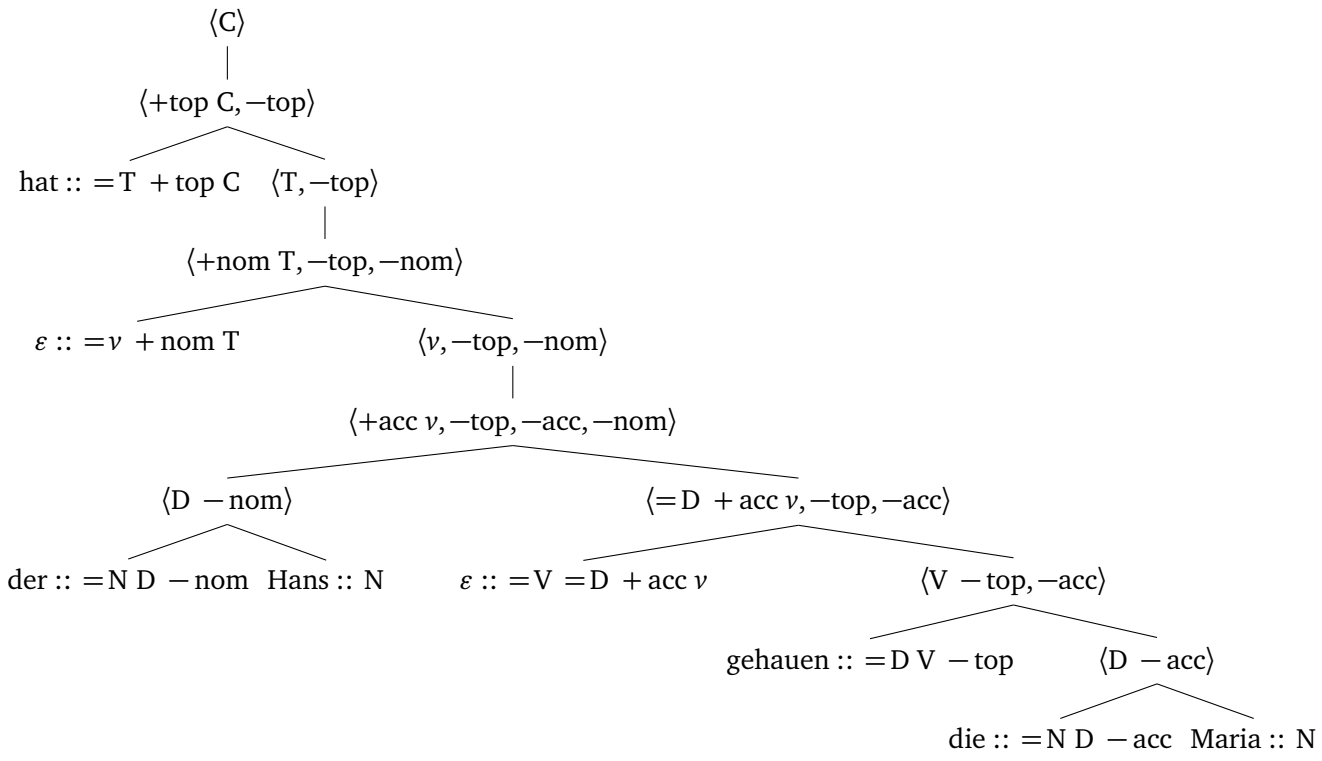


We can directly replicate this analysis with MGs.





At a quick glance it is actually hard to tell whether this derivation is well-formed. That's because the derivation tree does not directly keep track of which features are still unchecked at each point of the derivation. However, we can add this information by annotating every interior node with a tuple, each component of which is the list of unchecked features of some lexical item.



Notice how the feature-annotated derivation tree can easily be described by a small number of rewrite rules:

$$\begin{aligned}
\langle \text{C} \rangle &\rightarrow \langle +\text{top C}, -\text{top} \rangle \\
\langle +\text{top C}, -\text{top} \rangle &\rightarrow \langle =\text{T } +\text{top C} \rangle \ \langle \text{T}, -\text{top} \rangle \\
\langle \text{T}, -\text{top} \rangle &\rightarrow \langle +\text{nom T}, -\text{top}, -\text{nom} \rangle \\
\langle +\text{nom T}, -\text{top}, -\text{nom} \rangle &\rightarrow \langle =v\ +\text{nom T} \rangle \ \langle v, -\text{top}, -\text{nom} \rangle \\
\langle v, -\text{top}, -\text{nom} \rangle &\rightarrow \langle +\text{acc } v, -\text{top}, -\text{acc}, -\text{nom} \rangle \\
\langle +\text{acc } v, -\text{top}, -\text{acc}, -\text{nom} \rangle &\rightarrow \langle \text{D } -\text{nom} \rangle \ \langle =\text{D } +\text{acc } v, -\text{top}, -\text{acc} \rangle \\
\langle \text{D } -\text{nom} \rangle &\rightarrow \langle =\text{N D } -\text{nom} \rangle \ \langle \text{N} \rangle \\
\langle =\text{D } +\text{acc } v, -\text{top}, -\text{acc} \rangle &\rightarrow \langle =\text{V } =\text{D } +\text{acc } v \rangle \ \langle \text{V } -\text{top}, -\text{acc} \rangle \\
\langle \text{V } -\text{top}, -\text{acc} \rangle &\rightarrow \langle =\text{D V } -\text{top} \rangle \ \langle \text{D } -\text{acc} \rangle \\
\langle \text{D } -\text{acc} \rangle &\rightarrow \langle =\text{N D } -\text{acc} \rangle \ \langle \text{N} \rangle
\end{aligned}$$

$$\begin{aligned}
\langle =\text{T } +\text{top C} \rangle &\rightarrow \text{hat} \\
\langle =v\ +\text{nom T} \rangle &\rightarrow \varepsilon \\
\langle =\text{V } =\text{D } +\text{acc } v \rangle &\rightarrow \varepsilon \\
\langle =\text{D V } -\text{top} \rangle &\rightarrow \text{gehauen} \\
\langle =\text{N D } -\text{nom} \rangle &\rightarrow \text{der} \\
\langle =\text{N D } -\text{acc} \rangle &\rightarrow \text{die} \\
\langle \text{N} \rangle &\rightarrow \text{Hans} \mid \text{Maria}
\end{aligned}$$

While these rewrite rules are incredibly difficult to make sense of for humans, they generate the desired derivation tree (with the minor difference that for the sake of readability every lexical item has been split into two nodes, with the phonetic exponent as the daughter of the feature component). In fact, there is a fully automatic procedure for converting an MG G with lexicon *Lex* into a CFG C such that C generates all well-formed derivation trees of G, and only those (this result hinges on the SMC; see Kobele et al. 2007 and section 2.1.3 of Graf 2013).

Clearly every phrase structure tree is fully described by its derivation tree (or trees, if there are multiple ways of building and one and the same phrase structure tree), since the latter is a set of instructions for building the former. This means that the structure of a sentence is completely specified by assigning it a derivation tree. So even though an MG parser has to work for string languages that are not context-free, it still only has to assign each sentence a context-free structure – the derivation trees. As we will see next, this idea works fairly well, but there is one major complication: the string yield of a derivation tree does not correspond to the sentence being parsed because movement can change the order of words.

2 Top-Down Parser with Feature Annotated Derivations

2.1 MGs without Movement — A Naive Top-Down Parser

For an MG without movement — i.e. an MG where no lexical item has any licensee features — designing a parser is straight-forward. In this case, the order of the leaves in the derivation tree can be taken to mirror the order of the words in the input sentence. So if MG G is movement-free, we can translate it into a CFG that generates G's derivation trees and use any one of our familiar parsers for this CFG.

But let's see if we can design a parser that reflects the MG feature calculus more directly. If there are only Merge nodes in the derivation, every interior node has exactly

two daughters. In addition, all the features of the node must have been contributed by exactly one of its daughters, namely the one with the selector feature (take a minute to convince yourself that this is indeed the case!). Since the selector may be linearized either to the left or to the right, we need two distinct inference rules.

$$\textbf{Merge1} \qquad \frac{\langle \alpha \rangle}{\langle =\mathrm{F}\ \alpha \rangle \qquad \langle \mathrm{F} \rangle} \; F \text{ a feature name of } G$$

$$\textbf{Merge2} \qquad \frac{\langle \alpha \rangle}{\langle \mathrm{F} \rangle \qquad \langle =\mathrm{F}\ \alpha \rangle} \; F \text{ a feature name of } G$$

There is one minor problem with those rules, though, and that's that the selector should always be to the left of its first argument, and always to the right of its other arguments. Our rules do not capture this fact. For example, if α is C then the parser could linearize the complementizer to the left or to the right of TP. But an MG would only entertain the first option. The simplest fix is to annotate each selector feature with information about how the argument is linearized, and that's the solution we will use for now.

$$\textbf{Merge Right} \qquad \frac{\langle \alpha \rangle}{\langle =\mathrm{F}_r\ \alpha \rangle \qquad \langle \mathrm{F} \rangle} \; \mathrm{F} \text{ a feature name of } G$$

$$\textbf{Merge Left} \qquad \frac{\langle \alpha \rangle}{\langle \mathrm{F} \rangle \qquad \langle =\mathrm{F}_l\ \alpha \rangle} \; \mathrm{F} \text{ a feature name of } G$$

The only thing remaining, then, is a rule for the removal of lexical items, similar to the scan rule.

$$\textbf{LI} \qquad \frac{\langle \alpha \rangle}{\mathrm{w} :: \alpha} \; \mathrm{w} :: \alpha \in Lex_G$$

2.2 Recursive Descent Parser for Movement-Free MGs

The previous discussion is sufficient to outline the feature-based logic of an MG parser, but it does not serve as a parsing schema. Just like in recursive descent parsing, our parsing items need to keep track of all the non-terminal symbols and put the in the right linear order. This, however, is very simply to do. For a recursive-descent MG parser, the only axiom is $[0, \bullet \langle \mathrm{C} \rangle]$, and the only goal is $[n, \bullet]$.

$$\textbf{Merge Left} \qquad \frac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i, \bullet \langle \mathrm{F} \rangle \langle =\mathrm{F}_l\ \alpha \rangle \beta, j]} \; \mathrm{F} \text{ a feature name of } G$$

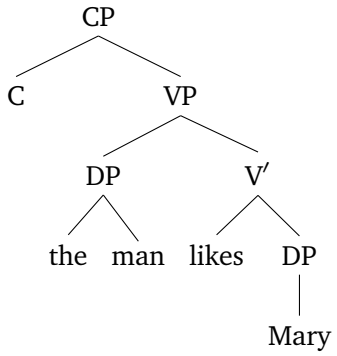
$$\textbf{Merge Right} \qquad \frac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i, \bullet \langle =\mathrm{F}_r\ \alpha \rangle \langle \mathrm{F} \rangle \beta, j]} \; \mathrm{F} \text{ a feature name of } G$$

$$\textbf{LI (pronounced)} \qquad \frac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i+1, \bullet \beta, j]} \; w_i :: \alpha \in Lex_G,\ w_i \neq \varepsilon$$

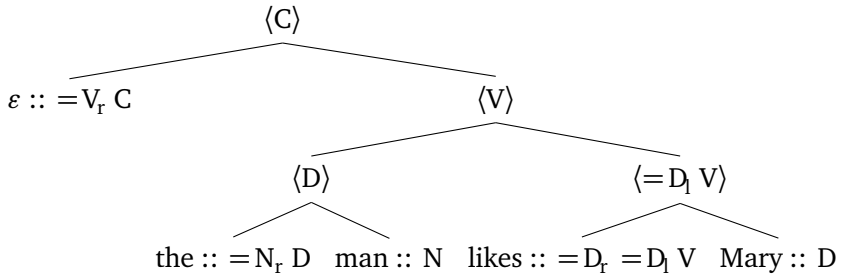
LI (empty) $\dfrac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i, \bullet \beta, j]}\ w_i :: \alpha \in Lex_G, w_i = \varepsilon$

Example 10.1 MG Top-Down Parse of *The man likes Mary*

Suppose the sentence *The man likes Mary* has the structure below.



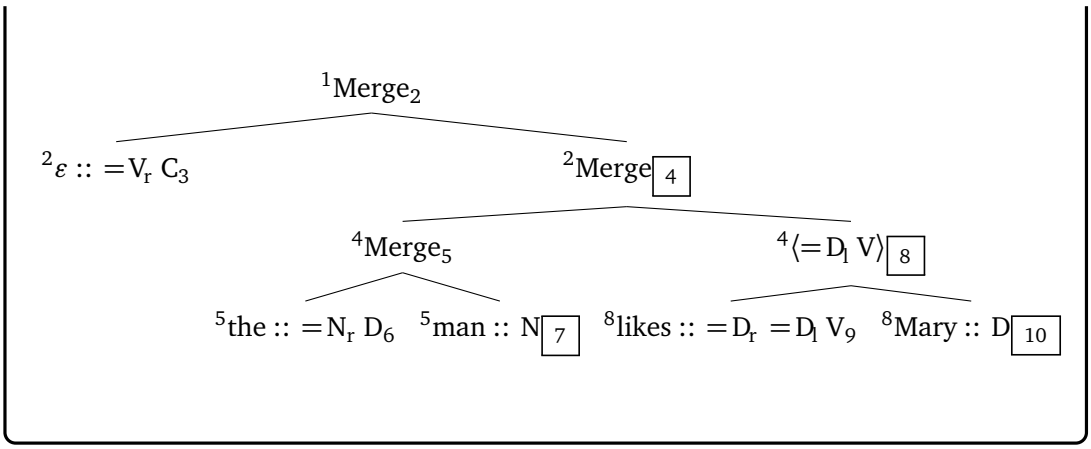
The feature-annotated Minimalist derivation tree for this sentence looks as follows.



The parse table directly reflects this structure.

parse item	inference rule
$[0, \bullet\langle C\rangle]$	axiom
$[0, \bullet\langle =V_r\ C\rangle\ \langle V\rangle]$	Merge Right
$[0, \bullet\langle V\rangle]$	LI (empty)
$[0, \bullet\langle D\rangle\ \langle =D_l\ V\rangle]$	Merge Left
$[0, \bullet\langle =N\ D\rangle\ \langle N\rangle\ \langle =D_l\ V\rangle]$	Merge Right
$[1, \bullet\langle N\rangle\ \langle =D_l\ V\rangle]$	LI (pronounced)
$[2, \bullet\langle =D_l\ V\rangle]$	LI (pronounced)
$[2, \bullet\langle =D_r =D_l\ V\rangle\ \langle D\rangle]$	Merge Right
$[3, \bullet\langle D\rangle]$	LI (pronounced)
$[4, \bullet]$	LI (pronounced)

We can annotate the final derivation tree as usual to indicate how it is built by the parser.



2.3 Adding Movement

In principle, movement inference rules aren't all that different from Merge inference rules: we have a given expression of features, and infer which expressions this could have been produced from. But there are three important differences:

- Move is a unary rule, so we go from $\langle\alpha\rangle$ to $\langle\beta\rangle$, where β differs minimally from α,
- a tuple can now contain multiple feature strings, all but one of which are sequences of licensee features,
- we need a way to incorporate how Move changes the linear order of lexical items.

Let's deal with points 1 and 2 first. In general, our feature tuples will now have the form $\langle\alpha,\beta_1,\ldots,\beta_n\rangle$, where each β_i is a string of licensee features. This means that we have to slightly change our Merge rules.

Merge Right $\dfrac{\langle\alpha,\beta_1,\ldots,\beta_n\rangle}{\langle =\mathrm{F}_r\ \alpha,\gamma_1,\ldots,\gamma_k\rangle \qquad \langle \mathrm{F}\delta,\gamma_{k+1},\ldots,\gamma_n\rangle}$ F a feature name of G

Merge Left $\dfrac{\langle\alpha,\beta_1,\ldots,\beta_n\rangle}{\langle \mathrm{F}\delta,\gamma_1,\ldots,\gamma_k\rangle \qquad \langle =\mathrm{F}_l\ \alpha,\gamma_{k+1},\ldots,\gamma_n\rangle}$ F a feature name of G

Notice that we now distribute the strings of licensee features over the two daughters of a Merge node in a non-deterministic fashion. That is to say, we require in both rules that

- for all $1 \leq i,j \leq n$, the first features of β_i and β_j are distinct,
- if $\delta = \varepsilon$, then each β_i is identical to exactly one γ_j, and *vice versa*,
- otherwise, there is some β_i such that $\beta_i = \delta$ and the previous condition holds for all β_j with $j \neq i$.

The logic of the movement rule is even simpler.

$$\textbf{Move} \qquad \frac{\langle \alpha, \beta_1, \ldots, \beta_i, \ldots, \beta_n \rangle}{\langle +\text{f}\ \alpha, \beta_1, \ldots, -\text{f}\ \beta_i, \ldots, \beta_n \rangle} \; -\text{f a licensee feature of } G$$

Notice that β_i may be empty. In that case, we are dealing with the final landing site of whichever lexical item hosts the −f feature. Strictly speaking, then, the item in the antecedent line of the rule would have two adjacent commas with nothing inbetween the two since β_i is empty. Our rules never create an item of this form, so the Move rule couldn't apply in this case. Hence we should actually distinguish two cases of movement.

$$\textbf{Move (intermediate)} \qquad \frac{\langle \alpha, \beta_1, \ldots, \beta_i, \ldots, \beta_n \rangle}{\langle +\text{f}\ \alpha, \beta_1, \ldots, -\text{f}\ \beta_i, \ldots, \beta_n \rangle} \; -\text{f a licensee feature of } G$$

$$\textbf{Move (final)} \qquad \frac{\langle \alpha, \beta_1, \ldots, \beta_{i-1}, \beta_{i+1} \ldots, \beta_n \rangle}{\langle +\text{f}\ \alpha, \beta_1, \ldots, \beta_{i-1}, -\text{f}, \beta_{i+1} \ldots, \beta_n \rangle} \; -\text{f a licensee feature of } G$$

These movement rules do not take care of the true challenge posed by movement however, and that's keeping track of the order of words in the sentence, which no longer corresponds to the order of words in the derivation tree. Assume that we conjecture a Move node for topicalization. Then we have to keep track of the fact that we expect to see an LI later on, which is handled by the presence of −f or −f β_i in the tuples. But we also need to know that once we have found this item, the phrase it is a head of should match a sequence of words at the position where we initially conjectured topicalization movement.

2.4 Keeping Track of Conjectured Movers

In order to keep track of which positions in the string are associated with movers, we use place holders of the form $[f]$, where f is a feature name. These place holders are introduced by Move rules and allow us to reorder the tuples in a parse item where possible. Given a substring ϕ of a parse item, $\phi_{/[f_1 \cdots f_n]}$ is the result of removing the placeholders $[f_1], \ldots, [f_n]$ from ϕ.

Once again the axiom is $[0, \bullet \langle \text{C} \rangle]$ and the goal is $[n, \bullet]$. We need 4 Merge rule, 2 Move rules, and a few ancillary rules. The Merge rules are parameterized according to linearization and whether the merged item is a mover.

$$\textbf{Merge Right} \qquad \frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, \beta_n \rangle \psi]}{[i, \phi \bullet \langle =\text{F}_r\ \alpha, \gamma_1, \ldots, \gamma_k \rangle \langle \text{F}\delta, \gamma_{k+1}, \ldots, \gamma_n \rangle \psi]}$$

$$\textbf{Merge Left} \qquad \frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, \beta_n \rangle \psi]}{[i, \phi \bullet \langle \text{F}\delta, \gamma_{k+1}, \ldots, \gamma_n \rangle \langle =\text{F}_l\ \alpha, \gamma_1, \ldots, \gamma_k \rangle \psi]}$$

Merge Mover Right

$$\frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, f_1 \cdots f_m, \beta_n \rangle \psi]}{[i, \phi_{/[f_1 \cdots f_{m-1}]} \bullet \langle =\text{F}_r\ \alpha, \gamma_1, \ldots, \gamma_{k-1} \rangle [_{f_m} \langle \text{F}\ f_1 \cdots f_m, \gamma_{k+1}, \ldots, \gamma_n \rangle] \psi_{/[f_1 \cdots f_{m-1}]}]}$$

Merge Mover Left

$$\frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, f_1 \cdots f_m, \ldots \beta_n \rangle \psi]}{[i, \phi_{/[f_1 \cdots f_{m-1}]} \bullet [_{f_m} \langle \mathrm{F} f_1 \cdots f_n, \gamma_{k+1}, \ldots, \gamma_n \rangle] \langle = \mathrm{F}_l \ \alpha, \gamma_1, \ldots, \gamma_{k-1} \rangle \psi_{/[f_1 \cdots f_{m-1}]}]}$$

The f-subscripted brackets around a tuple mark it as an f-mover, which is necessary to identify it with the right placeholder in ϕ or ψ. Notice also that we eliminate all placeholders for intermediate movement as soon as the mover is merged. Since the mover will never occur at an intermediate landing site — after all, it has to move to its final target position — those placeholders are string vacuous. However, we have to introduce them first to make sure that the feature checking requirements are satisfied: every non-final licensee feature needs a corresponding move node, which corresponds to a placeholder in the parse item.

Move (intermediate) $$\frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, \beta_i, \ldots, \beta_n \rangle \psi]}{[i, \phi[\mathrm{f}] \bullet \langle +\mathrm{f}\ \alpha, \beta_1, \ldots, -\mathrm{f}\ \beta_i, \ldots, \beta_n \rangle \psi]}$$

Move (final) $$\frac{[i, \phi \bullet \langle \alpha, \beta_1, \ldots, \beta_{i-1}, \beta_{i+1} \ldots, \beta_n \rangle \psi]}{[i, \phi[\mathrm{f}] \bullet \langle +\mathrm{f}\ \alpha, \beta_1, \ldots, \beta_{i-1}, -\mathrm{f}, \beta_{i+1} \ldots, \beta_n \rangle \psi]}$$

The Move rules are almost exactly the same, except that Move (intermediate) extends an existing β_i, whereas Move (final) adds a completely new one.

The actual reordering of tuples in a parse item is handled by the rule displace, which takes a mover and puts it in the position of the placeholder.

Displace Left $$\frac{[i, \phi[f]\phi' \bullet [_f \alpha]\psi]}{[i, \phi \bullet \alpha\phi'\psi]}$$

Displace Right $$\frac{[i, \phi \bullet [_f \alpha]\psi[f]\psi']}{[i, \phi \bullet \psi\alpha\psi']}$$

The LI rule replaces a feature tuple by an LI with that feature specification, and the scan rule eliminates LIs from the parse items. Scanning an LI is allowed only if it is at the very left edge of the parse item.

LI $$\frac{[i, \phi \bullet \langle \alpha \rangle \psi]}{[i, \phi \bullet a\psi]}\ a :: \alpha \in Lex_G$$

Scan $$\frac{[i, \bullet a\beta]}{[k, \bullet\beta]}\ k = i \text{ if } a = \varepsilon \text{ and } i + 1 \text{ otherwise}$$

The shift rule moves $\bullet$ to the right if no other rule can be applied.

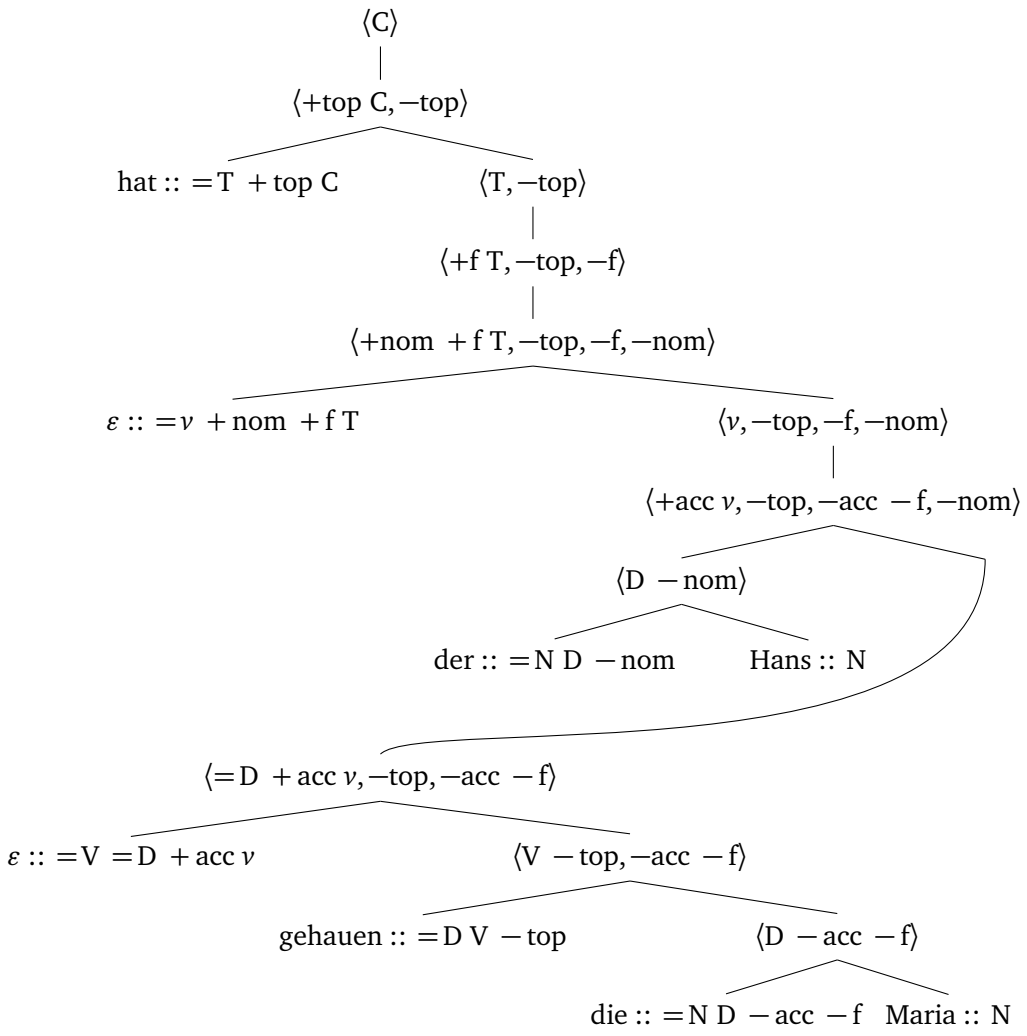
Shift $$\frac{[i, \phi \bullet x\psi]}{[i, \phi x \bullet \psi]}\ \phi \text{ not the empty string, } x \text{ a placeholder or an LI}$$

Example 10.2 MG Top-Down Parse of *Gehauen hat die Maria der Hans*

Let's look at a slightly more complicated version of our very first example sentence.

(2) Gehauen hat die Maria der Hans.
beaten has the Maria the Hans

Here the object *die Maria* not only moves out of the VP, but it also scrambles across the subject *der Hans* afterwards. A simplified derivation tree is given below.

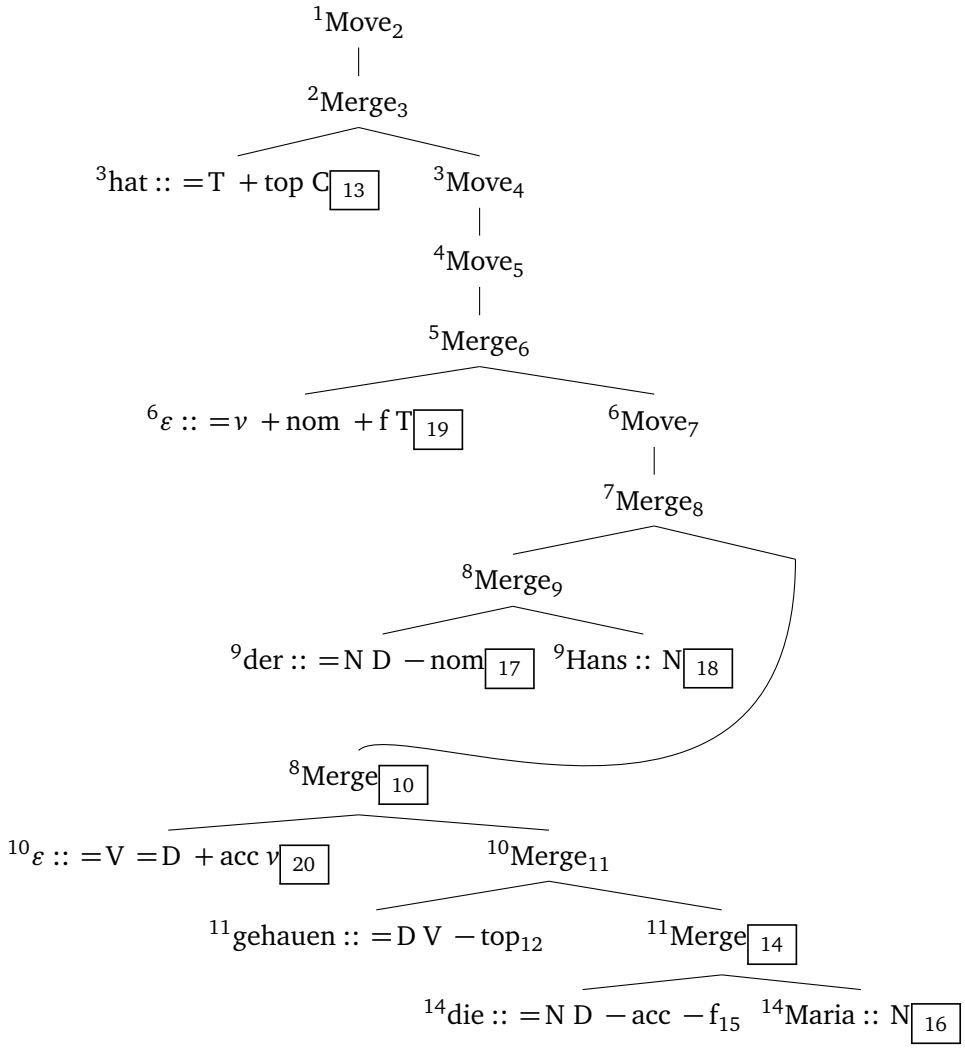


The parser infers the derivation with the following steps.

parse item	inference rule
[0, •⟨C⟩]	axiom
[0, [top] •⟨+top C, −top⟩]	Move (final)
[0, [top] •⟨=T +top C⟩ ⟨T, −top⟩]	Merge Right
[0, [top] hat •⟨T, −top⟩]	LI & Shift
[0, [top] hat [f] •⟨+f T, −top, −f⟩]	Move (final)
[0, [top] hat [f] [nom]	

$\bullet\langle$+nom +f T, −top, −f, −nom$\rangle]$	Move (final)
[0, [top] hat [f] [nom] $\bullet\langle = v$ +nom +f T$\rangle$ $\langle v$, −top, −f, −nom$\rangle]$	Merge Right
[0, [top] hat [f] [nom] ε $\bullet\langle v$, −top, −f, −nom$\rangle]$	LI & Shift
[0, [top] hat [f] [nom] ε [acc] $\bullet\langle$+acc v, −top, −acc −f, −nom$\rangle]$	Move (intermediate)
[0, [top] hat [f] [nom] ε [acc] $\bullet[_{\text{nom}}$ $\langle$D −nom$\rangle]$ $\langle =$D +acc v, −top, −acc −f$\rangle]$	Merge Mover Left
[0, [top] hat [f] $\bullet\langle$D −nom$\rangle$ ε [acc] $\langle =$D +acc v, −top, −acc −f$\rangle]$	Displace Left
[0, [top] hat [f] $\bullet\langle =$N D −nom$\rangle$ $\langle$N$\rangle$ ε [acc] $\langle =$D +acc v, −top, −acc −f$\rangle]$	Merge Right
[0, [top] hat [f] der $\bullet\langle$N$\rangle$ [f] [nom] ε [acc] $\langle =$D +acc v, −top, −acc −f$\rangle]$	LI & Shift
[0, [top] hat [f] der Hans $\bullet\varepsilon$ [acc] $\langle =$D +acc v, −top, −acc −f$\rangle]$	LI & Shift
[0, [top] hat [f] der Hans ε [acc] $\bullet\langle =$D +acc v, −top, −acc −f$\rangle]$	Shift*2
[0, [top] hat [f] der Hans ε [acc] $\bullet\langle =$V =D +acc $v\rangle$ $[_{\text{top}}$ $\langle$V −top, −acc −f$\rangle]]$	Merge Mover Right
[0, [top] hat [f] der Hans ε [acc] ε $\bullet[_{\text{top}}$ $\langle$V −top, −acc −f$\rangle]]$	LI & Shift
[0, $\bullet\langle$V −top, −acc −f$\rangle$ hat [f] der Hans ε [acc] $\varepsilon]$	Displace Left
[0, $\bullet\langle =$D V −top$\rangle$ $[_{\text{f}}$ $\langle$D −acc −f$\rangle]$ hat [f] der Hans ε $\varepsilon]$	Merge Mover Right
[0, •gehauen $[_{\text{f}}$ $\langle$D −acc −f$\rangle]$ hat [f] der Hans ε $\varepsilon]$	LI
[1, $\bullet[_{\text{f}}$ $\langle$D −acc −f$\rangle]$ hat [f] der Hans ε $\varepsilon]$	Scan
[1, •hat $\langle$D −acc −f$\rangle$ der Hans ε $\varepsilon]$	Displace Right
[2, $\bullet\langle$D −acc −f$\rangle$ der Hans ε $\varepsilon]$	Scan
[2, $\bullet\langle =$N D −acc −f$\rangle$ $\langle$N$\rangle$ der Hans ε $\varepsilon]$	Merge Right
[2, •die $\langle$N$\rangle$ der Hans ε $\varepsilon]$	LI
[3, $\bullet\langle$N$\rangle$ der Hans ε $\varepsilon]$	Scan
[3, •Maria der Hans ε $\varepsilon]$	LI
[4, •der Hans ε $\varepsilon]$	Scan
[5, •Hans ε $\varepsilon]$	Scan
[6, $\bullet\varepsilon$ $\varepsilon]$	Scan
[6, $\bullet\varepsilon]$	Scan
[6, •]	Scan

We can represent this parse more succinctly by annotating the derivation tree with indices in the usual fashion.



2.5 Two Issues

The parser as defined in the previous section has two issues. First, the Merge and Move rules are only limited by the requirement that the features F or f must be valid features of the grammar. This, however, can still lead to conjecturing tuples that could never be derived from the LIs of the grammar. These parses will eventually fail, of course, but for efficiency reasons it would be nice if the parser would not conjecture items that can never occur in a well-formed derivation.

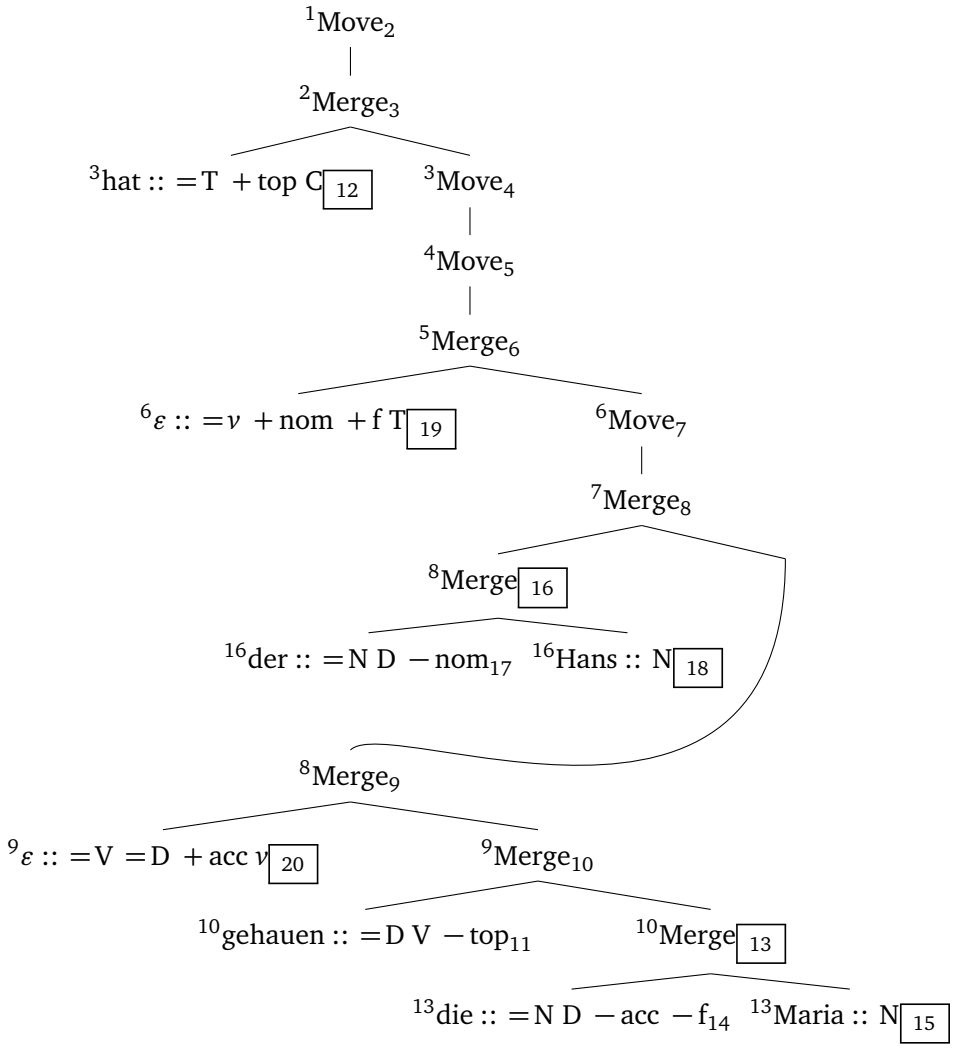
If we translate the MG into a CFG, the set of valid tuples corresponds to the set non-terminals of the CFG. So with a little bit of processing, the rules can be limited to only use non-terminals of this CFG. It would be more appealing, however, if we could

phrase our inference rules in a way so that they operate directly on the lexicon of our MG. This is exactly what is done in Stabler (2012).

Stabler's (2012) parser also fixes another problem with the current model. Right now, our parser does not sufficiently prioritize the search for movers. Since scanning of an LI l is delayed until all movers to the left of l have been found, memory usage would be minimized by searching for those movers. So once the parser stipulates the presence of a top-mover in the example above, it should first follow the branches that it believes will lead it to this mover. Once this mover has been found, it can scan *hat* and then continue it's search for an f-mover, and then for a nom-mover. Instead, our parser starts building the nom-mover before it has even found the top-mover; this increases the memory burden because there is no way the nom-mover could be fully scanned at this point.

Example 10.3 A More Efficient Traversal of the Derivation Tree

Using the search strategy outlined in Stabler (2012), the derivation tree from the previous example would be explored in the following order.



The Stabler parser has a lower payload because it does not fully expand the subject right away and instead moves on to the VP first. If the subject contained more than two LIs, the difference in payload would be even bigger. Maximum tenure stays the same because it is incurred at the T-head, which is introduced before the two parser diverge in their search path and cannot be scanned until the subject has been built and scanned. For this reason, the differences between the two parsers do not affect the node's tenure. We see a marked difference in summed tenure, however, because of the tenure contributed by *der* and *Hans*.

Metric	CF Parser	Stabler Parser
Payload	8	7
MaxTen	13	13
SumTen	57	48

Exercise 10.1. Draw a movement-free Minimalist derivation tree for *The anvil hit Daffy*. Assume a C-T-V spine where the subject is merged as the second argument of the T-head. The write down the parse table and annotate the derivation tree accordingly. Does the parser's behavior differ noticeably from what we observed for the recursive descent parser in Chapter 3? ⊙

Exercise 10.2. Following up on the previous exercise, assume that we actually have a C-T-v-V spine where the subject starts in Spec,vP and then moves to Spec,TP. Once again you have to write down the parse table and annotate the derivation tree. What differences do you observe? ⊙

Exercise 10.3. Draw Minimalist derivation trees for our left-embedding and right-embedding analyses of *John's father's car's exhaust pipe* disappeared. Assume a C-T-v-V spine where the subject moves from Spec,vP to Spec,TP. Then write down the parse table for each structure. Annotate the derivation trees according to how our parser would build them. Is the behavior of our parser similar to that of the recursive descent parser? ⊙

References and Further Reading

Graf, Thomas. 2013. *Local and transderivational constraints in syntax and semantics*. Doctoral Dissertation, UCLA.

Kobele, Gregory M., Christian Retoré, and Sylvain Salvati. 2007. An automata-theoretic approach to minimalism. In *Model Theoretic Syntax at 10*, ed. James Rogers and Stephan Kepser, 71–80.

Stabler, Edward P. 2012. Bayesian, minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5:611–633.

Lecture 11

A Move-Eager Parser for MGs

The context-free parser for MGs has two disadvantages. First, it freely infers feature tuples, with the only restriction being that the features of these tuples are features of the grammar. Consequently, the parser may conjecture tuples that never occur in a well-formed derivation of the grammar, which is a major waste of resources. Second, its search through the tree is not contingent on conjectured landing sites. The parser follows a simple pattern of "specifiers before complements" pattern, with the minor twist that if a mover is found, the parser inserts it at the final target site and continues parsing from there. This means that the parser has to store a lot of material in memory. If it progressed on the most direct path to where it expects the mover to originate, without building any of the structure along the other paths, it may safe quite some resources. Let's call the first type of parser *Merge-eager*, and the second *Move-eager*. The top-down MG parser defined in Stabler (2012) is Move-eager and operates directly on the lexicon, which prevents it from conjecturing useless feature tuples.

1 A Closer Look at the Stabler Parser

See the paper for details, in particular the appendix. The crucial insights are as follows:

- Every Minimalist lexicon can be represented as a *prefix tree*, where the prefixes correspond to the feature components of lexical items read from right to left.
- The prefix tree guides the parser in the structure building process.
- In order to deal with non-determinism, the parser keeps multiple parses in memory.
- Parses are ordered by their probability, and those with a probability below a fixed threshold p are discarded. This is called *beam parsing*.
- Since the probability of a tree decreases with every level of embedding, left branch recursion is no longer a problem for the parser. Instead of adding more and more levels of embedding *ad infinitum*, the parser eventually reaches the probability threshold p and conjectures no further levels.

2 Psycholinguistic Adequacy

2.1 Generalizations about Relative Clauses

Two major properties of relative clauses are firmly established in the literature (see Gibson 1998 and references therein).

- **SC/RC < RC/SC**
  A sentential complement containing a relative clause is easier to process than a relative clause containing a sentential complement.
- **SubjRC < ObjRC**
  A relative clause containing a subject gap is easier to parse than a relative clause containing an object gap.

These generalizations were obtained via self-paced reading experiments and ERP studies with minimal pairs such as (1) and (2), respectively.

(1) a. The fact $[_{SC}$ that the employee$_i$ $[_{RC}$ who the manager hired $t_i]$ stole office supplies] worried the executive.
    b. The executive$_i$ $[_{RC}$ who the fact $[_{SC}$ that the employee stole offices supplies] worried $t_i]$ hired the manager.

(2) a. The reporter$_i$ $[_{RC}$ who t_i attacked the senator] admitted the error.
    b. The reporter$_i$ $[_{RC}$ who the senator attacked $t_i]$ admitted the error.

We can use these sentences as test cases to determine whether the two types of MG parsers — Merge-eager and Move-eager — can account for these discrepancies, and if so, which one of them.

2.2 Refining our Metrics

In previous evaluations of parser predictions we used three metrics: payload for the number of nodes kept in memory, MaxTen for the maximum number of steps a node is kept in memory, and SumTen as the sum of all non-trivial tenure. For the MG-parser, it makes sense to introduce further subtypes of these. Since MGs have empty heads, the linguistic status of which is contentious, each metric can have two values, the standard one computed over all nodes, and a more restricted one that ignores all nodes that are empty heads. For the sake of succinctness we write these values in the slashed format m/n, where m is the standard value and n the restricted one. Another distinction that might be useful is the one between lexical items and phrasal nodes, which is why each metric also has a subtype that only considers leafs in the derivation. These subtypes are indicated by a subscripted *Lex*. Overall, then, we have four metrics and each metric has a slashed value.

Payload number of all/pronounced nodes kept in memory for at least 3 steps (this is increased from our previous default of 2, following Graf and Marcinek 2014)

Payload$_{Lex}$ number of all/pronounced leaves kept in memory for at least 3 steps

Max greatest number of steps that any/some pronounced node is kept in memory

Max$_{Lex}$ greatest number of steps that any/some pronounced leaf is kept in memory

These metrics are taken from Graf and Marcinek (2014), where they are called **Box**,**BoxLex**, **Max**, and **MaxLex**, respectively.

2.3 Sentential Complements and Relative Clauses

The Merge-eager and Move-eager derivation trees for (1a) and (1b) are given in Fig. 11.1–11.4. For the sake of clarity lexical items are given without their features, movement is indicated by dashed branches, and interior nodes are labeled in the fashion of X′-theory.

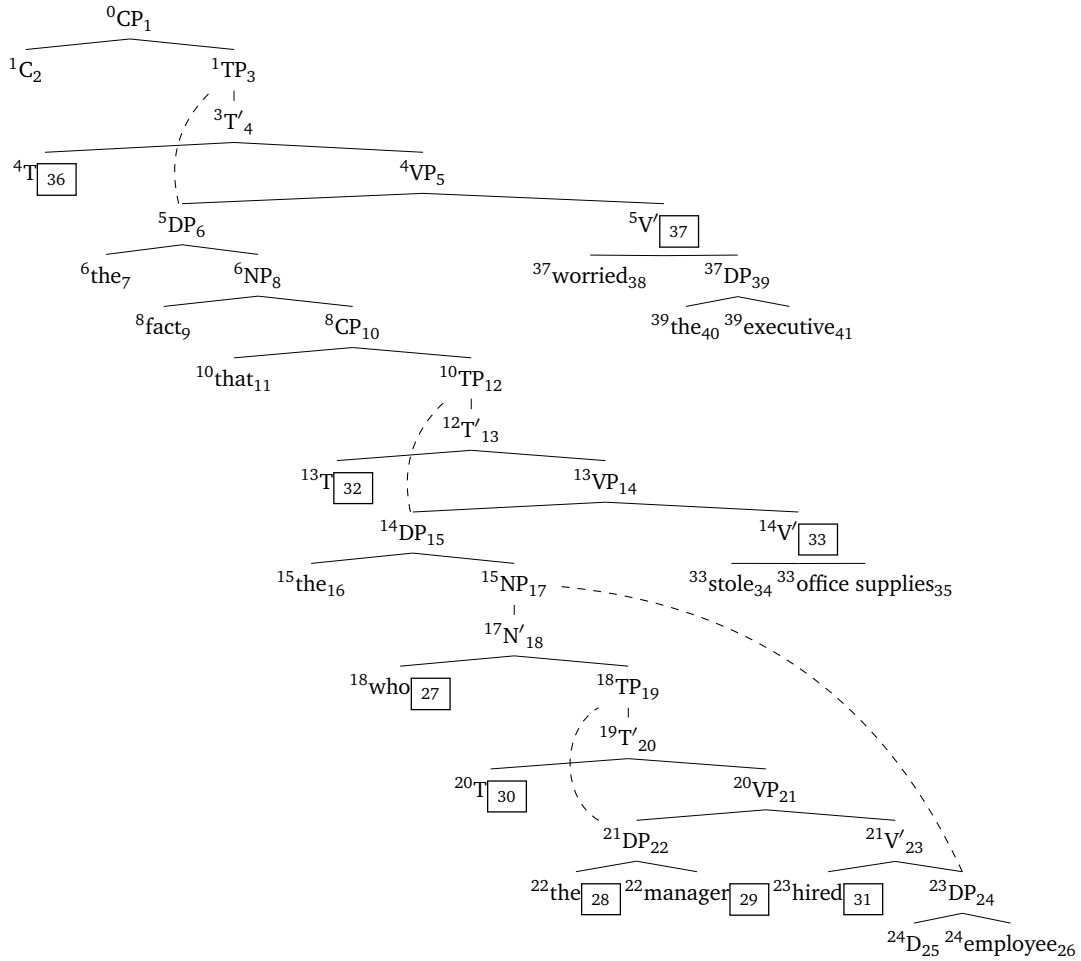


Figure 11.1: Merge-eager parse of sentential complement with embedded relative clause; **Max** = 32/32, $\textbf{Max}_{\textbf{Lex}}$ = 32/9, **Payload** = 9/6, $\textbf{Payload}_{\textbf{Lex}}$ = 7/4

As can be seen in Tab. 11.1, the Merge-eager and the Move-eager parsers behave almost exactly the same. With both of them **Max** makes barely a difference between the two structures, whereas $\textbf{Max}_{\textbf{Lex}}$ correctly prefers SC/RC if empty heads are ignored. Results are mixed for the payload metrics. With a Merge-eager parser they indeed capture the preference for SC/RC, but the Move-eager parser shows no difference for these metrics. This isn't too surprising considering that payload wasn't a useful metric for the CFG parsing models either. The surprising thing, then, is that the payload metrics actually work for Merge-eager parser.

2.4 Subject Gaps and Object Gaps

The results for subject gaps versus object gaps are slightly different. Once again $\textbf{Max}_{\textbf{Lex}}$ makes the right predictions for both parsers, and so do **Payload** and $\textbf{Payload}_{\textbf{Lex}}$

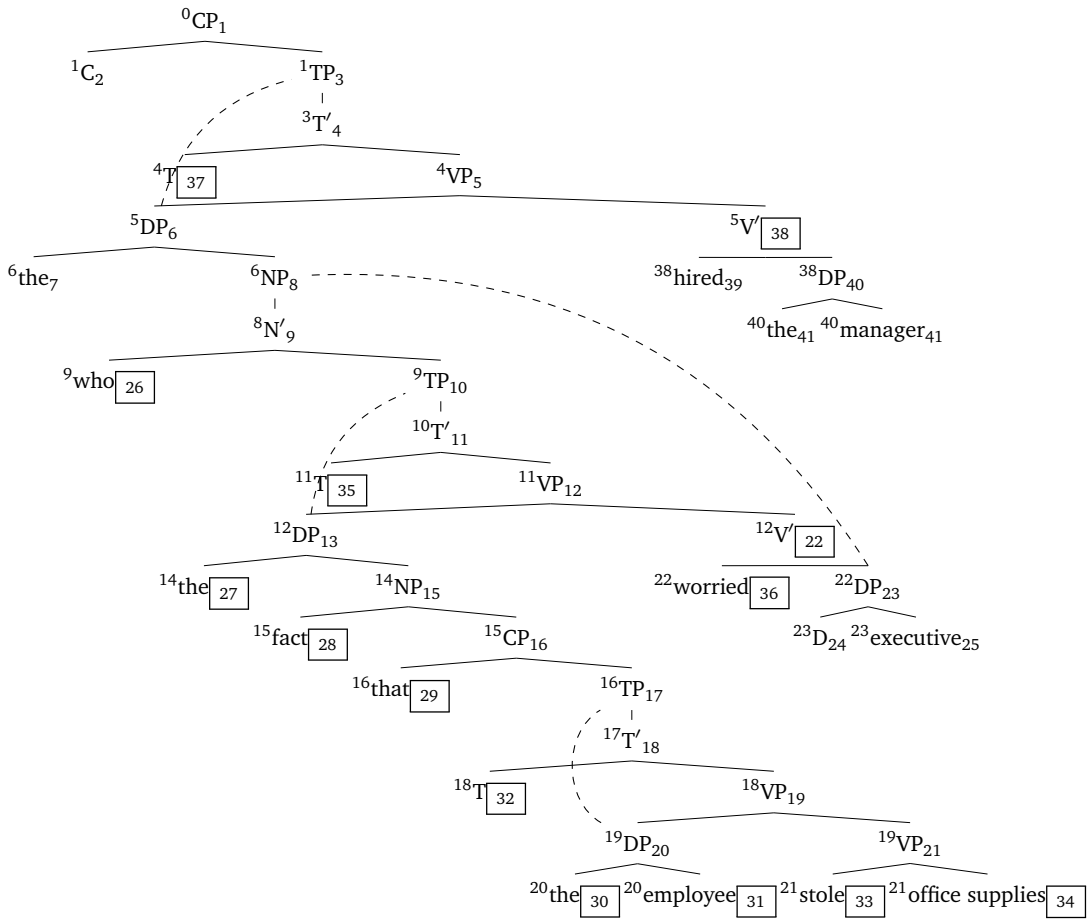


Figure 11.2: Merge-eager parse of relative clause containing a sentential complement; **Max** = 33/33, $\textbf{Max}_{\textbf{Lex}}$ = 33/17, **Payload** = 14/11, $\textbf{Payload}_{\textbf{Lex}}$ = 12/9

	SC/RC	**RC/SC**
Payload	9/6	14/11
$\textbf{Payload}_{\textbf{Lex}}$	7/4	12/9
Max	32/32	33/33
$\textbf{Max}_{\textbf{Lex}}$	32/9	33/17

(a) Merge-eager

	SC/RC	**RC/SC**
Payload	8/5	8/5
$\textbf{Payload}_{\textbf{Lex}}$	5/2	5/2
Max	31/31	32/32
$\textbf{Max}_{\textbf{Lex}}$	31/8	33/22

(b) Move-eager

Table 11.1: Overview of processing predictions for SC/RC and RC/SC

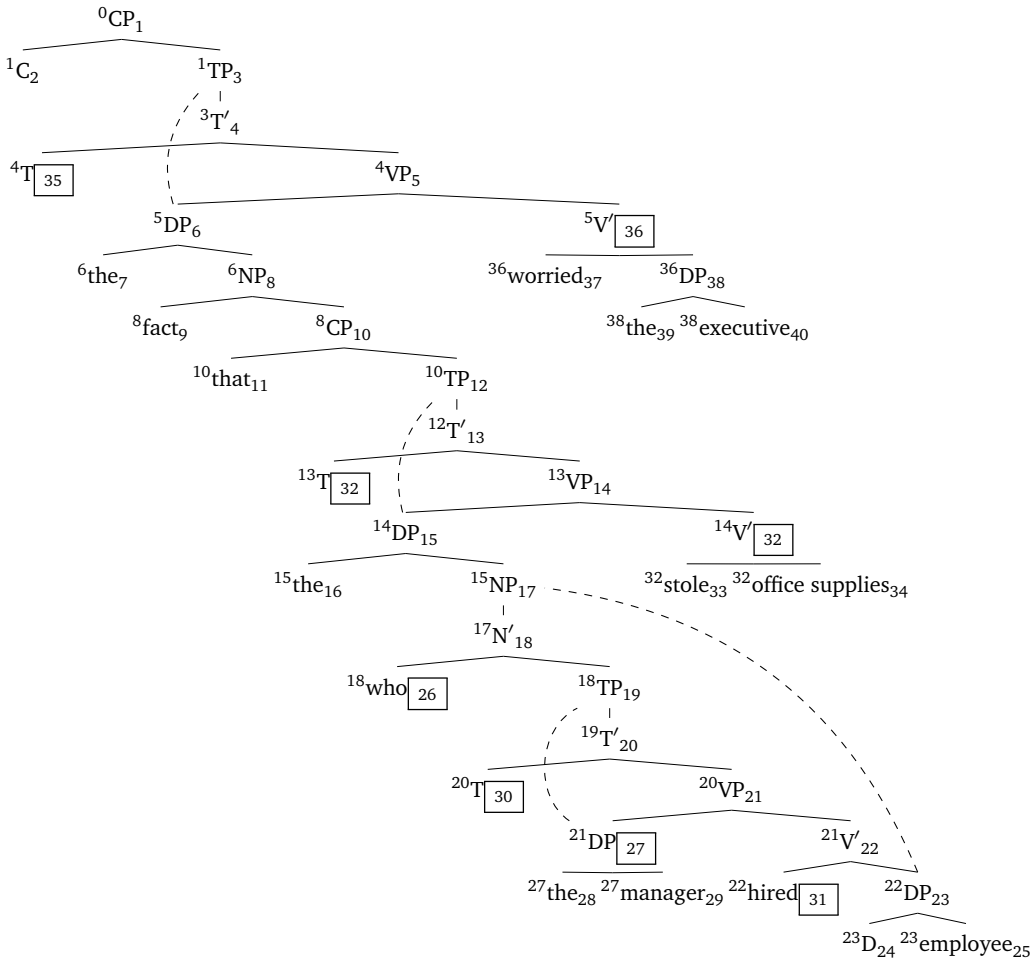


Figure 11.3: Move-eager parse of sentential complement with embedded relative clause; **Max** = 31/31, $\textbf{Max}_{\textbf{Lex}} = 31/8$, **Payload** = 8/5, $\textbf{Payload}_{\textbf{Lex}} = 5/2$

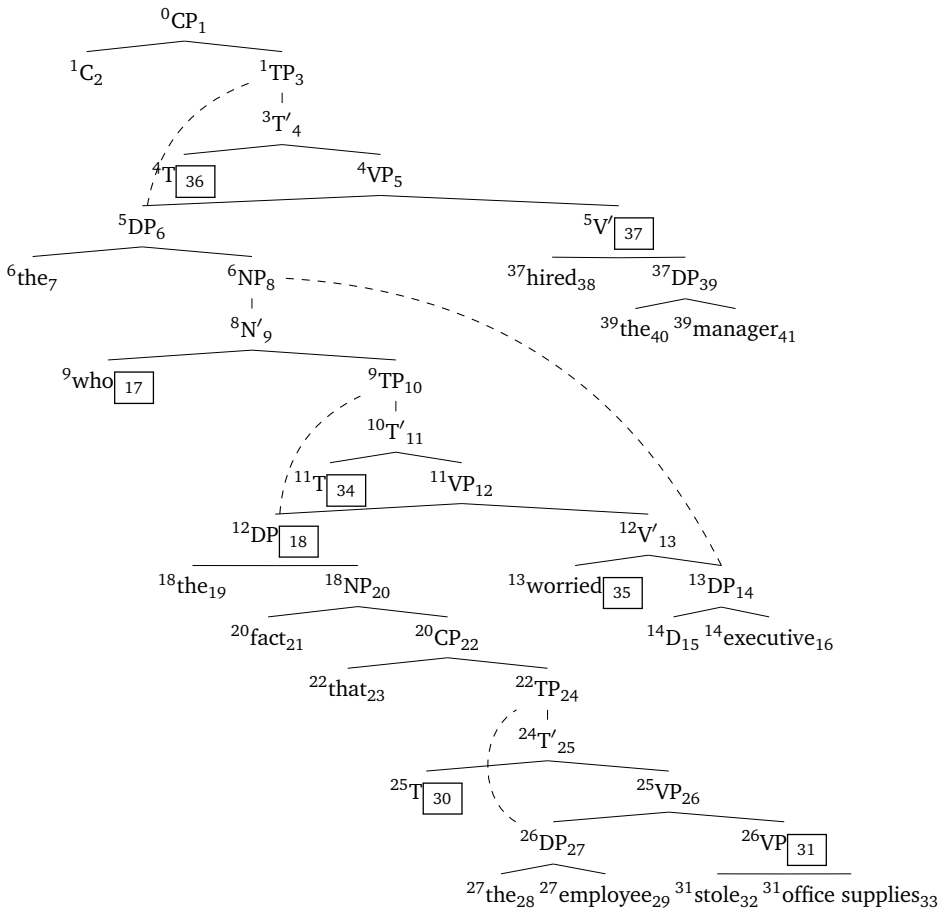


Figure 11.4: Move-eager parse of relative clause containing a sentential complement; **Max** = 32/32, $\textbf{Max}_{\textbf{Lex}}$ = 32/22, **Payload** = 8/5, $\textbf{Payload}_{\textbf{Lex}}$ = 5/2

(although the difference is much more pronounced with the Merge-eager parser in this case). **Max** seems to predict a tie, but earlier in the course we already said that if two trees have the same **Max** value, we can rank them according to the second-highest tenure value. With both parsers these values are 7/7 for the subject gap sentence and 10/9 for the object gap sentence, so a preference for subject gaps emerges even with **Max**.

	SubjRC	**ObjRC**
Payload	5/3	7/5
Payload$_{\text{Lex}}$	3/1	6/4
Max	19/19	19/19
Max$_{\text{Lex}}$	19/7	19/9

(a) Merge-eager

	SubjRC	**ObjRC**
Payload	5/3	6/4
Payload$_{\text{Lex}}$	3/1	4/2
Max	19/19	19/19
Max$_{\text{Lex}}$	19/7	19/9

(b) Move-eager

Table 11.2: Overview of processing predictions for SC/RC and RC/SC

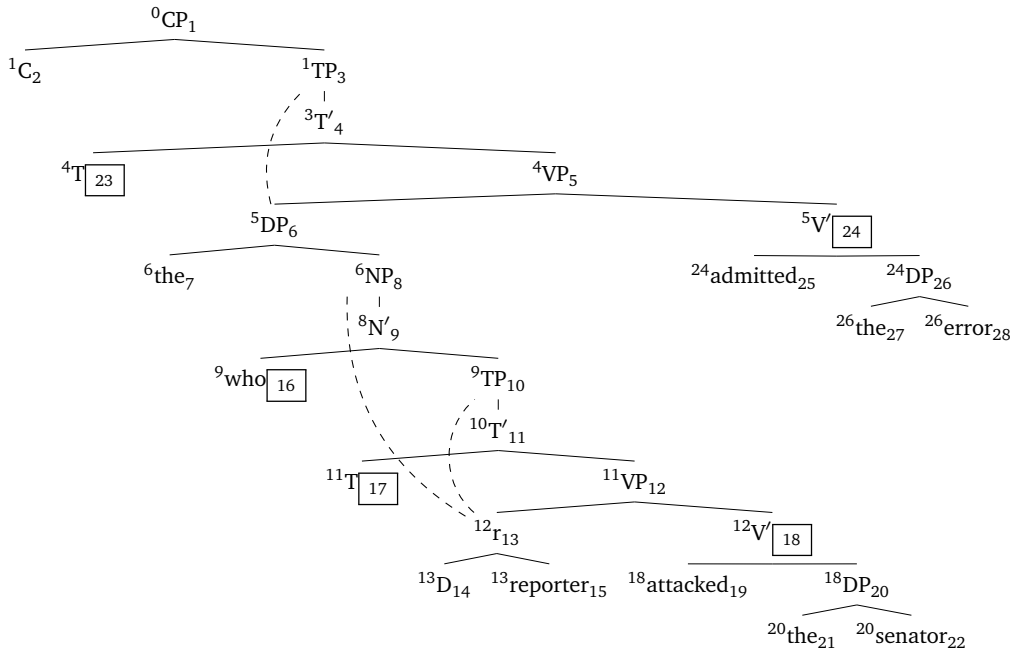


Figure 11.5: Merge-eager and Move-eager parse of relative clause with subject gap; **Max** = 19/19, **Max$_{\text{Lex}}$** = 19/7, **Payload** = 5/3, **Payload$_{\text{Lex}}$** = 3/1

References and Further Reading

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

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.

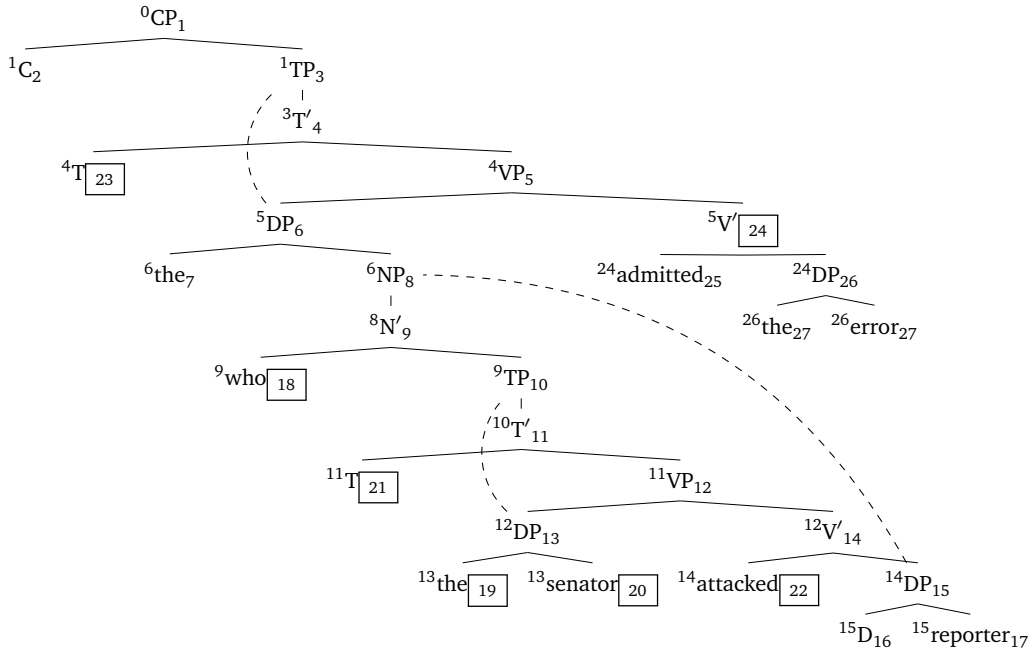


Figure 11.6: Merge-eager parse of relative clause with object gap; **Max** = 19/19, $\textbf{Max}_{\textbf{Lex}}$ = 19/9, **Payload** = 7/5, $\textbf{Payload}_{\textbf{Lex}}$ = 6/4

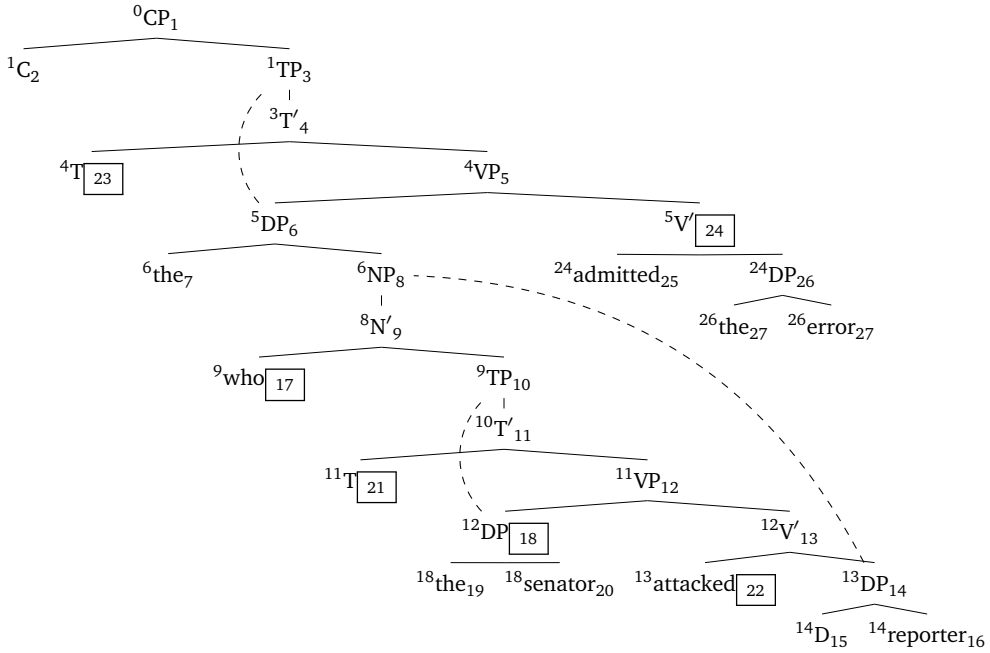


Figure 11.7: Move-eager parse of relative clause with object gap; **Max** = 19/19, $\textbf{Max}_{\textbf{Lex}}$ = 19/9, **Payload** = 6/4, $\textbf{Payload}_{\textbf{Lex}}$ = 4/2

Stabler, Edward P. 2012. Bayesian, minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5:611–633.

Lecture 12

(Quasi-)Deterministic Parsing

Lecture 13

Partial Parsing