# Parsing and Syntactic Processing

Spring 2018

Thomas Graf

# Contents

# Lecture 0

# Syllabus

**Course:** Parsing and Processing
**Course#:** Lin630
**Time:** M 11:10–12:00 & 12:10–2:00
**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, peer-graded homeworks. 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.

- **Peer Grading of Homeworks**
  The best way to check your own understanding is to see if you can evaluate the solutions of your fellow students. That is why every week two students will be in charge of grading all the homeworks (with my help, of course).

- **Project Pitch**
  At the end of the semester, you have to give me a 15 minute pitch for a follow-up project. This could be a psycholinguistic study, an implementation of a parsing algorithm, a proof of a specific theorem, whatever you find interesting. The pitch has to describe how the project intersects with your own research, explain its merit for linguistics or NLP, and draw from the material covered in this class. Ideally, you'll get me interested and we'll work on the project together.

- **Workload per Credits**

  - *1 credit*: regular attendance, class participation
  - *2 credits*: the above, plus doing all the homeworks
  - *3 credits*: the above, plus peer grading and project pitch

  Students who are retaking this course for credit can volunteer to give a guest lecture instead of a project pitch.

## 3 Outline

This outline assumes that we meet once a week for 110 minutes from 12:10 to 2:00pm. The time from 11:10 to 12:00 is reserved for discussion with the two peer-graders. If necessary, those 50 minute meetings can be moved to a different weekday so that students can get their corrected homeworks back before handing in the next batch.

A separate math & syntax primer will also be scheduled for week 1, and this time will be deducted from the very last meeting.

| Wk | Chap | Topic |
|---|---|---|
| 1 | 0 | Organization, big picture and relevance |
| 2 | 1,2 | Parsing across disciplines, modular view of parsing |
| 3 | 3 | Top-down parsing |
| 4 | 4 | Top-down predictions for sentence processing |
| 5 | 5 | Bottom-up parsing |
| 6 | 6 | (Generalized) Left-corner parsing |
| 7 | same | same |
| 8 | Spring break | |
| 9 | 7 | Chart parsing overview, CKY |
| 10 | 8 | Earley parsing |
| 11 | 10 | Parsing with probabilities |
| 12 | 12 | Moving beyond context-free grammars |
| 13 | 13,14 | CFG-parsing of Minimalist syntax |
| 14 | | guest lectures |
| 15 | | guest lectures, Q&A |

Depending on student interest, some of those chapters may be skipped to make room for other chapters, e.g. Dependency parsing, CCG parsing, or shallow parsing and partial parsing techniques.

# 4 Online Component

Rather than Blackboard, I use github to distribute lecture notes and readings (yes, you have to print them yourself). The (optional) 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.

# 5 Policies

## 5.1 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.

### 5.2 Disability Support Services

If you have a physical, psychological, medical or learning disability that may impact your course work, please contact Disability Support Services, ECC (Educational Communications Center) Building, Room 128, (631) 632-6748. They will determine with you what accommodations, if any, are necessary and appropriate. All information and documentation is confidential.

Students who require assistance during emergency evacuation are encouraged to discuss their needs with their professors and Disability Support Services. For procedures and information go to the following website: http://www.stonybrook.edu/ehs/fire/disabilities

### 5.3 Academic Integrity

Each student must pursue his or her academic goals honestly and be personally accountable for all submitted work. Representing another person's work as your own is always wrong. Faculty are required to report any suspected instances of academic dishonesty to the Academic Judiciary. Faculty in the Health Sciences Center (School of Health Technology & Management, Nursing, Social Welfare, Dental Medicine) and School of Medicine are required to follow their school-specific procedures. For more comprehensive information on academic integrity, including categories of academic dishonesty, please refer to the academic judiciary website at http://www.stonybrook.edu/uaa/academicjudiciary/

### 5.4 Critical Incident Management

Stony Brook University expects students to respect the rights, privileges, and property of other people. Faculty are required to report to the Office of Judicial Affairs any disruptive behavior that interrupts their ability to teach, compromises the safety of the learning environment, or inhibits students' ability to learn. Faculty in the HSC Schools and the School of Medicine are required to follow their school-specific procedures.

# 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.

```
4  a = 1
5
6
7  def increment_by_two(n):
8      """increment n by 2"""
9      a = 2
10     return n + a
11
12 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 $\sigma$, then $\sigma$ is a licit structure for $s$. In plain English: The parser says only correct things.

**Completeness** If $\sigma$ is a licit structure for string $s$, then the parser assigns $\sigma$ 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 $\alpha$ is rewritten as $\beta$. The definition above puts no restrictions on $\alpha$ and $\beta$ except that they are strings of symbols drawn from $N$ and $T$, and $\alpha$ 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:

$$A \overset{4}{\rightarrow} AA \overset{5}{\rightarrow} ABc \overset{3}{\rightarrow} ac$$

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

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

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

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

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

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

**Example 1.2 A Context-Free Grammar**

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

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

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

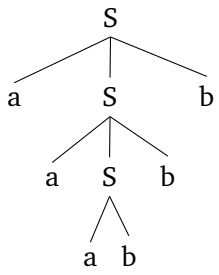
$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb$$

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

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

**Example 1.3 CFG Derivation Trees**

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




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

## 1.3 Recognizers and Parsers

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

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

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

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

# 2 Syntactic Processing

## 2.1 The Human Parser and its Quirks

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

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

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

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

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

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

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

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

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

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

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

## 2.2 Three Cognitive Oppositions

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

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

**Performance** A speaker's usage of his linguistic knowledge.

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

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

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

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

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

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

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

**Example 1.4 Set Intersection on Three Levels**

Suppose you have two sets of objects, $A$ and $B$, and you want to write a computer program that tells you which objects belong to both sets.

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

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

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

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

## 2.3 Why Processing Matters

Given what we have seen so far, the waters of syntactic processing are indeed very murky. Why, then, should we bother with syntactic processing at all rather than just focusing on computational parsing models and their usage in real-world applications? As so often in science, the most natural and immediate answer is that syntactic processing is an interesting and fun puzzle. And just like other aspects of human psychology studying it comes with the satisfaction of feeding mankind's innate vice of anthropocentric narcissism. But there are more tangible advantages.

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

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

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

explanation of this typological facts in terms of processing. In other words, some typological universals no longer need to be accounted for in the grammar and can instead be factored out into a processing-based account.

## References and Further Reading

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Lecture 2

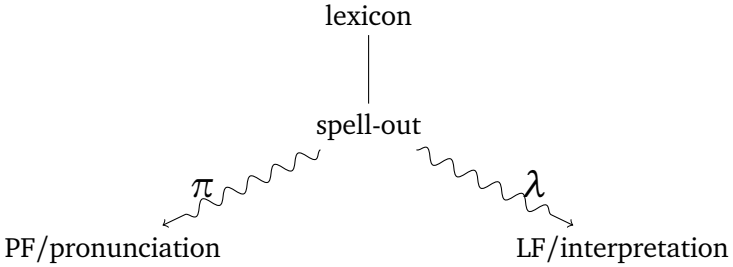
# A Modular View of Parsing

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 — that one and the same device can be described at distinct levels of abstraction — to the parser itself. We may thus differentiate 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 when studying the mathematics of parsing whereas the actual implementation in some programming language requires a lot more detail.

For this course, on the other hand, 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 cornerstones of transformational grammar is the inverted T-model, a metaphor for the place of syntax in the language faculty.




The canonical 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 $\pi$,
4. the other copy is turned into an interpretable structure by processes jointly referred to as $\lambda$.

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

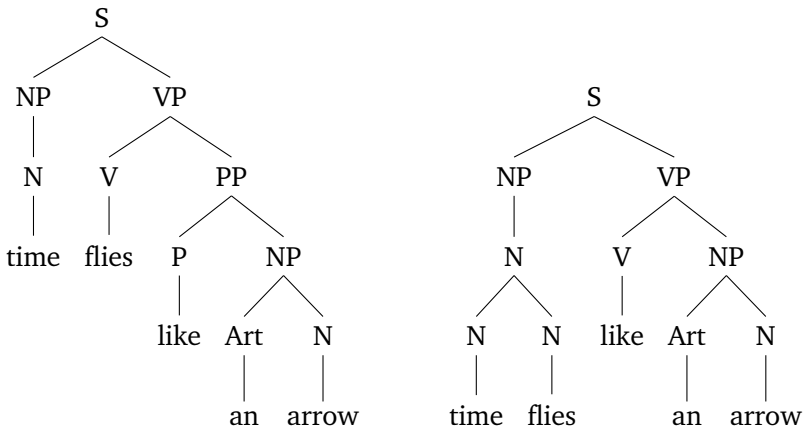
- the grammar specifies a set of well-formed representations (e.g. phrase structure trees),
- we can define various functions for mapping these representations to other structures; in the case at hand, a function $\pi$ from trees to strings and a function $\lambda$ from trees to semantic interpretations

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 structured object that has a linear realization in the form of its source code (its image under $\pi$) and a realization in terms of the instructions carried out by the computer it is run on (its image under $\lambda$). 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 $\pi$ from trees to strings, a parser for $G$ is a device that computes $\pi^{-1}$ (the inverse of $\pi$).

**Example 2.1 The $\pi^{-1}$ 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, $\pi$ can be viewed as a set of pairs $\langle a, b\rangle$, which means that $a$ is mapped to $b$ by $\pi$. So for our example, $\pi$ 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 $\pi$ is simply the set that contains the pair $\langle b, a\rangle$ iff $\langle a, b\rangle$ is a member of $\pi$. Hence $\pi^{-1}$ 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 $\pi^{-1}$ 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 $\pi^{-1}$ is parsing, what is $\lambda^{-1}$? ⊙

## 2 One Step Down: Parsing as Grammar Intersection

### 2.1 Parsing as Generation

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 $\pi^{-1}$, there is little we can do. Now for some choices of $\pi$ there are some very general algorithms for computing $\pi^{-1}$ (for example if $\pi$ 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 $G$.

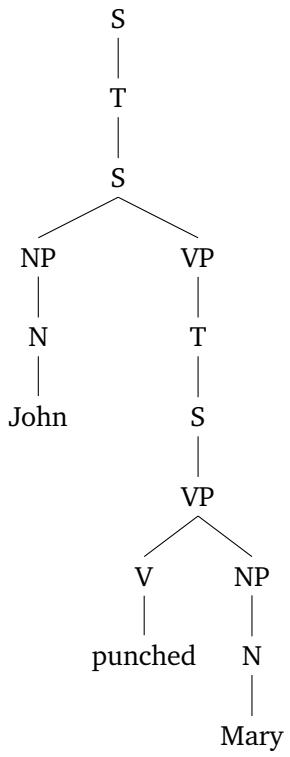
[1] The closure property actually extends to all regular string languages, which includes all finite languages.

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

**No empty heads** The empty string $\varepsilon$ 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 limits 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).

**Intersection parsing** 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? ⊙

## 2.2 Constructing the Intersection Grammar

For CFGs the intersection grammar can be constructed very easily using a technique due to Bar-Hillel et al. (1961). Essentially, we subscript each terminal with the start and end position of the substring it covers in the input sentence. That way, we end up with a grammar where each non-terminal can only generate a specific string that corresponds to a continuous part of the input sentence — each symbol is now tied to specific positions in the string.

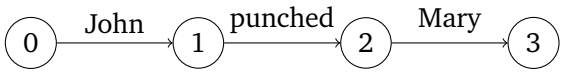
Suppose we have the following CFG.

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

This grammar generates four sentences.

(1) a. John punched John.
b. John punched Mary.
c. Mary punched John.
d. Mary punched Mary.

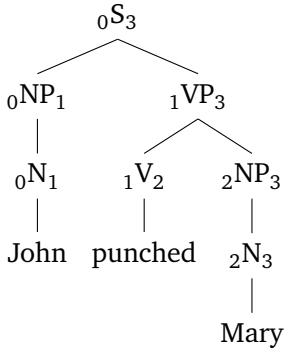
We will now construct an intersection grammar that generates all parses for the sentence *John punched Mary*. Integers are used to refer to specific positions in the input string as illustrated below.




We could construct the intersection grammar bottom-up, starting with the rules generating leaves, but we will proceed top-down instead. So consider the rule S → NP VP. Since S is the root of the generated tree, it must span the whole input string. Consequently, we replace S by ${}_0S_3$. It is also obvious that the NP starts at position 0 and the VP ends at position 3. So the new rewrite rule must be ${}_0S_3 \rightarrow {}_0NP_i\ {}_iVP_3$, where $i$ is some integer between 0 and 3. A computer would try all possible options and then discard the ones that never occur in any tree for the input sentence, but we can inspect the original grammar and see immediately that its NPs only generate strings containing exactly one word. Therefore the NP must span from 0 to 1 and we have ${}_0S_3 \rightarrow {}_0NP_1\ {}_1VP_3$. Continuing in this fashion with the other rewrite rules, we eventually obtain the intersection grammar.

| | | | | | |
|---|---|---|---|---|---|
| ${}_0S_3$ | → | ${}_0NP_1\ {}_1VP_3$ | | | |
| ${}_0NP_1$ | → | ${}_0N_1$ | ${}_0N_1$ | → | John |
| ${}_2NP_3$ | → | ${}_2N_3$ | ${}_2N_3$ | → | Mary |
| ${}_1VP_3$ | → | ${}_1V_2\ {}_2NP_3$ | ${}_1V_2$ | → | punched |

Note that we now have multiple NP rules that are almost identical except for their indices, which is expected because we have multiple NPs covering different spans of the input. This grammar also generates only one tree, which is shown below.




In principle it might be possible for an intersection grammar to generate an infinite number of trees (as a single input may have an infinite number of possible analyses). In that case it obviously isn't practical to generate all trees, but we do not need to. The intersection grammar fully represents the set of possible parses, just like the original grammar fully represents the set of well-formed trees of the language. So whatever you want to do with the parse trees, you can lift it to an operation over the intersection grammar and get your desired result this way. For this reason, intersection grammars are also called *parse forests*.

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. But try to keep intersection parsing in the back of your head — in the next weeks, you will (hopefully) come to realize that parsers are essentially algorithms for generating intersection grammars on the fly.

# 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.

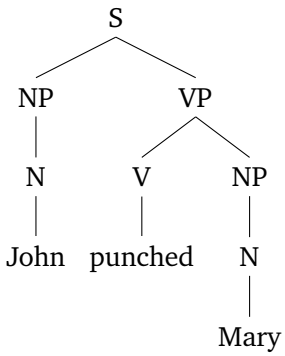
**Data structure** A system for storing and retrieving parsing items.

Example 2.3 A naive parser for CFGs

Suppose we want to parse the sentence *John punched Mary* with the same CFG as before, which is repeated here for your convenience.

| | | | | | |
|---|---|---|---|---|---|
| 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 are 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 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) and 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 every single CFG. That is an infinite class that includes many grammars that behave completely unlike natural language grammars. If a parser can be optimized for a specific subclass of CFGs, 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 (linear time, but not real-time).

## References and Further Reading

Bar-Hillel, Yehoshua, M. Perles, and E. Shamir. 1961. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* .

Chomsky, Noam. 1995. *The minimalist program*. Cambridge, MA: 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

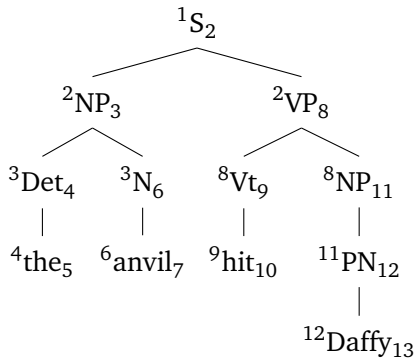
**depth-first** rewrite some symbol that was introduced during the previous rewriting step

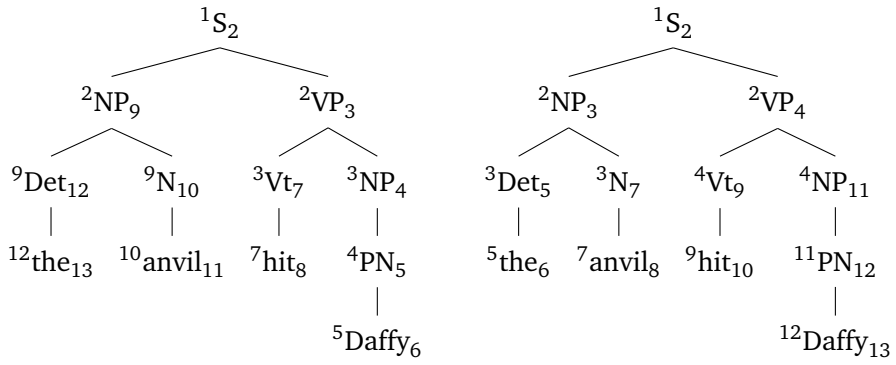
**breadth-first** before rewriting a symbol introduced during rewriting step $j$, all symbols that were introduced at rewriting step $i$ must have been rewritten, for every $i < j$

**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.





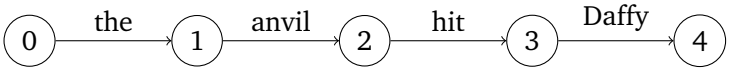

*Exercise 3.1.* Draw the table for a breadth-first, right-to-left parser, and the corresponding annotated phrase structure tree. ⊙

*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 $\beta$ 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.

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

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

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 $G$.

Strictly speaking the parsing schema does not give us the actual system of axioms and inference rules. Rather, these are obtained once the variables in the parsing schema are instantiated according to the rewrite grammar. For example, the predict rule would the instantations below (and several more) given the grammar presented earlier.

$$\frac{[i, \text{S}, j]}{[i, \text{NP VP}, j]}$$

$$\frac{[i, \text{NP VP}, j]}{[i, \text{Det N VP}, j]}$$

$$\frac{[i, \text{NP VP}, j]}{[i, \text{aN VP}, j]}$$

The instantiation of a parsing schema according to a grammar is called a *parsing system*. In contrast to the parsing schema, a parsing system may have an infinite number of rules. That's not the case here because our grammar does not have recursion and thus can only create a finite number of sentences.

*Exercise 3.3.* (Psycho-)Linguists do not distinguish between parsing schema and parsing system (nor do they factor out control structure and data structure). Is their notion of parser closer to a parsing schema or a parsing system? Or do they unwittingly switch between the two depending on context? ⊙

**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, could in principle apply every possible prediction step (depending on its control structure). So from [0,NP VP,4], for instance, the parser may not only predict [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.

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

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

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

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 3.4.* Specify the right-to-left versions of the parsers above. ⊙

*Exercise 3.5.* 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 3.6.* 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 3.7.* 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 until later, 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 3.8.* 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*. These are very powerful ideas, but their addition does not alter the parsing schema or the control structure.

The control structure also needs to include some strategy for how distinct parses should be prioritized. For instance, one may finish one parse before moving on to the next, or interleave them. Often parses are put in a *priority queue*. At each step, the first parse in the queue is opened and one inference step is applied. Depending on the outcome of this inference, the parse 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. As we will see next time, the choice of control structure plays a great role for what psycholinguistic predictions are being made.

## 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 and a serial strategy for expanding multiple parses.

## 2.2 Priority Queues and Prefix Trees

Let's assume that our control structure operates on 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/dictionaries). Priority queues provide an easy way of prioritizing specific parses. Suppose that our priority queue is $[i_1, \ldots, i_n]$, $n \geq 1$, where each $i_j$ is some parse item. Since $i_1$ is the first item in the queue it has the highest priority and should be acted on first. The parser removes $i_1$ from the queue, computes all parse items than can be inferred from $i_1$, and reinserts them into the queue at some position corresponding to their relative priority with respect to the other items in the queue.

**Example 4.1 Priority Queues for Serial and Parallel Parsing**

Consider once again our toy grammar from Cha. 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 |

Suppose that we have a recursive descent parser that explores the space of possible parses with a priority queue such that

- only the first item of the queue can be worked on,
- new items can only be added to the top of the queue.

Such a queue is also called a *stack*. Then the queue in the parse for *The anvil hit Daffy* grows as indicated below. We assume that the parser has some mechanism for preferring specific rewrite rules whenever more than one can apply.

| **queue** | **parser operation** |
|---:|---|
| [0,S] | axiom |
| [0.NP VP] | predict(1) |
| [0,PN VP] [0,Det N VP] | predict(2), predict(3) |
| [0,Bugs VP] [0,Daffy VP] [0, Det N VP] | predict(8) |
| [0,Daffy VP] [0,Det N VP] | failed parse |
| [0,Det N VP] | failed parse |
| [0,a N VP] [0,the N VP] | predict(6) |
| [0,the N VP] | failed parse |
| [1,N VP] | scan |
| [1,car VP] [1,anvil VP] [1,truck VP] | predict(7) |
| [1,anvil VP] [1,truck VP] | failed parse |
| [2,VP] [1,truck VP] | scan |
| [2,Vi] [2,Vt NP] [1,truck VP] | predict(4), predict(5) |
| [2,fell over] [2,Vt NP] [1,truck VP] | predict(9) |
| [2,Vt NP] [1,truck VP] | failed parse |
| [2,hit NP] [1,truck VP] | predict(11) |
| [3,NP] [1,truck VP] | scan |
| [3,PN] [3,Det N] [1,truck VP] | predict(2), predict(3) |
| [3,Bugs] [3,Daffy] [3,Det N] [1,truck VP] | predict(8) |
| [3,Daffy] [3,Det N] [1,truck VP] | failed parse |
| [4,] [3,Det N] [1,truck VP] | scan, parse found! |

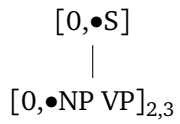
Now compare this to a recursive descent parser that operates almost exactly the same except that new items are added to the end of the queue. While the previous parser fully explored one parse before moving on to another one, this one alternates between parses after every step. In psycholinguistic parlance, then, the parser above is *serial* whereas the one below is *parallel*.

| queue | parser operation |
|---|---|
| [0,S] | axiom |
| [0.NP VP] | predict(1) |
| [0,PN VP] [0,Det N VP] | predict(2), predict(3) |
| [0, Det N VP] [0,Bugs VP] [0,Daffy VP] | predict(8) |
| [0,Bugs VP] [0,Daffy VP] [0,a N VP] [0, the N VP] | predict(6) |
| [0,Daffy VP] [0,a N VP] [0, the N VP] | failed parse |
| [0,a N VP] [0, the N VP] | failed parse |
| [0,the N VP] | failed parse |
| [1,N VP] | scan |
| [1,car VP] [1,anvil VP] [1,truck VP] | predict(7) |
| [1,anvil VP] [1,truck VP] | failed parse |
| [1,truck VP] [2, VP] | scan |
| [2,VP] | failed parse |
| [2,Vi] [2,Vt NP] | predict(4), predict(5) |
| [2,Vt NP] [2,fell over] | predict(9) |
| [2,fell over] [2,hit NP] | predict(10) |
| [2,hit NP] | failed parse |
| [3,NP] | scan |
| [3,PN] [3,Det N] | predict(2), predict(3) |
| [3,Det N] [3,Bugs] [3,Daffy] | predict(8) |
| [3,Bugs] [3,Daffy] [3,a N] [3,the N] | predict(6) |
| [3,Daffy] [3,a N] [3, the N] | failed parse |
| [3,a N] [3, the N] [4,] | scan, parse found! |

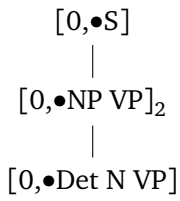
As so often, a more abstract perspective is helpful here to truly appreciate what is going on. Instead of a table listing the modifications of the priority queue, we can use a *prefix tree* as a unified way of representing multiple parses at once. The root of the prefix tree is always our start axiom [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, and a priority queue defines a specific traversal of the prefix tree.

**Example 4.2 Parse Tables as Trees**

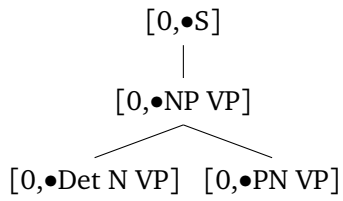
Suppose we are using the grammar from the beginning of Lecture 3 to parse *the anvil hit Daffy* with a recursive descent parser. Our parse table starts with [0,•S] as usual, from which we can only predict [0,•NP VP]. This can be represented as the tree below.



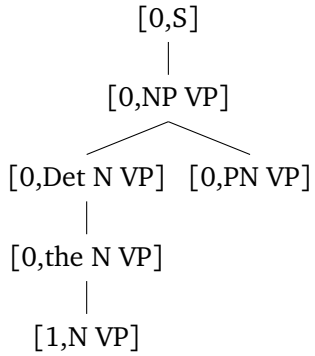

Note that [0,•NP VP] has subscripts 2 and 3 to indicate that we can use rules 2 and 3 of our CFG to predict new parse items. Suppose the parser uses predict(3) to create the item [0,•Det N VP]. Then this item is added as a daughter of [0,•NP VP] to the previous tree, and the subscript 3 is also removed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



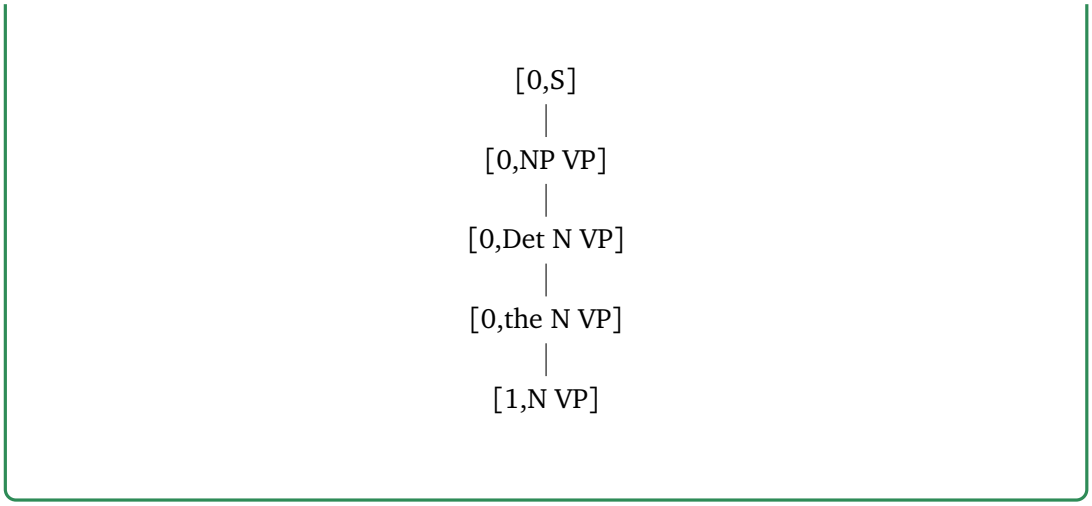

The next step of the parser now depends on how the control structure prioritizes distinct parses. We can either expand the table that ends in [0,•Det N VP], or go back to the one ending in [0,•NP VP] and apply predict(2). Suppose we do the latter, so that [0,•PN VP] is added as the second daughter of [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 the parse space 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, and as we saw in example 4.1 this can be implemented as a stack, i.e. a priority queue where new parse items are always added at the start of the queue.

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 or a priority queue where new parse items are added at the end 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

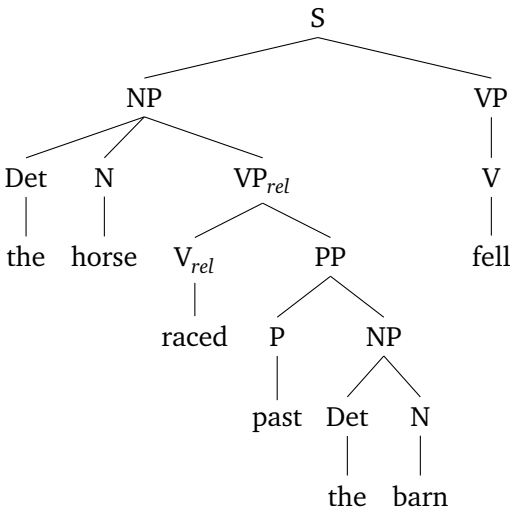
effects. In the case of *the horse raced past the barn fell*, for example, the parse builds a single parse table, and due to how certain rewrite rules are preferred over others, this parse table encodes the structure for *the horse raced past the barn*. When encountering *fell*, the parser has to backtrack. 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 4.3 Backtracking in *the horse raced past the barn fell***

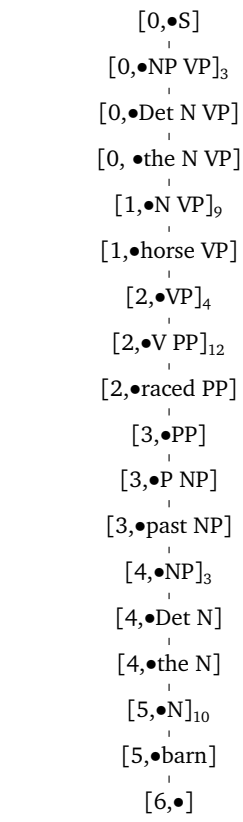
Assume we have the following (massively simplified) grammar:

| | | | | | |
|---|---|---|---|---|---|
| 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 |

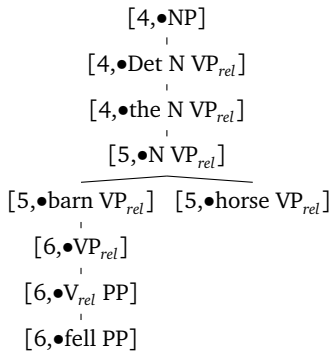
Here's the resulting tree for our garden path sentence.



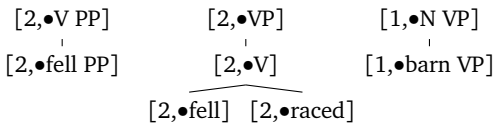

If the parser prefers NP → Det N over NP →Det N CP and operates in a recursive descent fashion, the construction of the first parse table results in the prefix tree below.



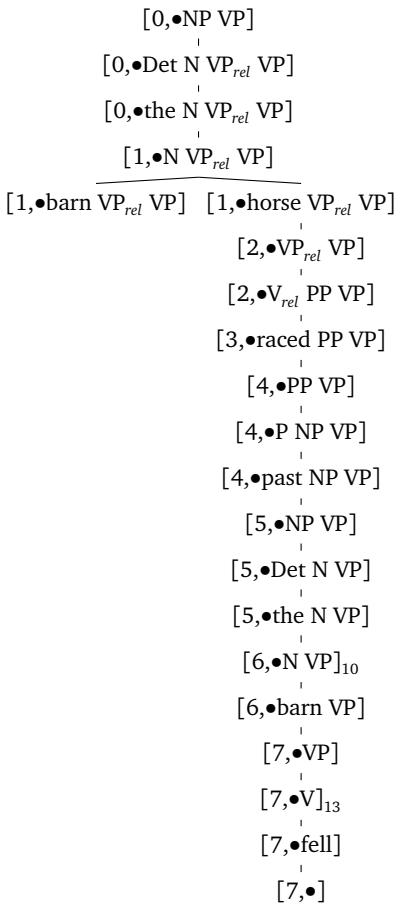

Since this parse does not succeed, the parser needs to backtrack. The closest choice point is $[5,\bullet N]_{10}$, which obviously does not fix the problem of integrating *fell* into the structure, as can be verified after a single scan step. The next choice point is $[4,\bullet NP]_3$. Here the parser 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.



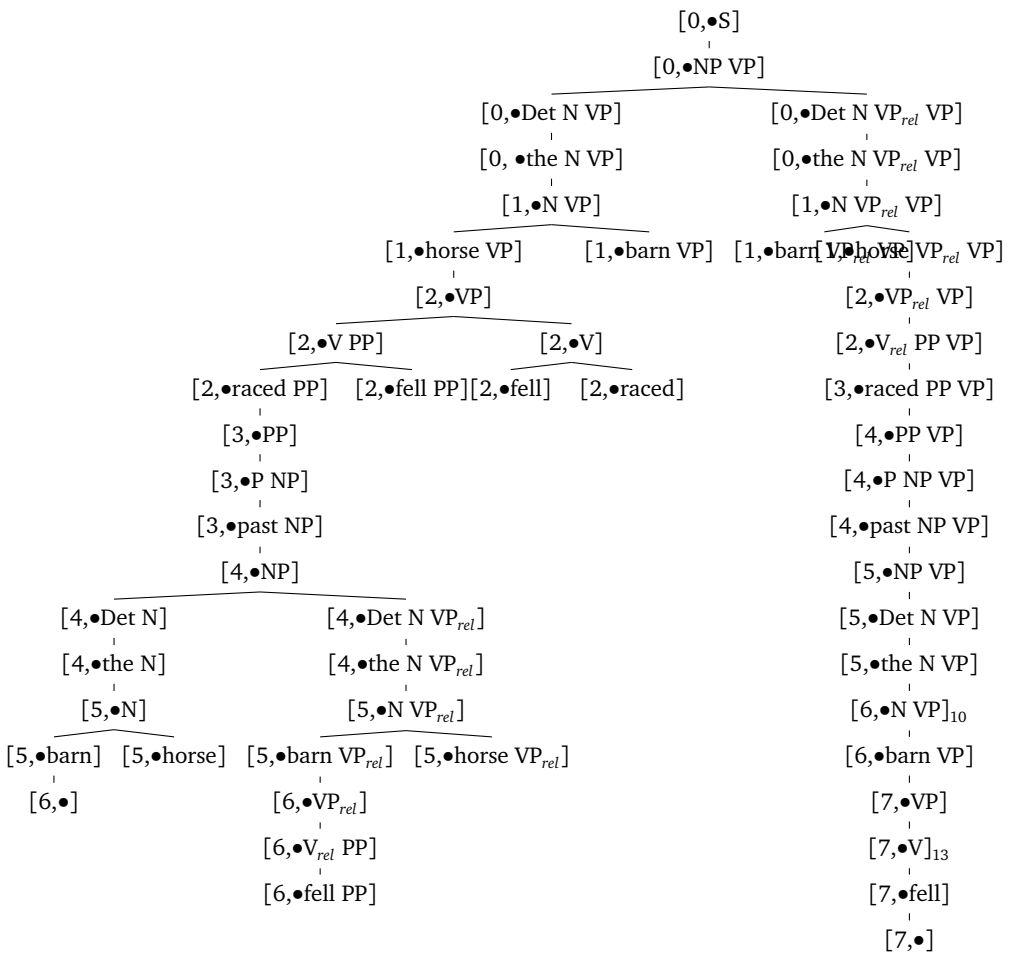

Since this venue didn't yield a successful parse either, the parse backtracks to $[2,\bullet V\ PP]_{12}$, and after this fails, to $[2,\bullet VP]_4$. Once again it is not successful, and the same holds once it expands [1,•N VP].




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.




In sum, the Frazier model can be defined more rigorously as a recursive descent parser with a stack as its priority queue. As the Frazier model is designed to account for garden paths, it is no surprise that its formal counterparts makes the right processing predictions in this case.

## 3 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 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. 2013, Graf and Marcinek 2014, and Graf et al. 2015).

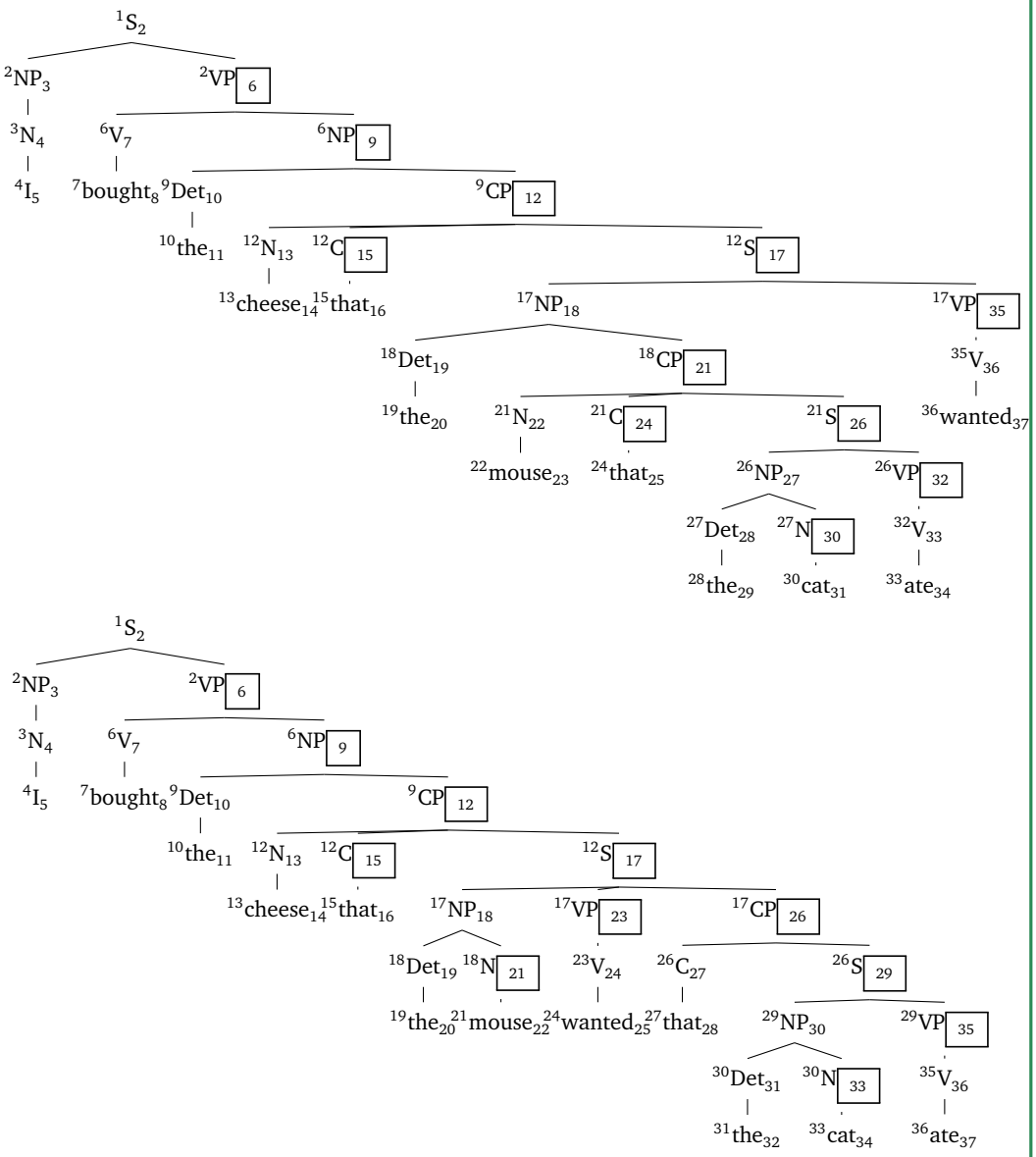
**MaxTen** greatest tenure among all the nodes; $\max(\{\text{tenure}(n)\})$

**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 4.4 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.

# 4 Problems

## 4.1 Memory Usage in Left Embedding

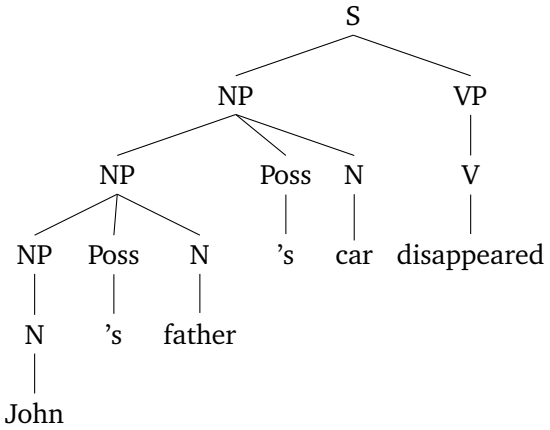
A careful examination of the top-down parsers behavior over center embedding constructions reveals that these configurations aren't the only ones that should cause an increased memory burden. Coupled with our metrics of processing difficulty, a recursive descent parser predicts that left embedding should 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.

## 4.2 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 limit 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 is 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. For more details on top-down beam parsing, see Roark (2001a,b, 2004).

### 4.3 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. More on that in chapter 5.

## 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*, 13–24.

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

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

Kayne, Richard S. 1994. *The antisymmetry of syntax*. Cambridge, MA: 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. 2013. Memory resource allocation in top-down Minimalist parsing. In *Formal Grammar: 17th and 18th International Conferences, FG 2012, Opole, Poland, August 2012, Revised Selected Papers, FG 2013, Düsseldorf, Germany, August 2013*, ed. Glyn Morrill and Mark-Jan Nederhof, 32–51. Berlin, Heidelberg: Springer. URL https://doi.org/10.1007/978-3-642-39998-5_3.

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.

Roark, Brian. 2001a. Probabilistic top-down parsing and language modeling. *Computational Linguistics* 27:249–276.

Roark, Brian. 2001b. *Robust probabilistic predictive syntactic processing: Motivations, models, and applications*. Doctoral Dissertation, Brown University.

Roark, Brian. 2004. Robust garden path parsing. *Natural Language Engineering* 10:1–24.

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 acts as the foundation of a variety of parsing algorithms. But just like top-down parsing, it has certain shortcomings regarding psycholinguistic adequacy.

## 1 Intuition

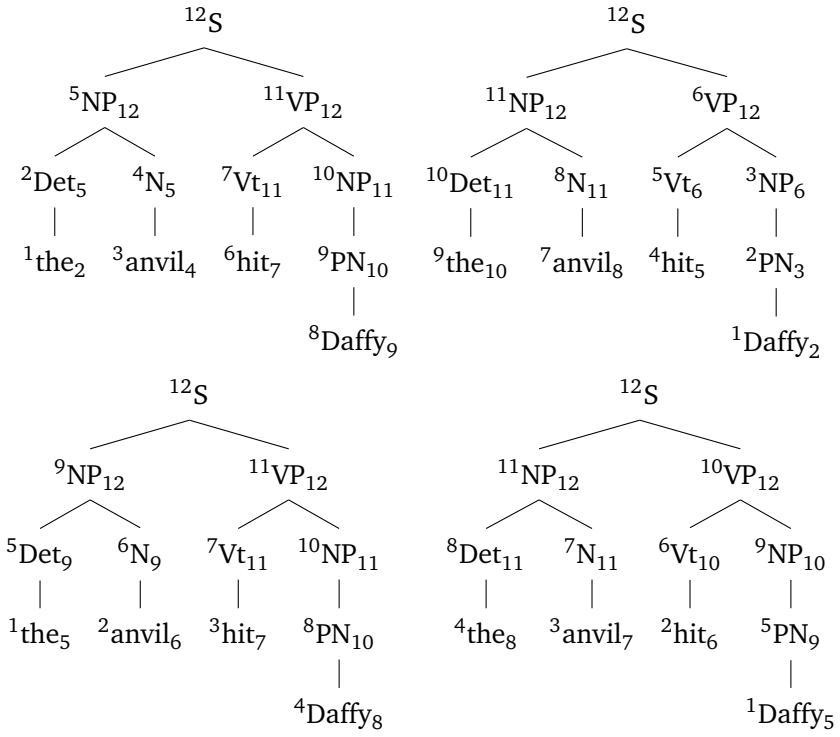
For illustration we use the same grammar as for the top-down parser in Lecture 3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1) | S | → | NP VP | 6) | Det | → | a \| the |
| 2) | NP | → | PN | 7) | N | → | car \| truck \| anvil |
| 3) | NP | → | Det | 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 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 a bottom-up parse table 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 (partially) 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? ⊙

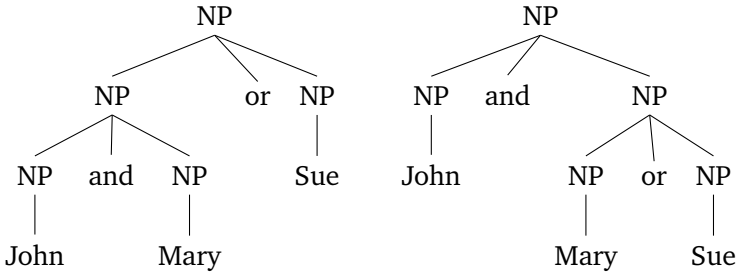
Notice that both parser have a certain overlap between the rules. In the case of the depth-first parser, the domains of the shift and reduce rules are not disjoint. That is to say, for some parse items both shift and reduce are valid continuations of the parse, which is called a *shift reduce conflict*. Shift reduce conflicts are an unavoidable 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 $\beta$, $\gamma$ and $\delta$ to reduce $\beta$. This rule is indispensable in 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

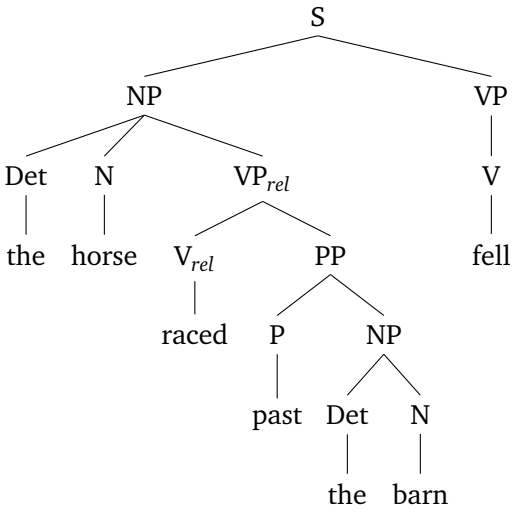
## 3.1 Garden Paths

If the parser prioritizes reduction over shifting (which is up to the control structure), a serial shift reduce parser with backtracking makes similar predictions to a recursive descent parser. Garden path effects cannot arise if shifting is prioritized to such an extent that *fell* is read in by the parser before it attempts to reduce *raced* to a V.

**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.

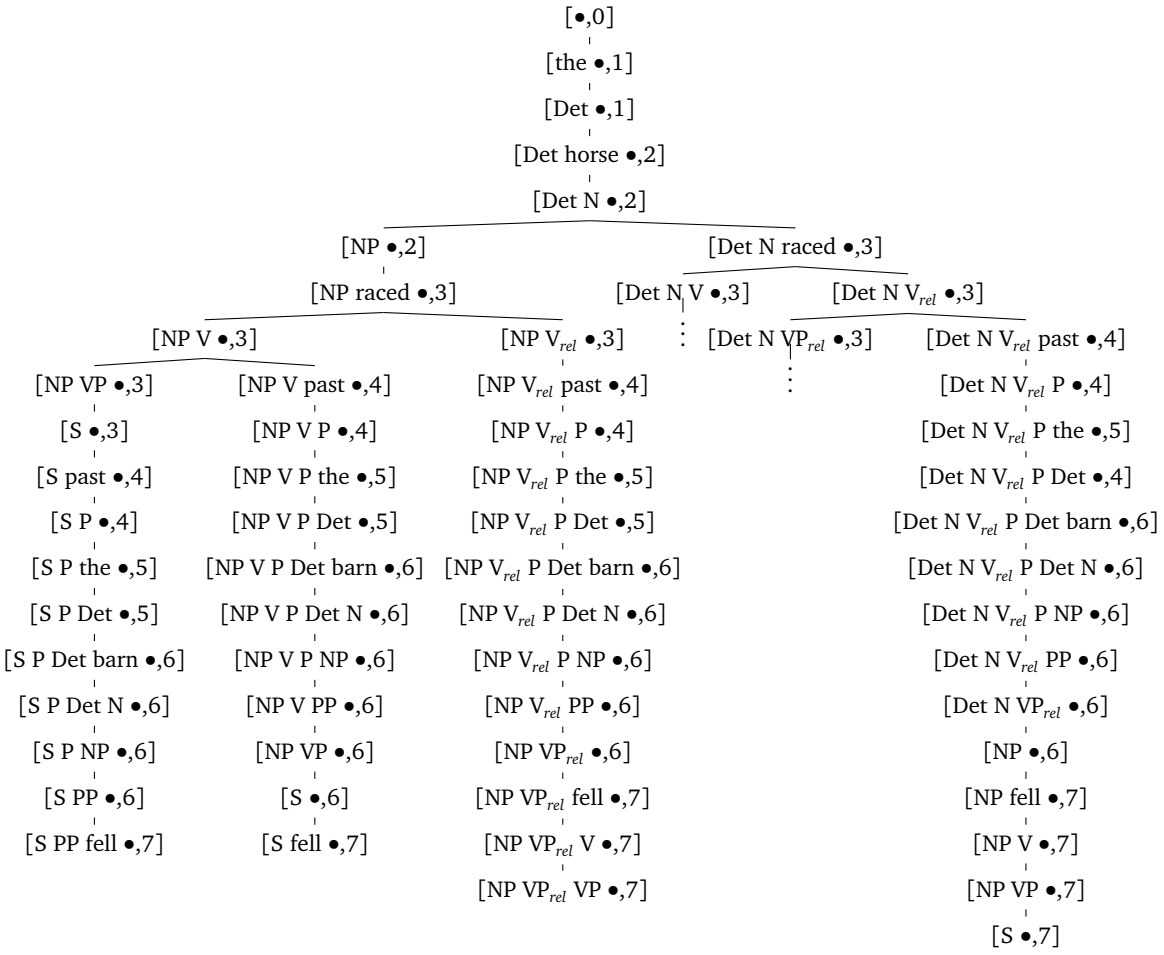


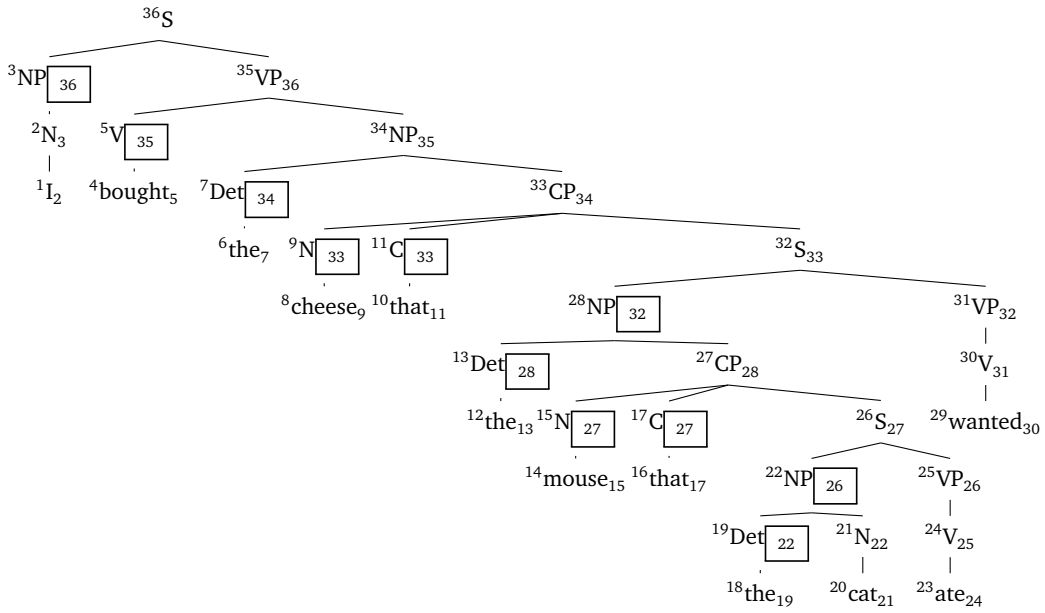

Figure 5.1: Parse history for serial shift reduce parse of *the horse raced past the barn fell*; the parser moves through the history in a recursive descent fashion

## 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. 3.




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 and those are problematic for top-down parsers, the bottom-up parser apparently struggles with center embedding because it also struggles with right embedding. The reason is simple:

1. Reduction 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 |

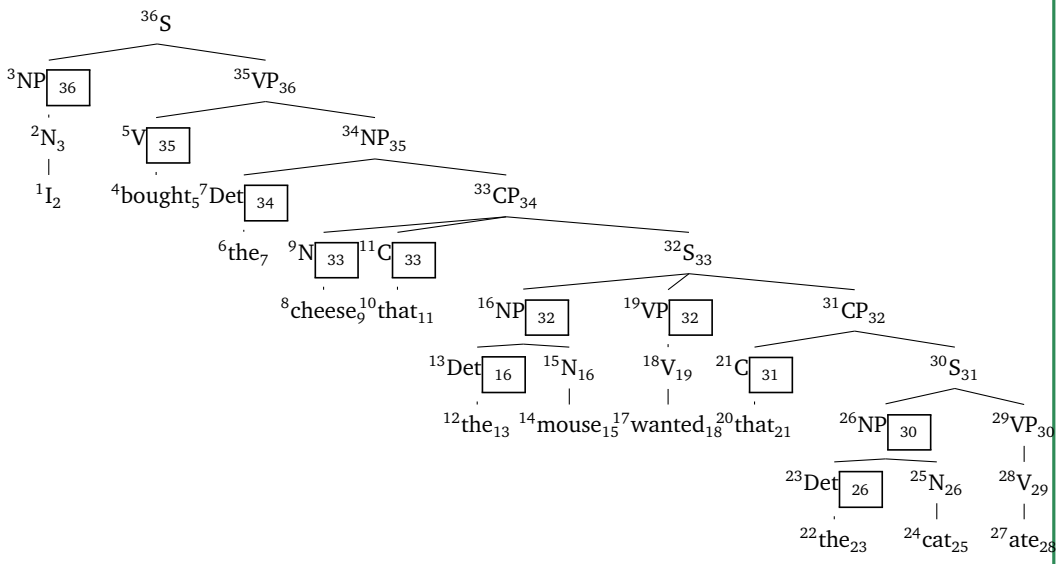
(a) Recursive Descent

| 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

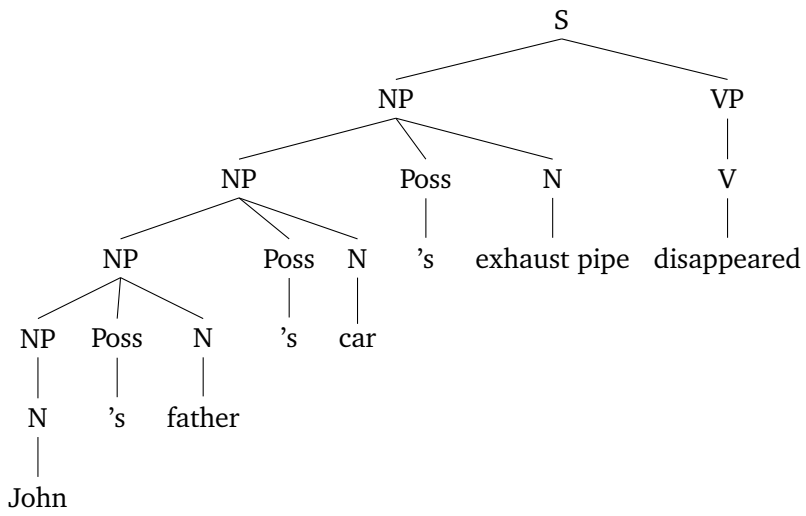
**Example 5.8 Run of shift reduce parser over right embedding sentence**




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.

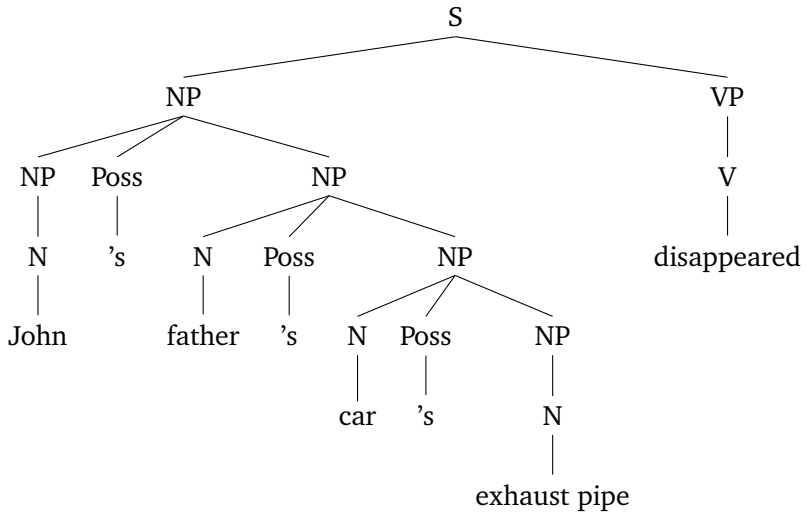
*Exercise 5.4.* Assume that the sentence *John's father's car's exhaust pipe disappeared* has the structure below.




⊙

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)? Do shift and reduce need to be prioritized in a specific fashion? ⊙

### 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 $\alpha$ has been fully analyzed — i.e. the parser has assigned a single connected subtree to $\alpha$ — the conjectured structure of $\alpha$ has no effect on which structures the parser may entertain for $\beta$. In particular, the parser will entertain analyses of $\beta$ that are incompatible with its analysis of $\alpha$.

This noncommittal attitude does not seem to be shared by the human parser. For one thing, $\alpha$ 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

# Left-Corner Parsing

Top-down parsers and bottom-up parsers each turned out to have their advantages as well as their 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 all the time.

Merely local syntactic coherence effects suggest that the human parser sometimes entertains incompatible parses, just like bottom-up parsers. But these effects are very rare and very minor compared to, say, the obvious difficulties with garden path sentences. The human parser is also predictive since ungrammatical sentences are recognized as such as soon as the structure becomes unsalvageable. At the same time, though, the prediction process differs (at least naively) from pure top-down parsing as it seems to be actively guided by the input. What we should look at, then, is a formal parsing model that integrates top-down prediction and bottom-up reduction. Left-corner parsing does exactly that.

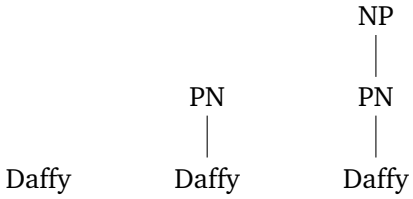
## 1 Intuition

The ingenious idea of left-corner (LC) parsing is to restrict the top-down 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. 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 $\gamma$ once it has reached $Y$ in a bottom-up fashion.

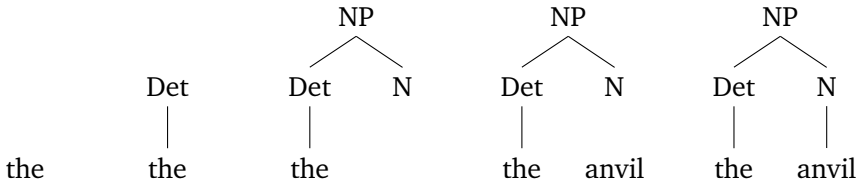
Consider our familiar toy grammar.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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, an LC parser waits until it has identified an NP before it tries to build an S. The NP, in turn, must be found in a bottom-up fashion. This may involve a sequence of bottom-up reductions: read *Daffy*, reduce *Daffy* to PN, reduce PN to NP.



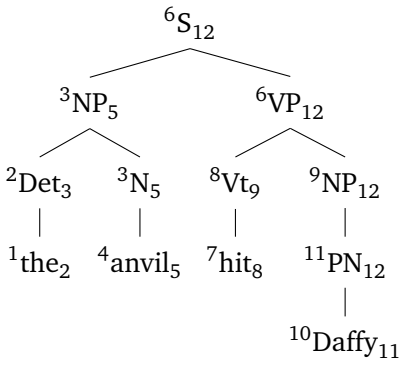

Alternatively, it may involve a mixture of bottom-up reduction and left-corner condition prediction: read *the*, reduce to Det, use the rewrite rule NP → Det N in your top-down prediction, read *anvil*, reduce to N, reduce Det N to NP.




Once the NP has been identified, the parser may use S → NP VP in a prediction step. The full parse for *the anvil hit Daffy* is depicted in tabular format below.

| **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 LC parser is depth-first left-to-right. A breadth-first LC parser behaves like a bottom-up parser if LC predictions are delayed, or like a depth-first LC parser if they apply as usual. And a right-to-left depth-first LC parser has no use for LC predictions since the predicted material has already been inferred in a bottom-up fashion anyways.




*Exercise 6.1.* The tree above shows how LC parsing can be represented via our usual annotation scheme of indices and outdices. 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. ⊙

*Exercise 6.2.* Building on your insights from the previous exercise, explain why a breadth-first LC parser is either a bottom-up parser or behaves exactly like a 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, we immediately define the parsing schema with some of the control structure incorporated via the familiar dot •. 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,
- $\alpha$ is the list of identified unused nodes (derived via bottom-up reduction), and
- $\beta$ is a list of labeled lists of phrases to be built (the top-down predictions).

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.

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

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

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

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

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

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.

The predict rule necessarily extends the prediction mechanism of a standard top-down parser since left-corner prediction is conditioned both bottom-up, when it infers the symbol to the left of the rewrite arrow, and top-down, when it infers the sister nodes to the right. An existing left-corner $N$ is removed, and instead we add to $\beta$ 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 have 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 is formally encoded by pushing it to the left side of $\bullet$.

Example 6.1 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,Vt •[$_{S}$ 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 |

*Exercise 6.3.* Consider the minimally different *The anvil suddenly hit Daffy* and *The anvil hit Daffy suddenly*.

1. Add appropriate rewrite rules to our toy grammar so that they can generate these sentences, with *suddenly* analyzed as a VP-adjunct.
2. Write down the parse tables for both sentences.
3. At what point do they differ from the one for *The anvil hit Daffy*?
4. Upon careful inspection, it is clear that *The anvil suddenly hit Daffy* is less likely to be misanalyzed by the LC parser than *The anvil hit Daffy suddenly*. Explain why! ⊙

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 •. So • indicates the demarkation line between confirmed and conjectured material in all three parsers. Viewed from this perspective, the inference rules of the left-corner parser highlight its connections to top-down and bottom-up parsing. This becomes even more apparent when the inference rules of the parser are aligned next to each other as in Tab. 6.1) (the empty sides of recursive descent and shift reduce parsers are filled by variables to highlight the parallel to LC parsing).

| | 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 6.1: Comparison of recursive descent, shift reduce, and left-corner parser

The connections between the parsers can be strengthened even more. The scan rule of the recursive descent parser does not quite match the one in the LC parser, it looks as if the two are performing very different tasks. The former checks a prediction against the input, the latter cancels out a prediction against some previously found material. But this is in fact just a generalization of recursive descent scanning from terminals to non-terminals. To make this more apparent, we can decompose the recursive descent scanning rule into a shift rule and a second rule that closely mirrors the LC scan rule:

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

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

So the scan rule we used for the recursive descent parser is just a convenient shorthand for shift followed by scan as defined above. This is also called a *step contraction* (Sikkel 1997): a sequence of inference rules is compressed into the application of a single inference rule.

*Exercise 6.4.* The LC parser itself also contains a step contraction. Show that the reduce rule is just a step contraction of two other inference rules. ⊙

## 2.2 Adding Top-Down Filtering

As the predict rule of the LC parser is conditioned by the presence of recognized material, bottom-up information serves to prune down the number of possible predictions. Note, however, that reduction steps can still apply very freely. This is somewhat wasteful. Top-down information should also be used to restrict the set of reductions, and as we will see next this is very simple in an LC parser.

Consider once again the LC parse for the sentence *the anvil hit Daffy*, and suppose that our grammar allows for *hit* to be reduced to either N or Vt (only the latter is the case in our usual toy grammar). The parser does not encounter *hit* in the input until a VP has already been predicted: first the parser recognizes then NP, then it uses NP as the left corner for predicting S and VP, and then it shifts one word to the right in the input and finally reads in *hit*. A quick glance at our grammar will reveal that it is impossible for *hit* to be a noun in this parse. Obviously *hit* must be the leftmost word of the string spanned by the VP, and there is no sequence of rewrite rules in our grammar that could generate a VP with a noun at its left edge. If we could incorporate that line of reasoning into the inference rules of the parser, we might be able to save us a lot of work exploring doomed parses.

This idea can be made precise by generalizing the notion of left corner. So far, a left corner was defined as the leftmost element of the right-hand side of a rule. This will now be given the more specific term *direct left corner*, and $X$ is a left corner of $Z$ iff $X = Z$ or there are $Y_1, \ldots, Y_n$ such that $X$ is the left corner of $Y_1$, each $Y_i$ is a left corner of $Y_{i+1}$, and $Y_n$ is the left corner of $Y_n$. More succinctly:

---

*Definition 6.1 (Left Corner).* The *direct left corner relation* holds between $X$ and $Y$ in grammar $G$ iff $G$ contains a rewrite rule $Y \rightarrow X\beta$, $\beta \in (N \cup T)^*$. The *left corner relation* lc is the reflexive transitive closure of the direct left corner relation. We write $\text{lc}(Y)$ for the set $\{X \mid \langle X, Y\rangle \in \text{lc}\}$ of left corners of $Y$.

---

If this is still confusing to you, just remember that $X$ is a left corner of $Y$ iff our grammar can generate a subtree with root $Y$ where $X$ is the root or occurs along the leftmost branch.

Keep in mind that we can compute in advance for every pair of non-terminals $X$ and $Y$ whether $X$ is a left corner of $Y$. So we can use the left corner relation as a side condition in our inference rules without worrying about whether they can still be refined into a parsing system.

*Exercise 6.5.* Explain step by step why this is the case. What properties of CFGs, parsing systems, and the left corner relation are relevant here? ⊙

All we have to do now is to add a side condition to the reduce rule that implements top-down filtering.

$$\textbf{Reduce} \qquad \frac{[i, \alpha\gamma \bullet \beta]}{[i, \alpha N \bullet \beta]} \; N \rightarrow \gamma \in R, \text{ and if } \beta = [_X Y]\delta \text{ then } N \in \text{lc}(Y)$$

*Exercise 6.6.* Consider a modified version of our toy grammar that also contains the rewrite rule N → hit. For each non-terminal, compute its set of left corners. Then write down a detailed parse table of *the anvil hit Daffy* and highlight the step at which the parser is forced to reduce *hit* to a verb. ⊙

*Exercise 6.7.* In an earlier exercise you had to show that the reduce rule is a step contraction of two other rules. Consequently, restricting the applicability of reduce is not enough to add top-down filtering since the parser also has an alternative means of reducing that is still completely unrestricted. Try to patch up this loop hole. *Hint*: you can either put a similar restriction on those other two rules or ensure that the two can no longer be used as an alternative to reduce. ⊙

## 2.3 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. We can also let this threshold vary between rewrite rules to hold off on cases with more ambiguity while committing quickly whenever a specific rewrite step is much more likely than the alternatives.

An LC parser where the threshold for predictions is allowed to vary for each rule is called a *generalized left-corner parser* (GLC). It uses the same rules as a standard left-corner parser, except that the prediction rule is slightly modified. First, assume that each rewrite rule is associated with a specific index that indicates the threshold at which the left corner prediction is triggered. We can indicate this position pictorially by putting a $\star$ at the appropriate position in the rewrite rule. For example, NP → Det A N might be written NP →Det A $\star$ N. Then we can generalize the predict rule from a simple LC parser to a GLC parser.

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

The GLC parser points out yet another close connection to top-down and bottom-up parsing, which now turn out to simply be special cases of the latter. A bottom-up parser is a GLC parser where the star is always at the end of a rewrite rule, 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 $[i, \alpha M \bullet]$. So bottom-up reduction is a step contraction of prediction followed by completion.

A top-down parser is similar to a GLC parser where the star is always at the beginning of the right-hand side of the rewrite rule, so the prediction rule is never restricted by a left corner. This analogy is not completely right, however, because such a GLC 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. But LC parsers are nonetheless very closely related to top-down parsing, as we will see in the next section.

**An important terminological remark** The term generalized left-corner parser is used very differently in psycholinguistics and computer science. The definition above is the psycholinguistic one. In computer science, there are at least two alternative definitions. One is simply the standard LC parser covered in this section — GLC is then used to distinguish it from a so-called deterministic left-corner parser (which is

of little interest to us at this point since natural language sentences are ambiguous and thus inherently non-deterministic). The other usage of GLC refers to an LC parser with a particular graph-based data structure.

## 3 Left-Corner Parsing as Top-Down Parsing

Remember from the discussion in Cha. 2 that parsers can be viewed as algorithms for constructing intersection grammars in an incremental fashion. From this perspective, a parser is a particular kind of map from grammars to grammars, which is also called a *grammar transform*. We will now look at another instance of this idea: an LC parser for grammar $G$ is a top-down parser operating on the *left-corner transform* of $G$ (Rosenkrantz and Lewis II. 1970; Aho and Ullman 1972).

Intuitively, the left-corner transform rotates trees by 90 degrees to the right so that the bottom left corner becomes the top left corner (cf. Fig. 6.1). As a result, left corners end up c-commanding their mother as well as their right siblings. Thanks to their new structural prominence, left corners are now also conjectured by the top-down parser before the other nodes. While this may sound rather confusing, the left corner transform is actually very easy to define.

---

*Definition 6.2.* Let $G := \langle N, T, S, R\rangle$ be a CFG. The *left-corner transform* of $G$ is the CFG $G^{\text{lc}} := \langle N', T, S, R'\rangle$ such that

| | |
|---|---|
| $A \rightarrow\ a\ A\text{-}a$ | for all $A \in N$ and $a \in T$ |
| $A \rightarrow\ A\text{-}B$ | for all $A \in N$ and $B \rightarrow \varepsilon \in R$ |
| $A\text{-}X \rightarrow\ \beta\ A\text{-}B$ | for all $A \in N$ and $B \rightarrow X\beta \in R$ |
| $A\text{-}A \rightarrow \varepsilon$ | for all $A \in N$ |

---

**Example 6.2 Left-Corner Transform of our Example Grammar**

Our toy example grammar consists of

- the non-terminals S, NP, VP, Det, N, PN, Vi, and Vt,
- the terminals a, the, car, truck, anvil, Bugs, Daffy, fell over, hit,
- the ten rewrite rules listed below.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

We now apply the left-corner transform. First we have to add a rule of the form $A \rightarrow aA\text{-}a$ for all $A \in N$ and $a \in T$. Even with our small toy grammar that is a lot of rules.

| | | | | | |
|---|---|---|---|---|---|
| S | → | a S-a | S | → | the S-the |
| S | → | car S-car | S | → | truck S-truck |
| S | → | anvil S-anvil | S | → | Bugs S-Bugs |
| S | → | Daffy S-Daffy | S | → | fell_over S-fell_over |
| S | → | hit S-hit | | | |
| NP | → | a NP-a | NP | → | the NP-the |
| NP | → | car NP-car | NP | → | truck NP-truck |
| NP | → | anvil NP-anvil | NP | → | Bugs NP-Bugs |
| NP | → | Daffy NP-Daffy | NP | → | fell_over NP-fell_over |
| NP | → | hit NP-hit | | | |
| VP | → | a VP-a | VP | → | the VP-the |
| VP | → | car VP-car | VP | → | truck VP-truck |
| VP | → | anvil VP-anvil | VP | → | Bugs VP-Bugs |
| VP | → | Daffy VP-Daffy | VP | → | fell_over VP-fell_over |
| VP | → | hit VP-hit | | | |
| Det | → | a Det-a | Det | → | the Det-the |
| Det | → | car Det-car | Det | → | truck Det-truck |
| Det | → | anvil Det-anvil | Det | → | Bugs Det-Bugs |
| Det | → | Daffy Det-Daffy | Det | → | fell_over Det-fell_over |
| Det | → | hit Det-hit | | | |
| N | → | a N-a | N | → | the N-the |
| N | → | car N-car | N | → | truck N-truck |
| N | → | anvil N-anvil | N | → | Bugs N-Bugs |
| N | → | Daffy N-Daffy | N | → | fell_over N-fell_over |
| N | → | hit N-hit | | | |
| PN | → | a PN-a | PN | → | the PN-the |
| PN | → | car PN-car | PN | → | truck PN-truck |
| PN | → | anvil PN-anvil | PN | → | Bugs PN-Bugs |
| PN | → | Daffy PN-Daffy | PN | → | fell_over PN-fell_over |
| PN | → | hit PN-hit | | | |
| Vi | → | a Vi-a | Vi | → | the Vi-the |
| Vi | → | car Vi-car | Vi | → | truck Vi-truck |
| Vi | → | anvil Vi-anvil | Vi | → | Bugs Vi-Bugs |
| Vi | → | Daffy Vi-Daffy | Vi | → | fell_over Vi-fell_over |
| Vi | → | hit Vi-hit | | | |
| Vt | → | a Vt-a | Vt | → | the Vt-the |
| Vt | → | car Vt-car | Vt | → | truck Vt-truck |
| Vt | → | anvil Vt-anvil | Vt | → | Bugs Vt-Bugs |
| Vt | → | Daffy Vt-Daffy | Vt | → | fell_over Vt-fell_over |
| Vt | → | hit Vt-hit | | | |

Our grammar contains no rules that rewrite a non-terminal as the empty string, so we can skip the second step of the transform. Next we have to add rules of the form $A\text{-}X \rightarrow \beta\ A\text{-}B$ for all non-terminals $A$ and every $B$ such that $B \rightarrow X\ \beta$. We only list the rewrite rules for S, but exactly the same rules exist for every other non-terminal.

| | | | | | |
|---|---|---|---|---|---|
| S-NP | → | VP S-S | S-a | → | S-Det |
| S-PN | → | S-NP | S-the | → | S-Det |
| S-Det | → | N S-NP | S-car | → | S-N |
| S-Vi | → | S-VP | S-truck | → | S-N |
| S-Vt | → | NP S-VP | S-anvil | → | S-N |
| | | | S-Bugs | → | S-PN |
| | | | S-Daffy | → | S-PN |
| | | | S-fell_over | → | S-Vi |
| | | | S-hit | → | S-Vt |

Finally, we add *A*-*A* → $\varepsilon$ for all non-terminals *A* of the original grammar.

| | | | | | |
|---|---|---|---|---|---|
| S-S | → | $\varepsilon$ | Det-Det | → | $\varepsilon$ |
| NP-NP | → | $\varepsilon$ | N-N | → | $\varepsilon$ |
| VP-VP | → | $\varepsilon$ | PN-PN | → | $\varepsilon$ |
| | | | Vi-Vi | → | $\varepsilon$ |
| | | | Vt-Vt | → | $\varepsilon$ |

This is an enormous grammar with a confusing arrangements of rules, which is furthermore exacerbated by the unfortunate fact that most of these rules cannot occur in a well-formed derivation. Fortunately there are algorithms for pruning away such useless rules (cf. Grune and Jacobs 2008). The compacted grammar is still much bigger than the original though.

Even an extended look at the transformed grammar in the example above does not readily reveal how it relates top-down parsing to LC parsing. However, things become clearer when one compares which tree the original grammar generates for *The anvil hit Daffy* to the tree licensed by the transformed grammar. Both trees are shown in Fig. 6.1. When the transformed tree is traversed in the fashion of a recursive descent parser, it closely lines up with the LC parse table for *the anvil hit Daffy* in example 6.1. Figure 6.2 depicts this in full detail. Closer analysis of the tree also reveals that the names of the non-terminals were not chosen arbitrarily. Instead, they follow a simple pattern were the hyphen serves a similar function as the dot in our items: recognized material is to the right of the hyphen, conjectured material to its left (note that this is the exact reverse of the order used in our parse items). The node S-NP, for instance, encodes the fact that we are conjecturing an S and have successfully identified its NP subtree. A non-terminal *X* without hyphen is simply shorthand for *X*-. That is to say, it represents the fact a node has been conjectured but no part of its subtree has been recognized yet.

*Exercise 6.8.* Now that you know how to interpret the non-terminal symbols, go back to the definition of the left-corner transform and provide an intuitive explanation for each one of the four rule templates. ⊙

Decomposing LC parsing into top-down parsing with a left corner transform may seem like little more than a technical trick — a very impressive one, admittedly, yet merely a trick. But just like intersection parsing, this abstracted perspective has a lot to offer. On a practical level, any kind of algorithmic improvements for recursive descent

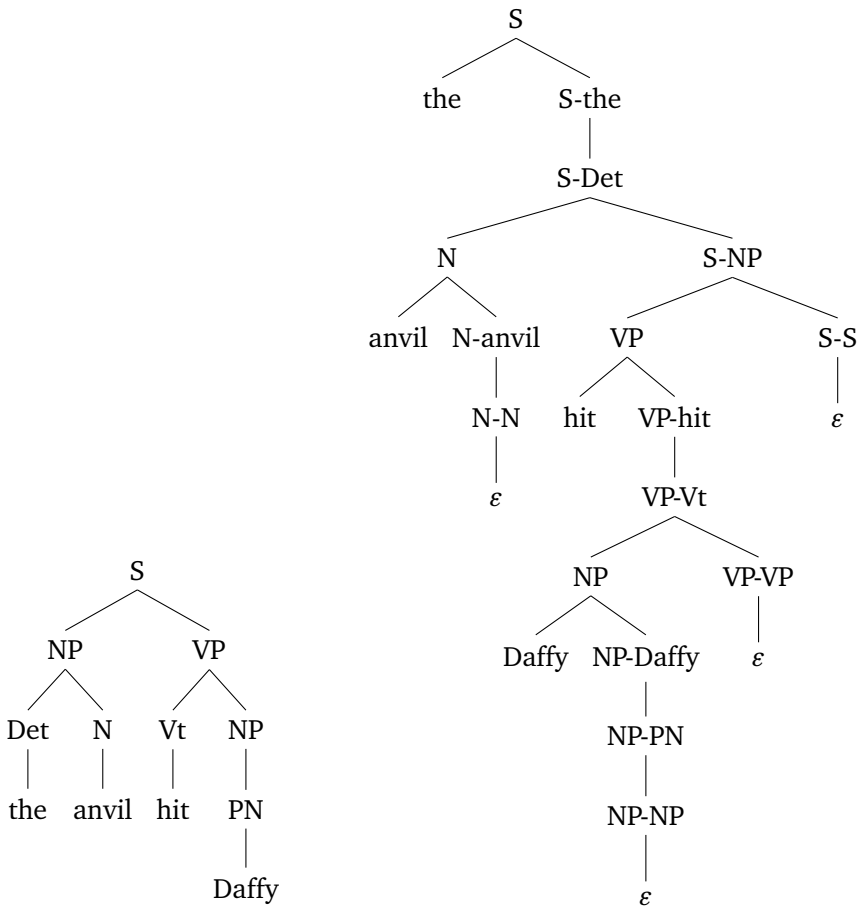



Figure 6.1: Parse tree for *The anvil hit Daffy* with original and transformed grammar

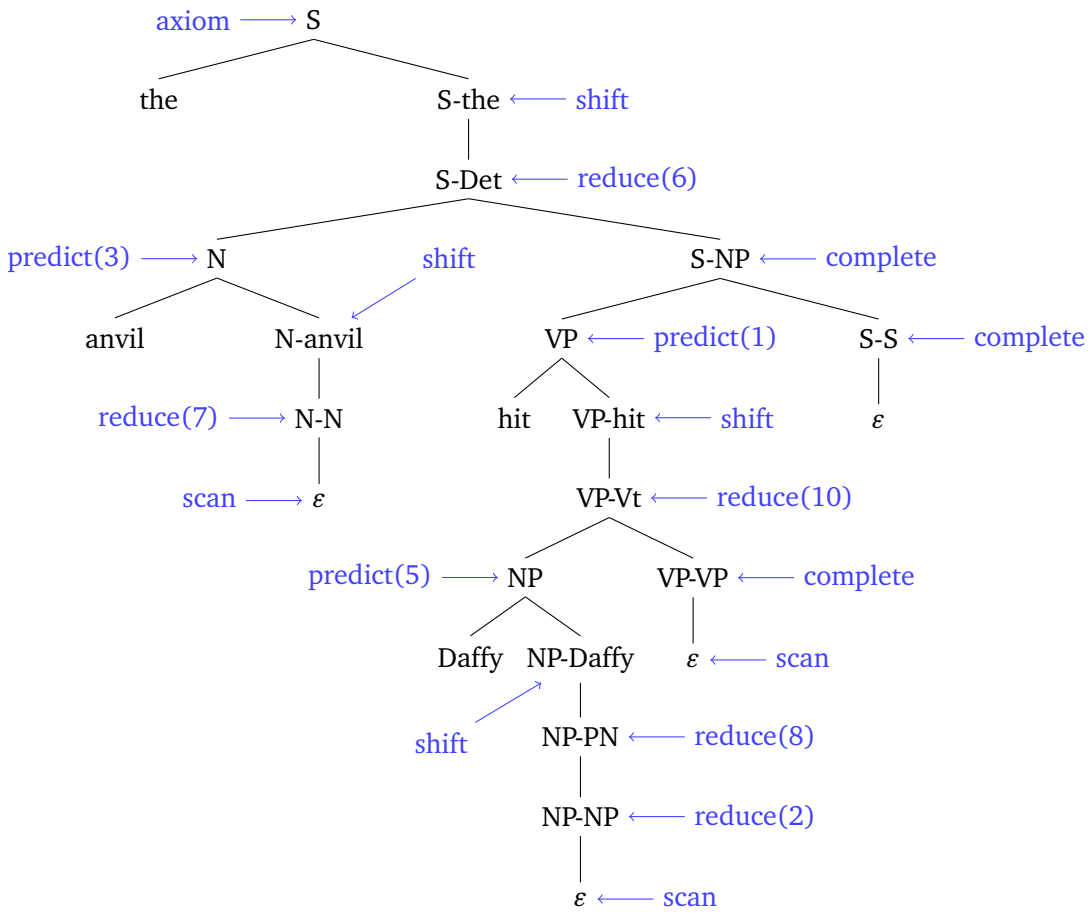



Figure 6.2: Recursive descent traversal of the parse tree closely mirrors the steps of an LC parser

parsers can be carried over to LC parsers via the left-corner transform, which is a lot simpler than adapting the top-down techniques to LC parsing (for a very different application see Johnson 1996). Quite generally top-down parsers are much more versatile and better understood than LC parsers. For example, top-down parsers have been extended greatly beyond the bounds of CFGs whereas no simple yet general notion of LC parsing exists for formaslisms that are more powerful than CFGs. Recent theorems show, however, that these formalisms still have a context-free backbone and thus it should be possible to parse them left-corner style by applying the left-corner transform to said context-free backbone. These and many other insights would be much harder to obtain without the left-corner transform.

The left-corner transform also has the advantage of being entirely non-destructive: one can always retrieve the original tree structure from the transformed parse tree. Any algorithm that can construct the original phrase structure tree from the left corner parse table can be extended to work over the left-corner transformed parse trees instead since the latter contains all the information of the former. The structural changes to the grammar are merely a result of shifting parts of the parser directly into the grammar, we are not forced into granting these structures any kind of cognitive reality at the level of grammatical description.

*Exercise 6.9.* Define a shift-reduce transform such that a top-down parser operating over the shift-reduce transform of grammar $G$ will explore the nodes in the order that corresponds to a shift-reduce parse table for any given input sentence that can be generated by $G$. ⊙

*Exercise 6.10.* Given that we can decompose a variety of parsers into a recursive descent parser coupled with a grammar transform, what are we to make of claims that the human parser must follow a specific strategy given the experimental evidence? Can the two more easily be reconciled under a view where grammar and parser are distinct cognitive objects, or one where the grammar is an abstraction of the parser (cf. the discussion in Chapter 1)? ⊙

# 4 Psycholinguistic Adequacy

## 4.1 Garden Paths

Since LC parsers combine top-down and bottom-up techinques of structure-building — both of which struggled with garden path sentences — one would also expect the LC parser to predict that garden path sentences are hard. This is indeed the case, as can be seen in the parse history in Fig. 6.3. Note that our toy grammar for garden paths has no lexical ambiguity, so the predictions are independent of whether the LC parser uses top-down filtering. It is also instructive to compare this parse history to the ones for the recursive descent parser and the shift reduce parser on pages 39 and 54, respectively. With the same preferences for rewrite rules, the LC parser finds the right parse after three failures whereas recursive descent has at least nine failures and shift reduce still five.

*Exercise 6.11.* Explain how the LC parser avoids some of the failed parses that are entertained by the recursive descent parser or the shift reduce parser. ⊙

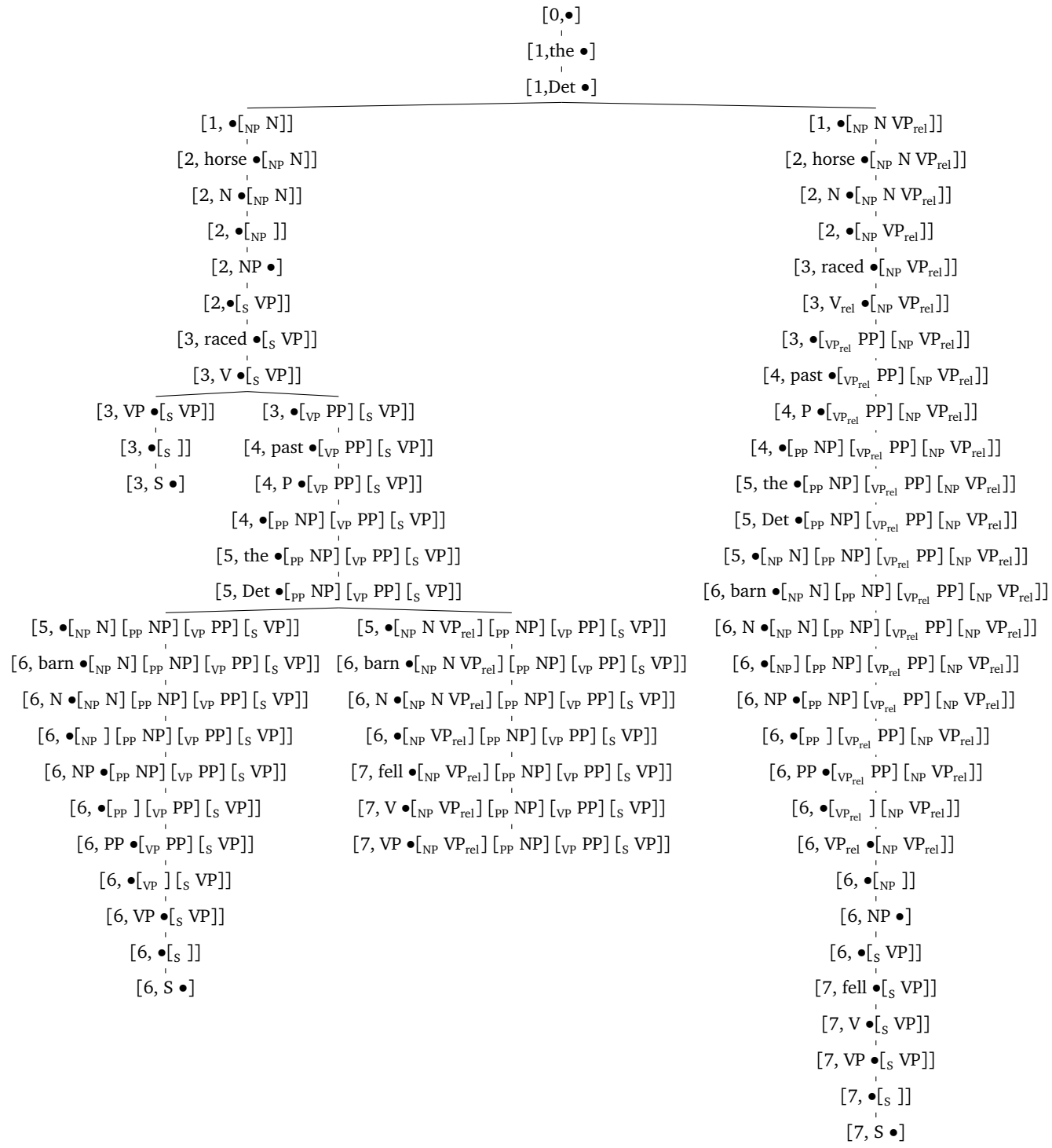



Figure 6.3: Parse history for LC parse of *the horse raced past the barn fell*; the parser moves through the history in a recursive descent fashion

## 4.2 Left Recursion and Left Embeddings

Left recursion was a problem for top-down parsing because it can prevent the parser from terminating (if the parser has to find all possible parses rather than just one). This is not an issue with the LC parser thanks to the bottom-up restrictions on the prediction steps.

**Example 6.3 LC Parse of Nested Possessive DPs**

The table below shows a successful parse for *John's father's car's exhaust pipe disappeared* with the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| S | → | DP VP | Poss | → | 's |
| DP | → | DP D′ \| PN | N | → | father \| car \| exhaust pipe |
| D′ | → | Poss NP | PN | → | John |
| NP | → | N | V | → | disappeared |
| VP | → | V | | | |

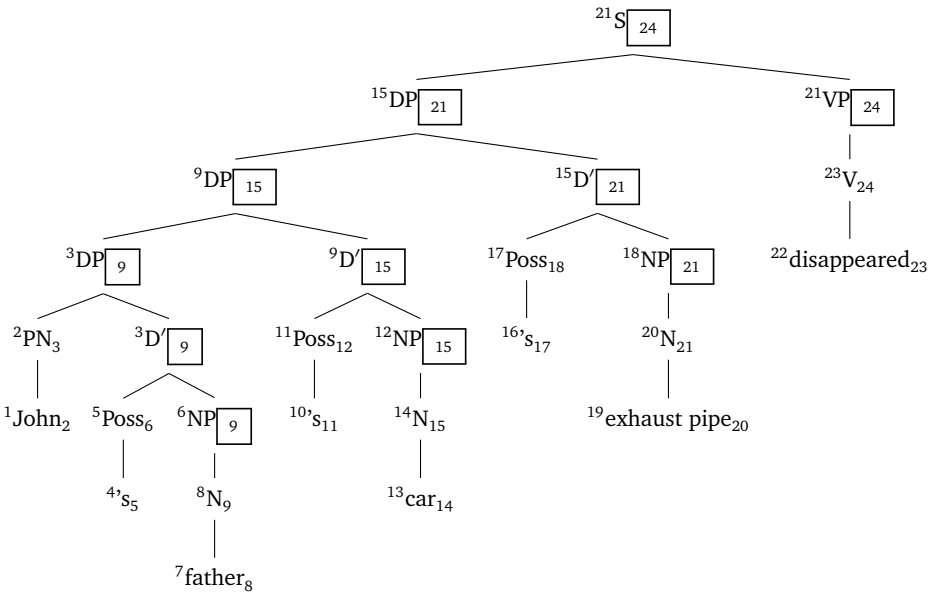
| parse item | inference rule |
|---|---|
| [0, •] | axiom |
| [1, John •] | shift |
| [1, PN •] | reduce |
| [1, DP •] | reduce |
| [1, •[$_{DP}$ D′]] | predict |
| [2, 's •[$_{DP}$ D′]] | shift |
| [2, Poss •[$_{DP}$ D′]] | reduce |
| [2, •[$_{D'}$ NP] [$_{DP}$ D′]] | predict |
| [3, father •[$_{D'}$ NP] [$_{DP}$ D′]] | shift |
| [3, N •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [3, NP •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [3, •[$_{D'}$ ] [$_{DP}$ D′]] | scan |
| [3, D′ •[$_{DP}$ D′]] | complete |
| [3, •[$_{DP}$ ]] | scan |
| [3, DP •] | complete |
| [3, •[$_{DP}$ D′]] | predict |
| [4, 's •[$_{DP}$ D′]] | shift |
| [4, Poss •[$_{DP}$ D′]] | reduce |
| [4, •[$_{D'}$ NP] [$_{DP}$ D′]] | predict |
| [5, car •[$_{D'}$ NP] [$_{DP}$ D′]] | shift |
| [5, N •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [5, NP •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [5, •[$_{D'}$ ] [$_{DP}$ D′]] | scan |
| [5, D′ •[$_{DP}$ D′]] | complete |
| [5, •[$_{DP}$ ]] | scan |
| [5, DP •] | complete |
| [6, 's •[$_{DP}$ D′]] | shift |
| [6, Poss •[$_{DP}$ D′]] | reduce |
| [6, •[$_{D'}$ NP] [$_{DP}$ D′]] | predict |

| | |
|---|---|
| [7, exhaust pipe •[$_{D'}$ NP] [$_{DP}$ D′]] | shift |
| [7, N •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [7, NP •[$_{D'}$ NP] [$_{DP}$ D′]] | reduce |
| [7, •[$_{D'}$ ] [$_{DP}$ D′]] | scan |
| [7, D′ •[$_{DP}$ D′]] | complete |
| [7, •[$_{DP}$ ]] | scan |
| [7, DP •] | complete |
| [7, •[$_{S}$ VP]] | predict |
| [8, disappeared •[$_{S}$ VP]] | shift |
| [8, VP •[$_{S}$ VP]] | reduce |
| [8, •[$_{S}$ ]] | scan |
| [9, S •] | complete |

Pay close attention to how the parser constantly goes through the same sequence of inference rules for each DP layer: a possessive DP is constructed and discharged via the left-corner prediction of another DP.

*Exercise 6.12.* In Sec. 3 we looked at the LC parser more abstractly as a top-down parser that operates on the left-corner transform of the original grammar. This view is actually very illuminating regarding why LC parsers do not struggle with left recursion. Construct the left-corner transform of the grammar and draw the tree it assigns to the example sentence. What is striking about this tree compared to the original one, and how does this further our understanding of left recursion in LC parsing? ⊙

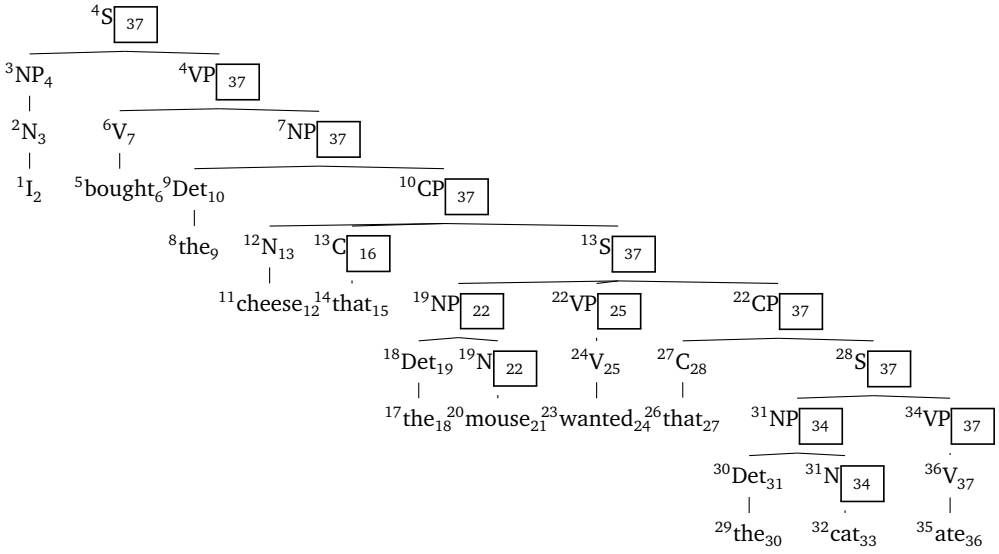
Since the LC parser completely finishes a possessive DP before starting the next higher one, it behaves similar to the shift reduce parser for left embeddings, with memory load remaining mostly unaffected by the number of embeddings. This can also be seen in the annotated parse tree, which may have a high payload but maximum tenure is always 6 irrespective of the number of embedded DPs.




## 4.3 Center Embedding and Right Embedding

The behavior of the LC parser for center embedding and right embedding is not as easy to evaluate as left embedding. Let us look at right embedding first. Given what we have said about LC parsing so far, it should not be too hard to see that memory load increases with right embedding. That's because a node is introduced at the same time as its right daughters but cannot be completed until they are. So the bigger the right daughters of a node, the higher its tenure.

**Example 6.4** Run of standard LC parser over right embedding sentence



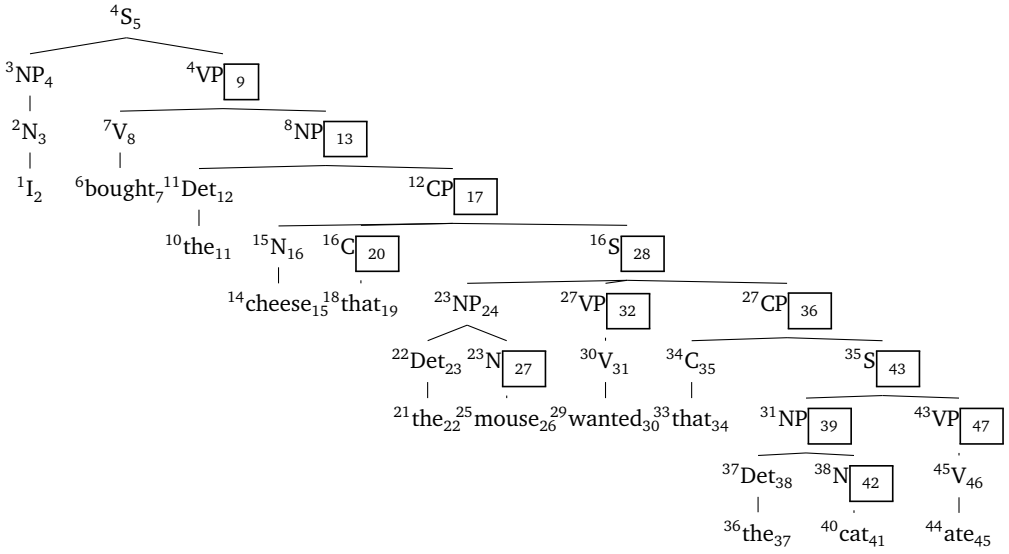

The payload of right embedding is enormous with an LC parser, and tenure also increases with every level of embedding.

The problem is that the LC parser is extremely conservative in how it discharges hypothesis. A node with right daughters $d_1, \ldots, d_n$ is held in memory until each $d_i$ has been recognized bottom-up. This contrasts quite sharply with the prediction step, where the parser considers just the leftmost daughter $d_0$ sufficient evidence for positing a hypothesis. Suppose, on the other hand, that the LC parser could also use hypothesis in the scan steps so that a node can be discharged as soon as all its daughters have been predicted.

$$\textbf{Eager Scan} \qquad \frac{[i, \alpha \bullet [_{N_1} \delta_1] \cdots [_{N_k} \delta_k][_M \ \gamma] \ \beta]}{[i, \alpha \bullet [_{N_1} \delta_1] \cdots [_{N_k} \delta_k] \ \beta]} \ \gamma := N_1 \cdots N_k$$

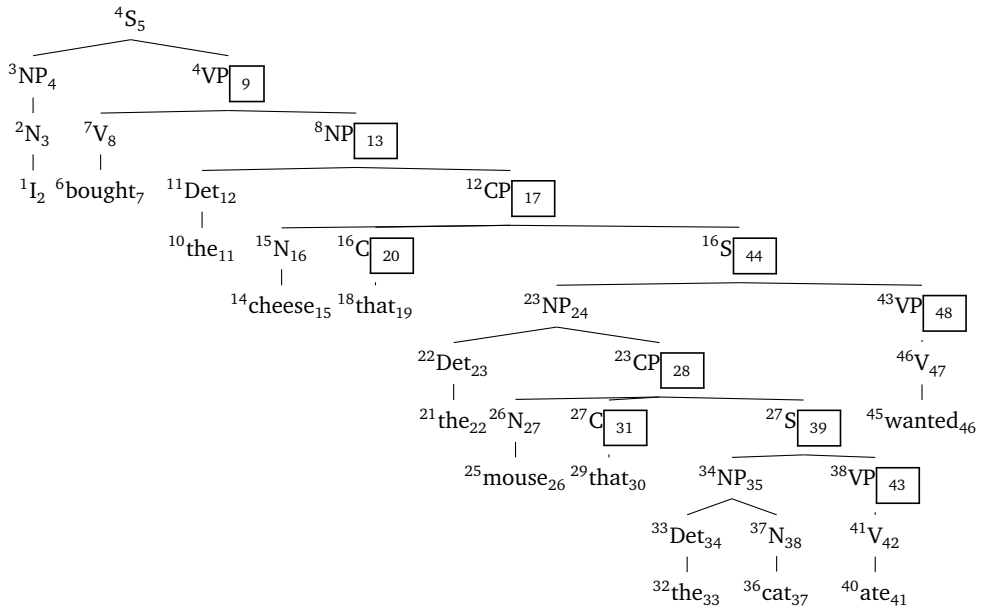
Resnik (1992) studies this type of LC parser in detail and calls it *arc-eager* (the intuition being that if there are conjectured nodes than can be linked by an arc, the parser establishes that arc immediately). Resnik concludes that an arc-eager LC parser predicts both left embedding and right embedding due to their small memory load, whereas center embedding is correctly predicted to be much harder. We can confirm this claim using a modified parse annotation scheme to represent the arc-eager LC parse.

**Example 6.5** Run of arc-eager LC parser over right embedding sentence




As you can see arc-eagerness has no effect on the payload, but it greatly reduces the maximum tenure, which is now a function of how many daughters and left siblings a node has. The size of those siblings and daughters, however, is completely irrelevant.

**Example 6.6** Run of arc-eager LC parser over center embedding sentence




At first sight the center embedding parse does not look too different from the right embedding variant. In fact, right embedding even looks more challenging than center embedding since the payload is slightly higher (10 for center embedding, 12 for right embedding). The real difference, however, is maximum tenure. The S nodes in the center embedding sentence have a tenure of 12 and 28, respectively, whereas their right embedding analogues only have a tenure of 8 and 12.

*Exercise 6.13.* There is a subtle but important discrepancy between the LC parser with eager scan and the arc-eager one depicted above. As the eager scan rule completely eliminates $M$, it can no longer serve in any left-corner prediction. To some extent that state of affairs is very welcome: if we could use $M$ immediately for a left-corner prediction right after eager scan, the nodes predicted this way would appear immediately to the right of the dot. Consequently, the parser would work on them first before it has even verified that the daughters of $M$ exist. Not only would this defeat the point of left-corner parsing, it would also produce very high tenure for the daughters of $M$.

In the examples above, I use a particular strategy where LC predictions of $M$ are triggered by the last recognized node in the subtree routed by $M$, which is always the rightmost daughter of $M$. Complete the parsing schema for the arc-eager LC parser by extending the predict rule in this fashion. ⊙

The findings of Resnik (1992) have led many researchers to believe that LC parsers are the ideal model of human sentence processing. But we should not accept this view too readily. First of all, arc-eagerness is essential to get the right memory-load for the three types of embedding. But as Resnik himself points out in a brief remark at the end of the paper, an LC parser that always operates in an arc-eager fashion is not complete since it fails to find a parse for simple sentences like *John met Mary yesterday*.

*Exercise 6.14.* Explain why this is necessarily the case under the assumption that *yesterday* is a VP-adjunct and the parser can only make arc-eager inferences. ⊙

Even if one posits a more sophisticated control structure hard problems arise. Suppose that both arc-eager and standard LC inferences are available but the former are preferred over the latter in the control structure. Then a very simple sentence like *John met Mary briefly yesterday with a friend at a party* would require a massive parse history to be built that rivals that of garden paths. The reason for that is simple: for each adjunct, the parser has a choice between arc-eager or standard inference, only the latter of which yields a successful parse. If arc-eager is the default, then only the last out of all options will yield the right parse. With 4 adjuncts, there are $2^4 = 16$ parses to explore, so the parser will fail 15 times before finding the right parse. This is much worse than what we saw in our discussion of garden paths. So either our VP-adjunct example is incorrectly predicted to be a garden path sentence, too, or we lose our account of garden paths.

One idea might be that the human parser has a control structure that is exceedingly good at estimating the risk of an arc-eager inference in a given syntactic context and prefers standard inference in these cases. In a certain sense, this extends the idea of a GLC parser, where specific rules are used more or less predictively, and extends it to the control structure. This would be a very complicated and overly expressive model, though. With that many parameters to tune, there is little doubt that almost any processing effect can be accounted for. This is not a good situation to be in, for if the model can account for absolutely anything, it tells us absolutely nothing. If you find that remark puzzling, just keep in mind that this is the very same reason we put strong restrictions on our linguistic theories, be it in phonology, morphology or syntax. We do not want descriptions, we want generalizations and predictions. A weak formalism makes very strong predictions, an overly malleable one does not.

*Exercise 6.15.* Are merely local syntactic coherence effects expected with a standard LC parser? What about GLC parsers or variants with top-down filtering? ⊙

## References and Further Reading

Aho, Alfred V., and Jeffrey D. Ullman. 1972. *The theory of parsing, translation and compiling; volume 1: Parsing*. Englewood Cliffs: Prentice Hall.

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

Johnson, Mark. 1996. Left corner transforms and finite state approximations. *Ms., Rank Xerox Research* .

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

Rosenkrantz, Stanley J., and Philip M. Lewis II. 1970. Deterministic left corner parser. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata*, 139–152.

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

# Lecture 7

# The Importance of Data Structures

Remember that we analyze parsers as the combination of three components: a parsing schema, a control structure, and a data structure. While parsing schema and certain aspects of the control structure have occupied a lot of our attention, we haven't talked much about data structures so far. Our psycholinguistic evaluations used a prefix tree as a representation of the parse space and how it might be explored by the parser, but we did not assume that the parser actually uses a prefix tree to store information. However, we did propose that the control structure might operate on a priority queue as an encoding of which parse to explore next. A priority queue is a form of data structure as it holds the parse items inferred by the parser. But it cannot be the full data structure: since items are frequently removed from the queue, it does not provide a permanent record of the actual parse. One option would be to build and store each individual parse trees in parallel with the parse, but as we will see this is too inefficient even for computers, whose working memory vastly outstrips that of humans. It is about time, then, that we take a more careful look at data structures and the parsers that make use of them.

## 1 Why Data Structures Matter

Recall from Chap. 1 that we distinguish between parsers and recognizers. A recognizer only has to determine for a given sentence whether it is well-formed, whereas a parser also has to find the right tree structures.

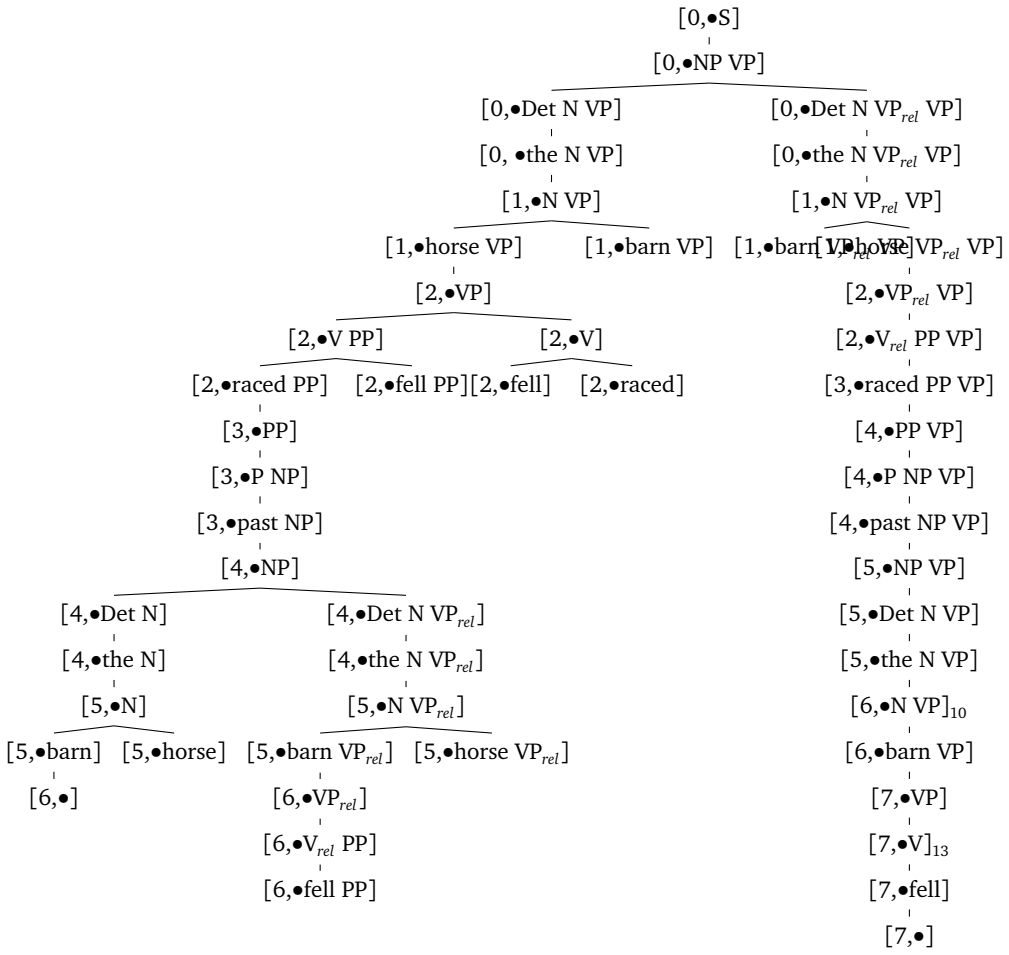
One might think that data structures are less of a concern for recognizers. Any one of our parsing schema coupled with a priority queue yields a working recognizer. The fact that the priority queue does not store parse items after they have been used in an inference rule is irrelevant since the only thing that matters is whether the priority queue contains a parse item that is a goal for the input sentence. As soon as this condition is satisfied, the recognizer can stop and declare the input sentence well-formed.

Even for a recognizer, though, there are downsides to not storing previous items. Most problematically, the recognizer may accidentally enter a loop if there are two items $I$ and $J$ such that each one can be inferred from the other. If neither $I$ nor $J$ allow the sentence to be recognized, and the control structure is designed in such a way that $I$ and $J$ are preferred to other items that would eventually lead to successful recognition of the input, then the recognizer will try the $I$-route and the $J$-route over and over again. In less severe cases the recognizer does not enter a loop but ends up

exploring partial parses that have already turned out to be failures at an earlier point.

**Example 7.1 Needlees Reexploration of Failed Partial Parses**

The parse history of the recursive descent parser/recognizer for the garden path sentence *the horse raced past the barn* includes several cases of redundancy.




After the failed exploration of the left-most branch, the recognizer backtracks and uses [5,•N] to infer [5,•horse] instead of [5,•barn]. Since the word at position 5 is *horse*, not *barn*, this branch obviously fails, too. What more, we can infer that every parse item of the form [5,•horse $\gamma$] will lead to failure. Yet the parser attempts the very same thing in the next branch with [5,•horse VP$_{rel}$]. Later on, a similar instance of redundancy arises when [1, •barn VP$_{rel}$ VP] is conjectured even though [1,•barn VP] has already failed.

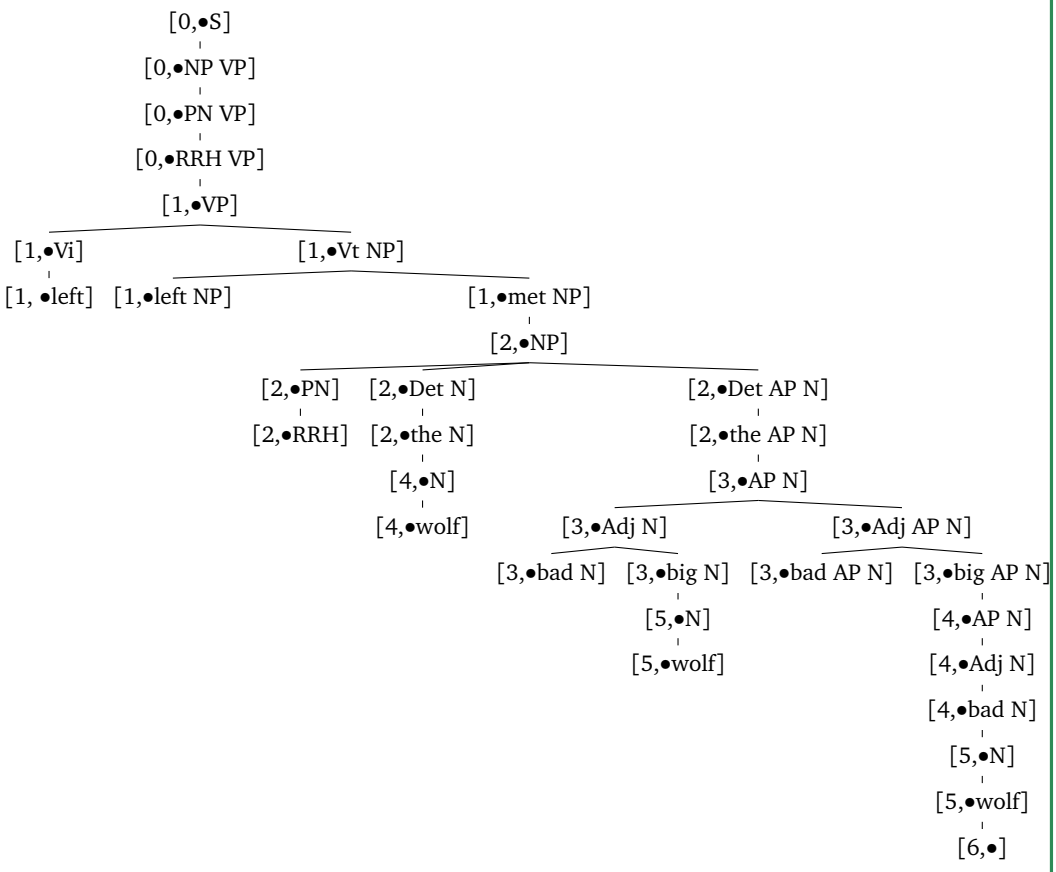
If parse items are discarded immediately after being used in an inference rule, the recognizer also loses the ability to recycle successful partial parses.

**Example 7.2 Needless Reexploration of Successful Partial Parses**

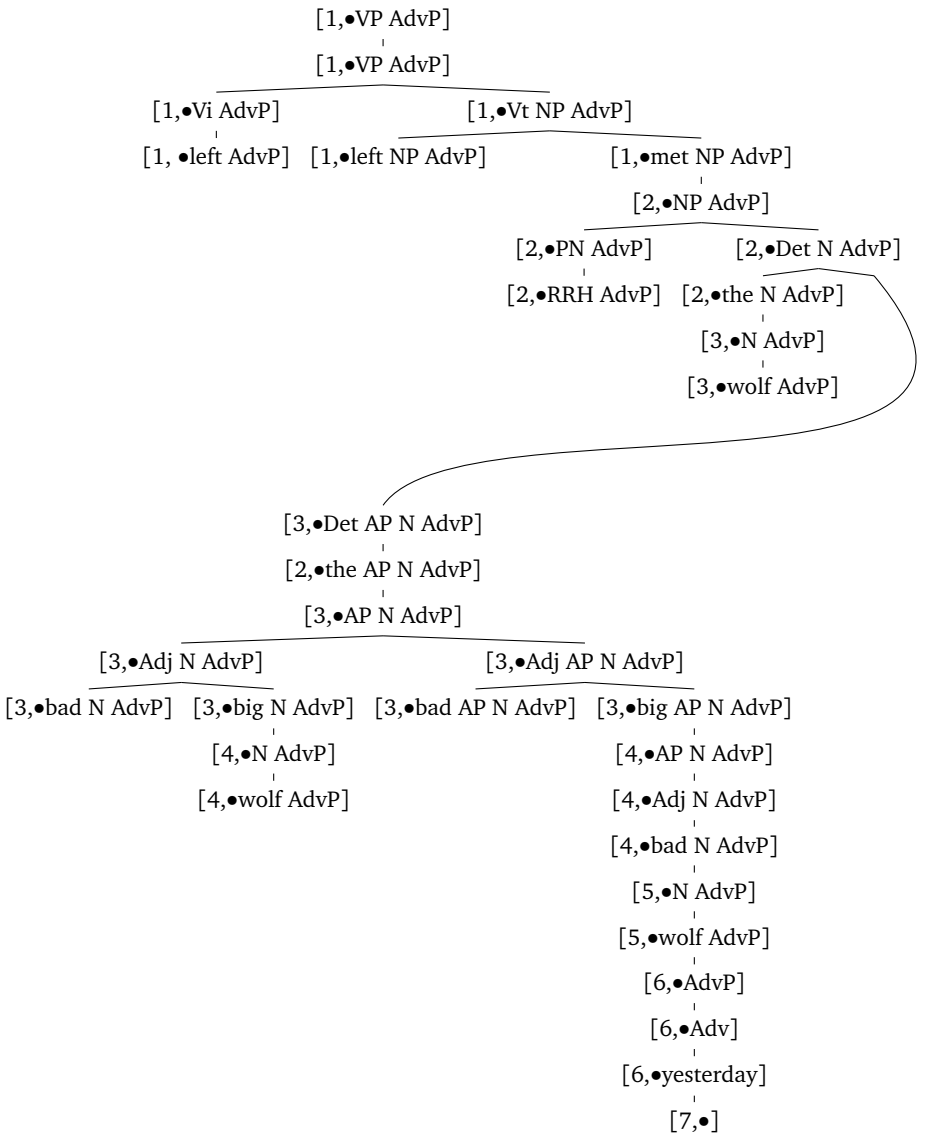
Suppose the sentence *Red Riding Hood met the big, bad wolf yesterday* is to be recognized using the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| S | → | NP VP | PN | → | Red Riding Hood |
| NP | → | PN \| Det N \| Det AP N | Det | → | the |
| AP | → | Adj \| Adj AP | N | → | wolf |
| VP | → | Vi \| Vt NP \| VP AdvP | Adj | → | bad \| big |
| AdvP | → | Adv | Adv | → | yesterday |
| | | | Vi | → | left |
| | | | Vt | → | left \| met |

Assume furthermore that the linear order of disjunctive right-hand sides encodes their priority in the control structure. Then a recursive descent recognizer will first go through the options below rewriting VP as Vi or Vt NP, neither of which produces a goal item.




As before we see that the recognizer repeats inferences that have already failed before, e.g. [1,•left NP] after [1,•left]. A bigger problem is revealed once we look at the parse history after the recognizer correctly infers [1,•VP AdvP] from [1,•VP].




This subtree of the parse history is virtually isomorphic to the one rooted in [1,•VP] — the only difference is that the last branch expands the AdvP and finally reaches a goal item. What this means is that the recognizer does not reuse any of the information it collected on its first attempt to build the VP. It has to verify again that the verb is *met*, and it builds the NP for *big, bad wolf* from scratch even though it had already been correctly recognized in the [1,•VP] subtree.

This shows that even for recognizers a good data structure is essential to

- detect and avoid loops,
- skip inferences that have failed before,
- reuse successful inferences.

Once a useful data structure is in place, using it as a record of parse trees and thereby expanding the recognizer to a parser is just a minor step.

## 2 Chart Parsing

chart agenda dynamic programming/tabulation/memoization

## 3 CKY

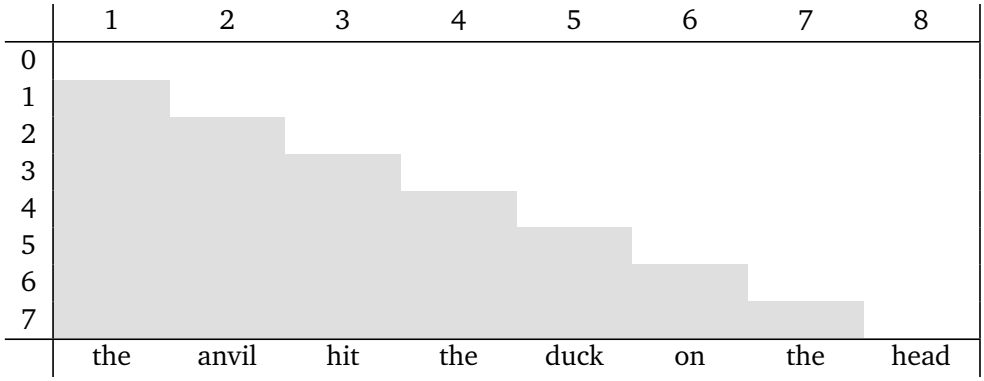
### 3.1 Intuition

The Cock-Younger-Kasami algorithm — usually abbreviated CYK or CKY — is not only the best-known chart parsing algorithm, it is also the most popular parser in computational linguistics. That's probably due to its high efficiency coupled with its conceptual simplicity. The idea behind the CKY parser is indeed very simple: we read in the entire input string and then carry out all possible bottom-up reductions in parallel. In a certain sense, the CKY parser behaves like a shift-reduce parser where I) we always apply shift until we have reached the end of the string, and II) parse items can participate in multiple reduction steps.

The CKY parser is usually specified in the format of a chart parser. Suppose we have the input string *the anvil hit the duck on the head*, which should be analyzed with an extended version of our usual toy grammar.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1) | S | → | NP VP | 9) | Det | → | a \| the |
| 2) | NP | → | PN | 10) | N | → | car \| truck \| anvil \| duck \| hit |
| 3) | NP | → | Det N | 11) | PN | → | Bugs \| Daffy |
| 4) | NP | → | NP PP | 12) | P | → | on |
| 5) | PP | → | P NP | 13) | Vi | → | fell over \| duck |
| 6) | VP | → | Vi | 14) | Vt | → | hit |
| 7) | VP | → | Vt NP | | | | |
| 8) | VP | → | VP PP | | | | |

Our first step is to draw a matrix that has as many columns and rows as there are words in the input. We will only use the fields above the diagonal, so the others are shaded out. It will be convenient to refer to individual cells by their address $(i, j)$, where $i$ is the row number and $j$ the column number. The idea is that each cell $(i, j)$ represents the substring of the input spanning from position $i$ to $j$.




For each word $w_i$ in the input, we enter all possible parts of speech in the cell $(i, i+1)$.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | Det | | | | | | | |
| 1 | | N | | | | | | |
| 2 | | | N,Vt | | | | | |
| 3 | | | | Det | | | | |
| 4 | | | | | N,Vi | | | |
| 5 | | | | | | P | | |
| 6 | | | | | | | Det | |
| 7 | | | | | | | | N |
| | the | anvil | hit | the | duck | on | the | head |

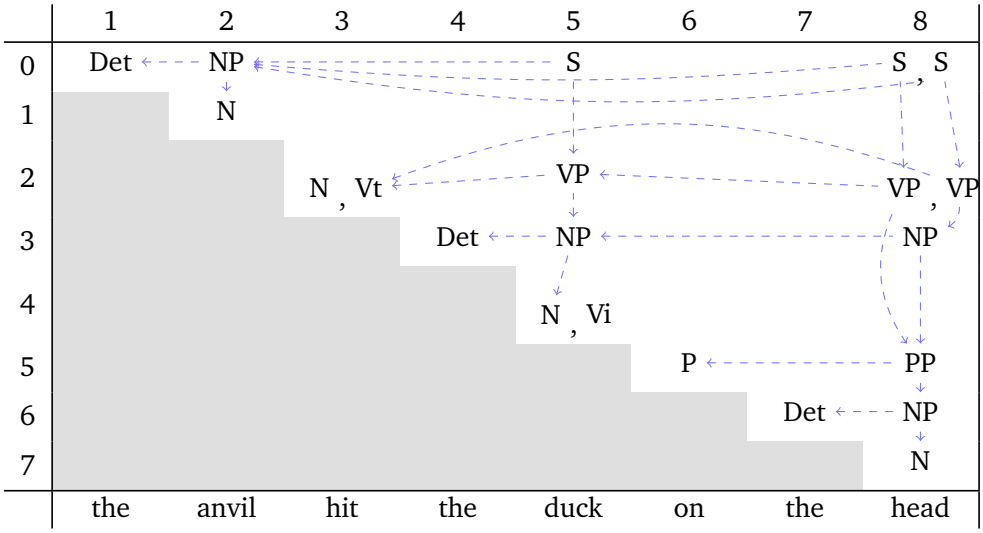
In order to fill in the remaining cells, we use the following algorithm: if cell $(i, j)$ contains category $B$ and cell $(j, k)$ contains category $C$ and $A \rightarrow BC$ is a rewrite rule of the grammar, then enter $A$ in cell $(i, k)$.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | Det | NP | | | S | | | S |
| 1 | | N | | | | | | |
| 2 | | | N,Vt | | VP | | | VP |
| 3 | | | | Det | NP | | | NP |
| 4 | | | | | N,Vi | | | |
| 5 | | | | | | P | | PP |
| 6 | | | | | | | Det | NP |
| 7 | | | | | | | | N |
| | the | anvil | hit | the | duck | on | the | head |

The input is well-formed iff the top-right cell contains the start category S. In other words, there is a constituent labeled S that spans the entire input string.

With a chart like the one above, the CKY algorithm is just a recognizer as it is not always obvious how specific cell values were derived from others. For instance, the cell $(3, 5)$ contains the value NP, but there is no indication whether this NP consists of Det N or Det V. Similarly, the VP in cell $(2, 8)$ could be the result of combining either the V in $(2, 3)$ with the NP in $(3, 8)$ or the VP in $(2, 5)$ with the PP in $(5, 8)$. These alternatives correspond to very different structures: one where the PP is an NP adjunct, and another one with the PP as a VP adjunct.

In order to turn CKY from a recognizer into a parser, we have to modify the cell values such that each category comes with backpointers that indicate which other categories were used in the reduction. We can represent this pictorially by adding arrows to the chart.




## 3.2 Alternative Data Structures

You might have noticed right away that the chart with arrows between cells looks similar to a tree. While it is not exactly a tree, the chart can indeed be viewed as a directed acyclic graph (DAG) like in Fig. 7.1. A DAG is a set of nodes that are connected by branches that can only be followed in one direction (like the backpointers in the chart) and which are distributed in such a way that it is impossible to follow a branch out of a node and find a path back to that very same node. In linguistic parlance, a directed acyclic graph would be a collection of one or more multi-dominance trees, each one of which can have multiple roots.

---

*Definition 7.1 (DAG).* A *graph* $G$ is a pair $\langle V, E \rangle$ such that

- $V$ is a set of *vertices* (also called nodes),
- $E \subseteq V \times V$ of *edges* (or arcs, branches) is a binary relation.

A *directed acyclic graph* (DAG) is a graph that satisfies the following axioms:

- **Directedness**
  $E$ is asymmetric: $\langle x, y \rangle \in E \rightarrow \langle y, x \rangle \notin E$.
- **Acyclicity**
  The transitive closure of $E$ is irreflexive: $\langle x, x \rangle \notin E^{+}$.

---

DAGs are used in a variety of parsing algorithms. In fact, many of the most efficient parsers proposed since mid 80s use some kind of DAG-like data structure, e.g. the generalized LR parser, also known as the *Tomita parser* (Tomita 1986, 1987), and generalized left corner parsing (Nederhof 1993; not to be confused with the notion of generalized left corner parser discussed in Sec. 2.3 of Cha. 6). As you can see, the prefix *generalized* is commonly used to denote parsers with some kind of graph-based component.

Yet another data structure keeps the basic notion of a chart but makes it easier to work with. The standard CKY chart is somewhat odd in that all the cells below the diagonal are completely useless. This is cumbersome for implementations, where it

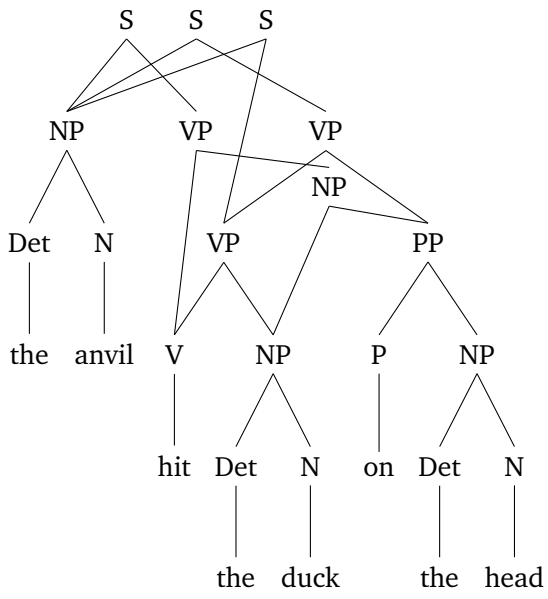



Figure 7.1: Directed acyclic graph representation of a CKY chart with backpointers

would be preferable if we could simply instantiate some kind of matrix and make full use of all its cells. Fortunately enough this is easy to accomplish.

The charts above are displayed as 2-dimensional objects where a cell contains zero or more categories. We can slightly modify this view by positing a third axis that lists all possible category symbols. Then an entry like N,V in cell $(4, 5)$ becomes a convenient shorthand for indicating that we have the value *true* in two cells, $(4, 5, \text{N})$ and $(4, 5, \text{V})$. Instead of a table, the chart is now a cube where the length of the $x$ and $y$-axes correspond to the length of the string and the $z$-axis is fixed by the number of categories in our grammar. Like any cube, we can rotate this 3-dimensional chart so that we are facing another one of its 6 sides. Suppose then that we topple this cube over so that the $x$-axis stays the same but the $z$-axis becomes the $y$-axis (making the old $y$-axis the new $z$-axis). When the cube is flattened back into a table at this point, we get a 2-dimensional chart where the $x$-axis still records the end point of substrings, but the $y$-axis is now just the list of categories in our grammar. In exchange, cells are no longer filled with categories but instead contain the starting index of substrings. Consequently, a cell at position $(\text{VP}, 8)$ with entry 2 means that the substring spanning from 2 to 8 can be reduced to a VP. Our example chart is repeated below using the new format.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| S | | | | | 0 | | | 0 |
| NP | | 0 | | | 3 | | | 3 |
| VP | | | | | 2 | | | 2 |
| PP | | | | | | | | 5 |
| Det | 0 | | | 3 | | | 6 | |
| N | | 1 | 2 | | 4 | | | 7 |
| P | | | | | | 5 | | |
| PN | | | | | | | | |
| Vi | | | | | 4 | | | |
| Vt | | | 2 | | | | | |
| | the | anvil | hit | the | duck | on | the | head |

This kind of table is much harder to decipher for humans, but it contains no wasted cells. In addition, it behaves like any matrix over natural numbers and thus can be manipulated using well-understood (and efficiently implemented!) operations like matrix multiplication.

These examples are just the tip of the iceberg, the number of viable data structures is myriad. That's why it is so important that we modularize parsers and study parsing schemata, control structures and data structures independently. If we had taken a purely algorithmic approach, then each one of these parsers would look very different because a graph must be handled very differently from a table. But as we will see next, the CKY parsing schema remains the same when we switch from charts to graphs or the other way round.

### 3.3 Formal Specification

The original CKY parsing algorithm combines two ideas: a mechanism for parallel bottom-up reduction and an efficient implementation of that idea via charts. But since both are rolled into one algorithmic specification, it is needlessly hard to make out for the uninitiated what CKY parsing is really about. As usual, the specification via a parsing schema is much more succinct and brings out the underlying logic more clearly.

In contrast to all the parsers we have seen so far, the CKY parser absolutely needs both indices in its parse items, so that the general form of parse items is $[i, \beta, j]$. As another distinguishing property we have multiple axioms instead of just one. For each rewrite rule $A \rightarrow a$ such that $a = w_i$, there is a corresponding axiom $[i, A, i+1]$. This emphasizes the parallel nature of the CKY-parser — the parser is not at all incremental but rather operates on the whole string at once. If desired, we could have no axiom at all and add an antecedent-less inference rule instead.

$$\textbf{Shift} \qquad \frac{}{[i, A, i+1]}\, A \rightarrow a, a = w_i$$

The goal item of CKY is $[0, S, n]$, just like in our original parsing schema for bottom-up parsers. Just from the goal and the axioms, then, we can already infer essential aspects of CKY parsing.

Only the inference rules remain to be specified, of which the CKY parser has only two.

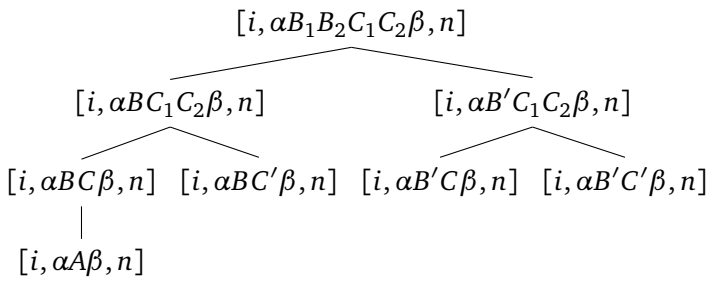



Figure 7.2: A standard bottom-up parse cannot efficiently reuse items and thus must take more steps

$$\textbf{Reduce} \qquad \frac{[i, B, j] \qquad [j, C, k]}{[i, A, k]} \; A \rightarrow BC \in R$$

It is instructive to contrast this version of the Reduce rule with the one used in a normal bottom-up parser. There we infer from a single item $[i, \alpha\gamma\beta, j]$ the validity of a new item $[i, \alpha N\beta, j]$. We may visualize this as adding some tree structure on top of a string of symbols. The CKY parser, on the other hand, combines two subtrees into one tree.

Nevertheless the same effect can be obtained in a bottom-up parser. Suppose that $B$ and $C$ in the pattern above are reduced from $[i, B_1, x][x, B_2, j]$ and $[j, C_1, y][y, C_2, k]$, respectively. In the CKY parser we can go from these four items to $[i, A, k]$ via three reduction steps. A breadth-first parser carries out a comparable reduction in the same number of steps.

| **parse item** | **inference rule** |
|---:|---|
| $[i, \alpha B_1 B_2 C_1 C_2 \beta, n]$ | already derived |
| $[i, \alpha B C_1 C_2 \beta, n]$ | reduce |
| $[i, \alpha B C \beta, n]$ | reduce |
| $[i, \alpha A \beta, n]$ | reduce |

What makes the CKY parser special thus cannot be its inference rules. Instead, it is the format of its parse items, which is ideally suited to memoization.

Suppose for the sake of argument that $B_1 B_2$ can also be reduced to $B'$ and $C_1 C_2$ to $C'$. In the CKY parser, that is not much of an issue. Two reductions for $B_1 B_2$ give us $B$ and $B'$, and two reductions for $C_1 C_2$ yield $C$ and $C'$. If only $BC$ can be reduced further, we are guaranteed to arrive at $A$ in 5 steps. The standard bottom-up parser takes longer. As can be seen in Fig. 7.2, it may require 7 steps to successfully reduce $[i, \alpha B_1 B_2 C_1 C_2 \beta, n]$ to $[i, \alpha A \beta, n]$ (depending on which reductions are prioritized by the control structure). This is purely due to the fact that reduction steps have to be repeated.

## 3.4 A Remark on Chomsky Normal Form

Throughout the entire discussion so far, we have implicitly assumed that rewrite rules are in *Chomsky Normal Form*.

---

*Definition 7.2 (Chomsky Normal Form).* A context-free grammar is in Chomsky Normal Form (CNF) iff every rewrite rule is in one of two forms:

- $A \rightarrow BC$, where $A, B, C \in N$, or
- $A \rightarrow a$, where $A \in N$ and $a \in T$.

---

As indicated by the term *normal form*, every CFG can be brought into CNF.

*Exercise 7.1.* Define an algorithm for transforming arbitrary CFGs into CNF. ⊙

*Exercise 7.2.* Another common normal form for CFGs is *Greibach Normal Form* (GNF), which only allows rewrite rules of the form $A \rightarrow a\alpha$, where $\alpha \in N^*$. Can you define an algorithm that converts any arbitrary CFG into GNF as long as it does not generate the empty string? *Hint:* You already know an algorithm that gets you halfway there. ⊙

It is often stated that the CKY-parser only works for grammars in CNF. This statement is somewhat misleading. The CKY algorithm, which combines the parsing schema from the previous section with a chart-based data structure, is indeed limited to CNF grammars. This is so because the chart does not provide a good way of dealing with rewrite rules that have more than two symbols on their right-hand side. The CKY parsing schema, on the other hand, can easily be generalized to arbitrary context-free grammars.

**Reduce** $$\frac{[i, B_1, j_1] \quad [j_1, B_2, j_2] \quad \cdots \quad [j_{n-1}, B_n, k]}{[i, A, k]} \; A \rightarrow B_1 B_2 \cdots B_n \in R$$

Any known implementation is still much faster with CNF grammars, but the claim that CKY parsing only works for those is misleading.

### 3.5 Formalizing the Data Structure

Agenda initialized as $\{[i, A, i+1] | A \rightarrow a, a = w_i\}$

## References and Further Reading

Nederhof, Mark-Jan. 1993. Generalized left-corner parsing. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, 305–314.

Tomita, Masaru. 1986. *Efficient parsing for natural language*. Norwell, MA: Kluwer Academic Publishers.

Tomita, Masaru. 1987. An efficient augmented context-free parsing algorithm. *Computational Linguistics* 31–46.

# Lecture 8

# Earley Parsing

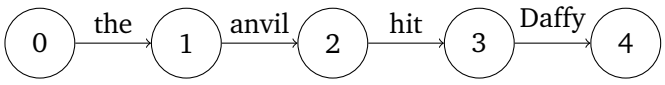
The CKY parser is a non-directional bottom-up parser with a chart, but chart parsing is in no way restricted to these parameters. In this chapter, we look at a directional chart parser with a top-down component: the Earley parser. The Earley parser combines a variety of techniques and insights we have encountered up to this point, and we will emphasize these connections by carefully working out the relation between Earley parsing and LC parsing. The Earley parser is also one of the fastest known algorithms for arbitrary CFGs, and in contrast to the CKY parser it does not require grammars to be in Chomsky Normal Form. In addition, Earley parsing has shown promise in some psycholinguistic modeling experiments (Hale 2001), which will be discussed further in Cha. 10.

## 1 Intuition

There are two intuitive perspectives of Earley parsing. We may either view it as a top-down parser where predictions are restricted by bottom-up reductions, or as a bottom-up parser where reductions are restricted by top-down predictions. In either case we are clearly dealing with a model that mixes top-down and bottom-up aspects, similar to the LC parser.

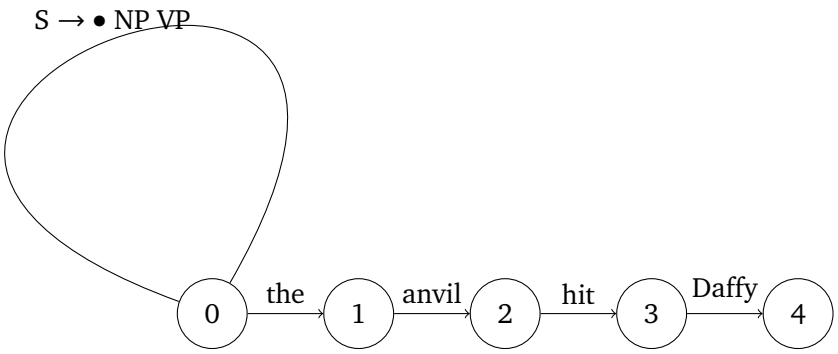
The basic idea is that the parser may freely make predictions like a top-down parser, while also using bottom-up information to quickly discard incompatible predictions before they have been fully explored. The parser moves through the input from left-to-right while trying to construct an arc that spans from the first position to the last. Arcs are just convenient abstractions of subtrees in this context. If the end position of one arcs is the start position of another arc, they can be combined into a larger arc, just like two subtrees can be combined into a single tree by attaching them below a new node. The strength of the Earley parser is how it avoids spending time on arcs that are bound to fail.

All of this is best explained through a concrete example. Suppose that we have our usual toy grammar and want to parse the input sentence *The anvil hit Daffy*, represented here as an indexed string.

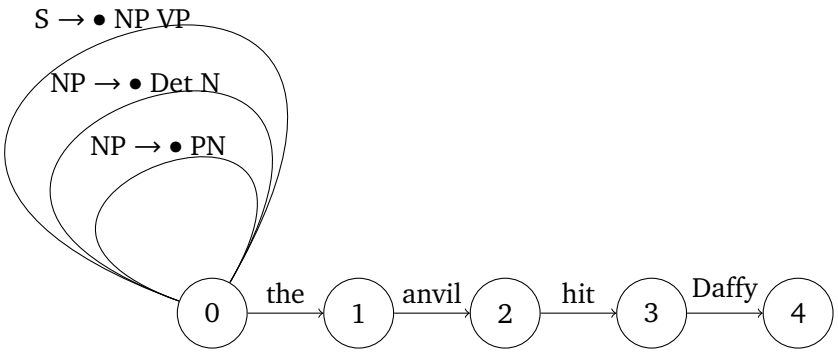



The parser will start out by conjecturing an arc from 0 to 0 that is labeled S → •NP VP. This is tantamount to the prediction that there will be an S-arc starting in position 0,

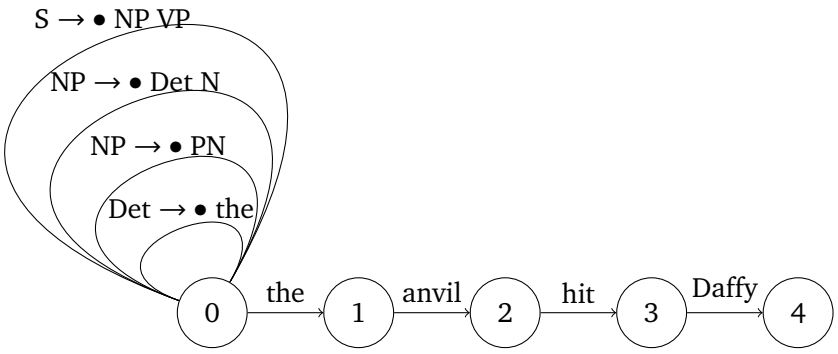
but so far we have not recognized an NP or a VP, so the confirmed part of the arc ends in position 0.



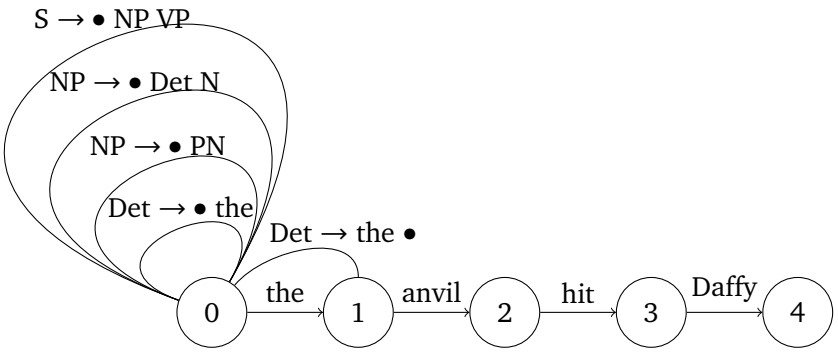

Since the S-arc can only be expanded in the presence of an NP, we also add arcs for the relevant NP-predictions.



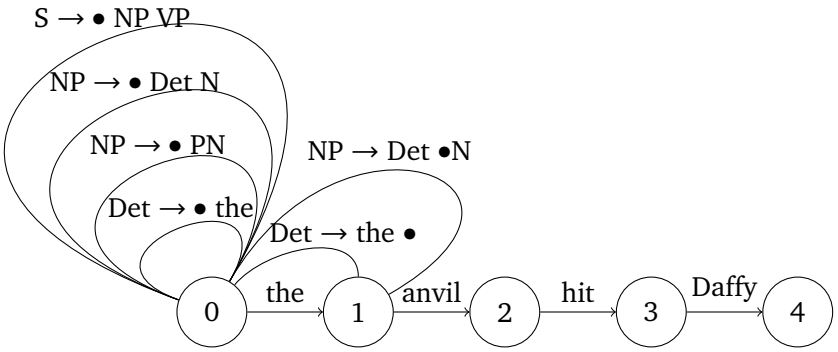

Following the same line of reasoning, we also have to add branches for each possible rewriting of Det and PN. We only add one Det-arc here to avoid cluttering the figure.



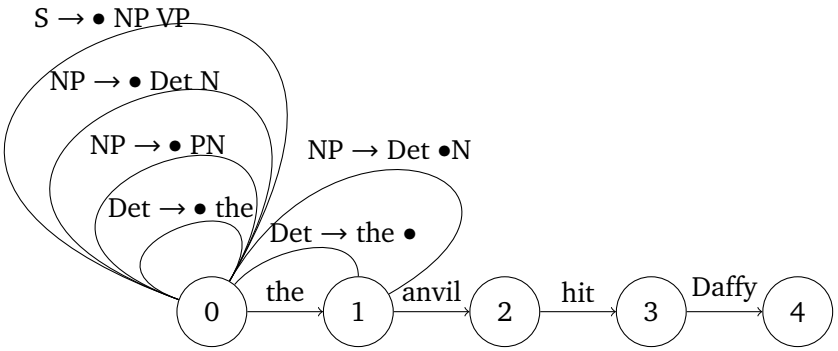

Now that we have an arc that can be expanded if we have the right terminal symbol, the parser reads in the word starting at position 0, which is *the*. This means that we have confirmed previously predicted information of the Det-arc in position 0, so that we also add a completed Det-arc from 0 to 1.




We can now combine this arc with the NP → •Det N arc from 0 to 0 to produce a new arc NP → Det •N from 0 to 1.




This arc, in turn is used to add arcs from 1 to 1 that are labeled with dotted rewrite rules for N, e.g. N → •anvil. We add these arcs because the NP-arc from 0 to 1 can only be completed if there is an N-arc starting at position 1.




*Exercise 8.1.* Continue the example parse until you have reached the S-arc. ⊙

*Exercise 8.2.* Use this arc-drawing system to parse the sentence *then anvil hit the duck on the head* with the expanded toy grammar we used for the CKY parser. Does this format offer a way of representing the structural ambiguity? ⊙

# 2 Formal Specification

## 2.1 Parsing Schema

In its arc-based representation the Earley parser may look rather unusual to you, but the item-based parsing schema should strike you as remarkably familiar. Items are of the form $[i, N \rightarrow \alpha \bullet \beta, j]$ and indicate that an $N$-constituent starts in position $i$ and has been recognized up to position $j$. As our axiom we pick $[0, \mathrm{S}' \rightarrow \bullet \mathrm{S}, 0]$. This is just a convenient shorthand so that we do not have to posit an axiom for every rewrite rule for S in our grammar. Given this convention, then goal item is $[0, \mathrm{S}' \rightarrow \mathrm{S} \bullet, n]$, of course.

The inference rules are variants of rules that we already encountered with the shift-reduce parser and the LC parser. Scan turns a predicted terminal symbol into recognized information (so it is the counterpart of the LC parser's shift rule rather than the scan rule, which is used to cancel out predicted and confirmed information).

$$\textbf{Scan} \qquad \frac{[i, A \rightarrow \alpha \bullet a\beta, j]}{[i, A \rightarrow \alpha a \bullet \beta, j+1]} \; a = w_j$$

Predict is similar to standard top-down prediction. But just like the CKY algorithm breaks large items into smaller ones that can be easily stored in a chart and used in several inference steps, the Earley parser predicts much smaller items that explicitly represent conjectured subtrees.

$$\textbf{Predict} \qquad \frac{[i, A \rightarrow \alpha \bullet B\beta, j]}{[j, B \rightarrow \bullet \gamma, j]} \; B \rightarrow \gamma \in R$$

The complete rule, finally, mirrors the behavior of the LC parser's complete rule in that it turns confirmed predictions into recognized material. The only difference is that now the combination of two parse items is needed to license this inference.

$$\textbf{Complete} \qquad \frac{[i, A \rightarrow \alpha \bullet B\beta, j] \qquad [j, B \rightarrow \gamma \bullet, k]}{[i, A \rightarrow \alpha B \bullet \beta, k]}$$

Exactly like the CKY parser, the Earley parser is mostly used as a fully parallel chart parser that explores all possible parses at the same. The Earley parsing schema, however, is independent of such considerations and can also be applied in a purely serial fashion. In this case the Earley parser behaves similar to a recursive-descent parser.

**Example 8.1 Parse Table of an Earley Parse**

Consider once more the sentence *the anvil hit Daffy*, parse with our usual toy grammar. Then the shortest possible sequential Earley parse yields a parse table that closely resembles the one of the recursive-descent parser.

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

The direct comparison of parse tables in the example above highlights that we may think of the recursive-descent parser as a particular instantiation of the Earley parser: a single recursive-descent scan represents an Earley scan followed by one or more completion steps. In general, a parser that takes fewer steps is more efficient than one that takes more steps. So if we are interested in designing a serial parser with backtracking that does not reuse parse items, recursive-descent is a better choice than Earley. In other words, recursive-descent is a smart modification of the Earley parsing schema for the purpose of serial parsing.

This view also highlights the downside of recursive-descent in comparison to Earley: the latter can easily reuse parse items and thus is a much more natural candidate whenever the parser is allowed to memorize and reuse parse items. This makes Earley a much better choice for chart parsing and industrial applications.

### 2.2 Adding Chart Parsing Techniques

agenda
    chart

## 3 Left Recursion is Unproblematic

When we looked at the recursive descent parser, we saw that left recursion can force it to enter an infinite loop of top-down predictions. The shift-reduce and LC parsers do

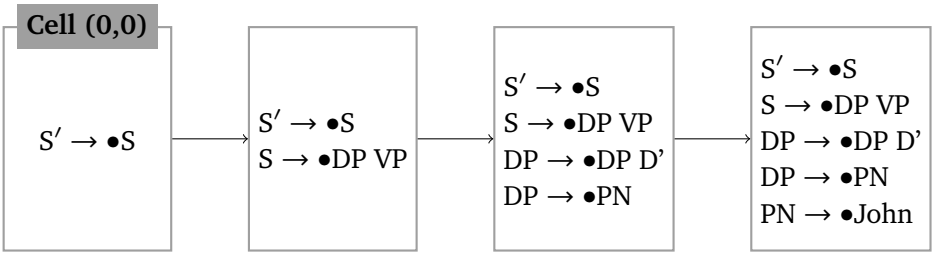
not run into this issue since they use bottom-up information. The same is true of the Earley parser, which uses bottom-up information to restrict top-down predictions. It is interesting to see, though, how exactly the Earley parser avoids infinite loops.

**Example 8.2 Earley Parse of Nested Possessive DPs**

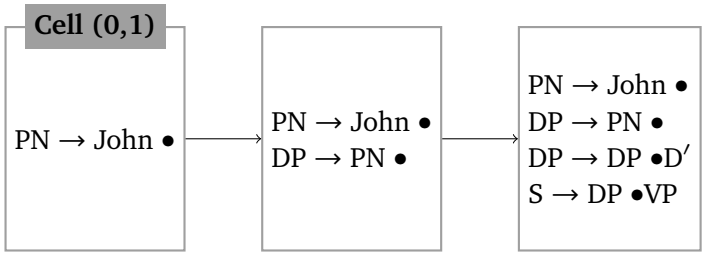
The sentence *John's father's car's exhaust pipe disappeared* is to be analyzed with the following grammar:

| | | | | | | |
|---|---|---|---|---|---|---|
| S | → | DP VP | Poss | → | 's | |
| DP | → | DP D′ \| PN | N | → | father \| car \| exhaust pipe | |
| D′ | → | Poss NP | PN | → | John | |
| NP | → | N | V | → | disappeared | |
| VP | → | V | | | | |

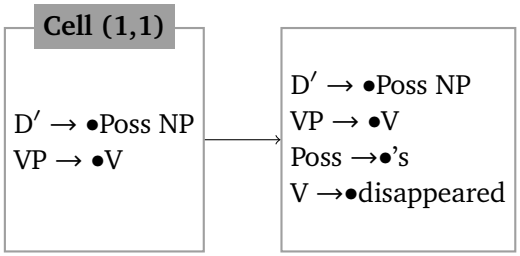
Since the chart for this sentence is rather large, we look at individual cells instead. Cell $(0,0)$ starts out with the axiom S′ → •S, which this triggers a prediction of S →•DP VP. This item, in turn, yields two predictions: DP →•DP D′ and DP →•PN. The latter also adds PN →•John. The construction process of the cell is shown below.



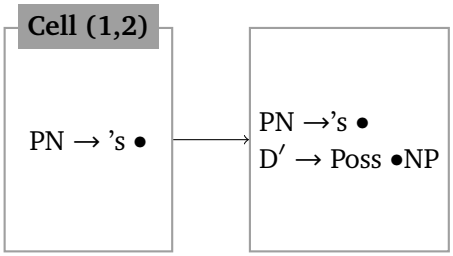

At this point the scan rule reads in *John* and thus adds PN → John •to cell $(0,1)$. This results in a number of complete step adding new items to the very same cell.



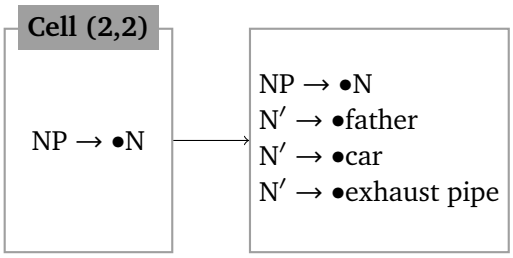

The items in cell $(0,1)$ now trigger prediction steps that add items to cell $(1,1)$ for VP and D′. This subsequently yields predictions for V and Poss, too.



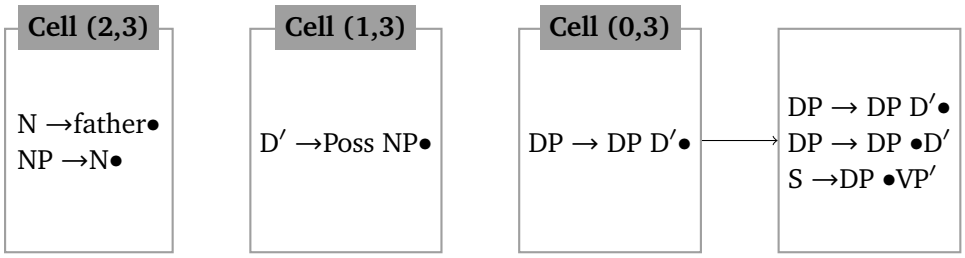
At this point scan can once again take place, reading in *'s*. As a result, the completed rule Poss →'s•is added to cell (1, 2). Using this item, we can also add a partially completed D′ item.




The NP-prediction is added to cell (2, 2), and so are the N-predictions triggered by the item.




Scan now adds N →father•to cell (2, 3), which is also used to complete the NP item. As a result, we can add a completed D′ item to cell (1, 3). The completed D′ allows to put completed DPs into cell (0, 3), from which we infer a partially completed S item and add it to the same cell.




Two crucial steps have taken place in the parse so far. First, the parser did not end up in a loop of left predictions at the very beginning. When the parser added DP → •DP D′ to cell (0, 0), it did not try to enter another instance of the item even though the corresponding predict rule is triggered by the addition of the item. The interplay of agenda and chart is to thank for that:

1. When S → •DP VP is at the top of the agenda, it triggers the prediction of DP → •DP D′ and DP → •PN.
2. Both items are added to the chart, and are also put in the agenda to indicate that no inference rules have been applied to them yet.
3. In the next step, the parser computes the predictions of DP →•DP D′ and checks if some of them are not in the chart yet. This is not the case, and thus the chart stays the same.

4. DP → •DP D′ is removed from the agenda and will not be operated on again.

The way items are moved in and out of the agenda and stored in the chart thus prevents an infinite loop of predictions.

The other interesting aspect is the addition of DP → DP •D′ to cell (0,3). The item is obtained from DP →DP D′• in cell (0,3) and DP → •DP D' in cell (0,0). This may seem surprising since the latter was already used in the assembly of *John's father* into a DP. But strictly speaking it only represents the prediction of a DP that starts at position 0. Since the nested possessives construction has multiple DPs with this start position, the Earley parser is allowed to recycle the prediction accordingly.

Proceeding in the very same fashion of recycling the DP prediction, the Earley parser can infer the structure for *John's father's car's exhaust pipe disappeared* without major problems.

*Exercise 8.3.* Draw the full chart for the Earley parse of *John's father's car's exhaust pipe disappeared*. ⊙

## 4 The GHR Algorithm

Several variants of Earley parsing have been proposed in the literature. One of the fastest yet also intuitive ones is GHR parsing, originally proposed by Graham, Harrison and Ruzzo in 1980.

GHR parsing relies on analyzing the grammar in advance and identifying those non-terminals from which one can generate the empty string. In linguistic parlance, we have to identify those labels that may be the root of a subtree that consists only of unpronounced material. There is little point in exploring such subtrees since the input can neither confirm nor disconfirm their existence. Consequently, a • to the left of such a node is allowed to move across it. This strategy requires nothing more than some minor modifications to the Earley parsing schema.

**Scan** $$\frac{[i, A \rightarrow \alpha \bullet a\beta\gamma, j]}{[i, A \rightarrow \alpha a\beta \bullet \gamma, j+1]} \; a = w_i, \beta \rightarrow^* \varepsilon$$

**Predict** $$\frac{[i, A \rightarrow \alpha \bullet B\beta, j]}{[j, C \rightarrow \alpha' \bullet \beta', j]} \; B \rightarrow^* C\gamma, \alpha' \rightarrow^* \varepsilon$$

**Complete1** $$\frac{[i, A \rightarrow \alpha \bullet B\beta\gamma, j] \qquad [j, B \rightarrow \delta \bullet, k]}{[i, A \rightarrow \alpha B\beta \bullet \gamma, k]} \; \beta \rightarrow^* \varepsilon$$

**Complete2** $$\frac{[i, A \rightarrow \alpha \bullet B\beta\gamma, j] \qquad [j, C \rightarrow \delta \bullet, k]}{[i, A \rightarrow \alpha B\beta \bullet \gamma, k]} \; B \rightarrow^* C, \beta \rightarrow^* \varepsilon$$

Side conditions of the form $\alpha \rightarrow^* \beta$ express that either $\alpha = \beta$ or $\beta$ can be obtained from $\alpha$ in one or more rewrite steps. For any given CFG, whether $B \rightarrow^* C\gamma$ and $B \rightarrow^* \varepsilon$ can be determined in advance. This can then be precompiled into the parser, e.g. in form of a boolean matrix. This matrix contains a row for each non-terminal symbol as well as a column for each non-terminal symbol and the empty string. A cell $(B, C)$ has value 1 iff $B \rightarrow^* C\gamma$, and similarly $(B, \varepsilon)$ is 1 iff $B \rightarrow \varepsilon$ (the latter is just a special case of the former). This strategy can be extended to sequences of non-terminals so that even $\alpha' \rightarrow^* \varepsilon$ in the predict rule can be verified in a single step.

*Exercise 8.4.* Every inference step in the Earley parse schema is still a valid inference step in the GHR parse schema. Explain why! ⊙

GHR parsing is an obvious variant of Earley parsing. That does not make it less of a parser, its speed-up over standard Earley parsing is a notable achievement. It also shows that parsing schemata highlight commonalities between distinct parsing algorithms. The relations between parsers that can be established this way extend far beyond simple cases like Earley and GHR. In the next chapter, we will establish principled connections between Earley, CKY, and LC parsing.

## References and Further Reading

Graham, Susan L., Michael Harrison, and Walter L. Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems* 2:415–462.

Hale, John. 2001. A probabilistic earley parser as a psycholinguistic model. In *Proceedings of the Second Meeting of the North American Chapter of the Association for Computational Linguistics*.

# Lecture 9

# Schema Refinement and Filtering

In this chapter we will look at several techniques for converting one parsing schema into another: item refinement, step refinement, static filtering, dynamic filtering, and step contraction. The latter we already encountered in Chap. 6, where we observed that I) the LC reduce rule is a step contraction of predict followed by complete, and II) the scan rule of recursive descent parsing schema can be viewed as a step contraction of LC shift followed by LC scan. Step contraction and the other mechanisms allow us to establish surprising relations between seemingly incomparable parsing schema. This is not only of theoretical interest. Since many of the techniques can be viewed as abstractions of algorithmic speed-up techniques (e.g. precompilation, run-time optimization), they also explain why some parsers are faster than others.

## 1 A Base Case: Bottom-Up Earley

As a useful base case that will simplify comparisons later on, we define a simplified variant of the Earley parser without the predict rule. The only valid inference rules thus are scan and complete.

$$\textbf{Scan} \qquad \frac{[i, A \to \alpha \bullet a\beta, j]}{[i, A \to \alpha a \bullet \beta, j+1]}\ a = w_j$$

$$\textbf{Complete} \qquad \frac{[i, A \to \alpha \bullet B\beta, j] \qquad [j, B \to \gamma \bullet, k]}{[i, A \to \alpha B \bullet \beta, k]}$$

The goal item is still $[0, S' \to S\bullet, n]$. The axiom, on the other hand, has to be changed since the absence of a predict rule means that all items must be inferred bottom-up. We do so by allowing every prediction to take place at every position in the input. That is to say, for every position $i$ and rewrite rule $A \to \alpha$ of the grammar, there is a corresponding axiom $[i, A \to \bullet\alpha, i]$.

*Exercise 9.1.* Construct the chart for a simplified Earley parse of *the anvil hit the duck on the head* using the extended toy grammar from Chap. 7. Feel free to omit some of the axioms, which would require a lot of ink to be spilled. ⊙

Before moving on, we can already note two particular insights. First, the simplified Earley parser is a pure bottom-up parser, similar to shift-reduce or CKY. Second,

standard Earley parsing constructs fewer arcs than simplified Earley parsing. So the former is the result of enriching the latter with top-down filtering on predicted items. The relation between the two parsing schemata is similar to the relation that holds between the standard LC parsing schema and the extended variant with top-down filtering (cf. Sec. 2.2 of Cha. 6). This also highlights that even though the Earley parser intuitively feels like a top-down parser, it is best viewed on a formal level as a bottom-up parser (which also explains immediately why left recursion is unproblematic). To further emphasize this point, we will refer to the simplified Earley schema as bottom-up Earley (buE).

# 2 Refinement

## 2.1 Defining Item and Step Refinements

We now introduce our first two tools for relating parsers: item refinement and step refinement. Both are fairly easy to understand on an intuitive level. A parsing schema $P$ is an *item refinement* of schema $Q$ if the items of $Q$ are more fine-grained. Technically, this means that we can define a total surjective map from $Q$'s parse items to $P$'s parse items such that the image of $Q$'s parses under this map is exactly the parses of $P$.

**total** A function $f$ from $A$ to $B$ is *total* iff $f(a)$ is defined for every $a \in A$.

**surjective** A function $f$ from $A$ to $B$ is *surjective* iff for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. In this case $f$ is also said to be *onto*.

Step refinement applies the same concept to sequences of inference rules. A parsing schema $P$ is a *step refinement* of schema $Q$ iff for every inference rule of $Q$ there is an equivalent sequence of one or more inference steps in $P$. If this still seems awfully abstract enough, don't worry, we will work through a concrete example next.

## 2.2 Item Refinement of CKY

Recall that CKY uses items of the form $[i,A,j]$, which are operated on by two rules:

$$\textbf{Shift} \qquad \frac{}{[i,A,i+1]}\, A \rightarrow a, a = w_i$$

$$\textbf{Reduce} \qquad \frac{[i,B,j] \qquad [j,C,k]}{[i,A,k]}\, A \rightarrow BC \in R$$

We will now refine CKY into a new parsing schema CKY$^{\text{E}}$ that uses Earley-style productions instead of just non-terminals. So items are of the general form $[i,A \rightarrow \alpha, j]$, assuming that $A \rightarrow \alpha$ is a valid rewrite rule of a context-free grammar in Chomsky Normal Form. It is a trivial matter to adapt the inference rules accordingly:

$$\textbf{Shift} \qquad \frac{}{[i,A \rightarrow a, i+1]}\, A \rightarrow a, a = w_i$$

$$\textbf{Reduce} \qquad \frac{[i,B \rightarrow \beta, j] \qquad [j,C \rightarrow \gamma, k]}{[i,A \rightarrow BC, k]}\, A \rightarrow BC \in R$$

The relation between parsing items is equally obvious. Every parse item $[i,A \rightarrow \alpha, j]$ of CKY$^{E}$ is mapped to $[i,A,j]$ in CKY. Whatever parse of CKY$^{E}$ this function is applied to, it yields a parse of CKY. Furthermore, every parse of CKY is the image of some CKY$^{E}$ parse under this function. These three facts jointly establish that CKY$_{E}$ is an item refinement of CKY.

## 2.3 Bottom-Up Earley as a Refinement of Generalized CKY

Considered in isolation, CKY$^{E}$ is rather unremarkable as it only adds unnecessary redundancy to the standard CKY schema. But in the grand scheme of things, it is a useful step towards relating CKY and Earley parsing. Towards this end, we now define another schema ECKY and show that it is a step refinement of CKY$^{E}$. ECKY modifies item format as CKY$^{E}$ to proper Earley-style dotted productions, i.e. $[i,A \rightarrow \alpha \bullet \beta, i]$. The inference rules are also modified.

$$\textbf{Scan} \qquad \frac{[i,A \rightarrow \alpha \bullet a\beta, j]}{[i,A \rightarrow \alpha a \bullet \beta, j+1]} \; a = w_j$$

$$\textbf{Complete} \qquad \frac{[i,A \rightarrow \alpha \bullet B\beta, j] \qquad [j,B \rightarrow \gamma\bullet, k]}{[i,A \rightarrow \alpha B \bullet \beta, k]}$$

In addition, we posit an axiom $[i,A \rightarrow \bullet\alpha, i]$ for every input position $i$ and rewrite rule $A \rightarrow \alpha$ of the grammar. You might point out that ECKY is identical to buE, but this is only partially correct. Since ECKY is only defined for grammars in Chomsky Normal Form, it is a special case of buE, which works with arbitrary grammars. The difference between the two is obscured by our use of variables like $\alpha$, $\beta$ and $\gamma$. Every licit instantiation of those variables for ECKY is also valid for buE, but not the other way round. Still, the point stands that ECKY is essentially buE restricted to grammars in Chomsky Normal Form.

In order to show that ECKY is a step refinement of CKY$^{E}$, we exhibit equivalent sequences of inference steps for CKY$^{E}$'s shift and reduce rules. This presupposes that we already know how to relate parse items of CKY$^{E}$ to ECKY. Fortunately this is easy in this case: every item $[i,A \rightarrow \alpha, j]$ of CKY$_{E}$ is equated with $[i,A \rightarrow \alpha\bullet, j]$ of ECKY. Now suppose that we infer $[i,A \rightarrow a, i+1]$ via the shift rule of CKY$^{E}$. ECKY instead uses the axiom $[i,A \rightarrow \bullet a, i]$ and the scan rule to infer $[i,A \rightarrow a\bullet, i+1]$, which is the ECKY correspondent of CKY$^{E}$'s $[i,A \rightarrow a, i+1]$. Where CKY$^{E}$ uses its reduce rule to infer $[i,A \rightarrow BC, k]$ from $[i,B \rightarrow \beta, j]$ and $[j,C \rightarrow \gamma, k]$, ECKY instead invokes two complete steps:

$$\frac{\dfrac{[i,A \rightarrow \bullet BC, i] \qquad [i,B \rightarrow B\bullet, j]}{[i,A \rightarrow B \bullet C, j]} \qquad [j,C \rightarrow \gamma\bullet, k]}{[i,A \rightarrow BC\bullet, j]}$$

As $[i,A \rightarrow BC\bullet, k]$ corresponds to $[i,A \rightarrow BC, k]$, ECKY is indeed a step refinement of CKY$^{E}$.

At this point, we know that ECKY is a step refinement of CKY$^{E}$, which in turn is an item refinement of CKY. For the sake of simplicity, any sequence of refinement steps is called a *refinement*. So ECKY is a *refinement* of CKY.

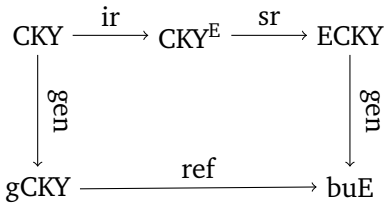
This result may still strike you as fairly artificial since neither ECKY nor CKY$^{E}$ have been proposed independently in the parsing literature. The only thing we have

accomplished is to relate parsing schema that we made up little more than a page ago, seemingly on a whim. But remember that ECKY is a special case of buE. The two behave exactly the same except that ECKY is only defined for grammars in Chomsky Normal Form. But of course the steps we took to refine CKY into ECKY can also be repeated with generalized CKY (gCKY).

*Exercise 9.2.* Guess what? You get to do just that. ⊙

The kind of generalized ECKY that is obtained this way is exactly buE. Consequently, buE is a refinement of gCKY. Or the other way round, gCKY is a more concise variant of buE that simplifies items and compacts certain inference sequences into a single inference step.

Our findings so far can be summarized in a simple graph. Here parsing schema $P$ is considered a generalization of $Q$ iff $P$ is defined for a superset of the grammars $Q$ is defined for and the two behave exactly the same over their shared grammars. The relations item refinement, step refinement, refinement and generalization are abbreviated as ir, sr, ref, and gen, respectively.




# 3 Filtering

## 3.1 Defining Static and Dynamic Filtering

Whereas refinement "grows" a parsing schema by adding new items and/or decomposing single inference steps into a sequence thereof, filtering is all about making parsing schemata more compact. Filters are in particular meant to reduce the amount of work that needs to be done during the construction of a parse. The simplest option is a *static filter*, which simply eliminates certain items or inference rules. A parsing schema $P$ is a static filtering of $Q$ ($Q \xrightarrow{sf} P$) iff I) all parse items of $P$ are parse items of $Q$, and II) every inference rule of $P$ is an inference rule of $Q$.

*Dynamic filters* are of greater interest for our purposes. Whereas static filters discard parts of the parsing schema to simplify the construction of parses, dynamic filters minimize workload by putting additional restrictions on the applicability of inference rules. Technically, parsing schema $P$ is a dynamic filtering of schema $Q$ ($Q \xrightarrow{df} P$) iff I) all items of $P$ are items of $Q$, and II) all valid inference steps of $P$ are valid inference steps of $Q$. Notice that this definition makes static filters a special case of dynamic filters. Whereas static filters are more blunt and remove entire inference rules, dynamic filters can retain all inference rules but restrict their applicability.

## 3.2 Earley is a Dynamic Filtering of Bottom-Up Earley

We have already encountered an instance of dynamic filtering: in Chap. 6, we added top-down filtering to the standard left-corner parser. This mechanism preserved all

parse items and inference rules (so it is not a static filtering), but it limited the reduce rule so that only valid left corners could be inferred. For another example, remember that buE was obtained from Earley by removing exactly the kind of top-down filtering that we added to the left-corner parser. It stands to reason, then, that buE is a dynamic filtering of Earley — that is indeed the case.

Note first that buE and Earley have exactly the same set of valid items. They also have exactly the same scan and complete rules. The main differences thus lie in the predict rule and the choice of axioms. Given the definition of dynamic filtering, it suffices to show that no licit inference step of Earley is illict in buE. So suppose we predict $[j, B \rightarrow \bullet\gamma, j]$ from $[i, A \rightarrow \alpha \bullet B\beta, j]$. Then $[j, B \rightarrow \bullet\gamma, j]$ is a valid inference in buE because it is one of its axioms. The same holds for the Earley parser's single axiom, $[0, S' \rightarrow \bullet S, 0]$.

*Exercise 9.3.* Show that the LC parsing schema with top-down filtering is a dynamic filtering of the standard LC schema. ⊙

### 3.3 Defining Step Contraction

The most powerful type of filtering is *step contraction*, which — somewhat surprisingly — is just the inverse of step refinement. That is to say, parsing schema $P$ is a step contraction of schema $Q$ ($Q \xrightarrow{sc} P$) iff $Q$ is a step refinement of $P$. Why do we need both step contraction and step refinement if one is just the inverse of the other? Because both step contraction and step refinement have their uses. In particular, both can lead to significant speed-ups in the parser (the flip side being that both can also make the parser slower).

It is also interesting that every dynamic filter is a step contraction, and every inverse step contraction (i.e. a refinement) is a static filter. This means we have the following subsumption relation: ref $\subseteq$ sf $\subseteq$ df $\subseteq$ sc.

*Exercise 9.4.* Explain why ref $\subseteq$ sf and df $\subseteq$ sc hold. ⊙

### 3.4 Left Corner Parsing is a Step Contraction of Earley

For the sake of convenience, we redefine the parsing schema for the left-corner parser with top-down filtering to bring it more in line with the Earley schema. Instead of the complicated LC parse items, we follow the format $[i, A \rightarrow \alpha \bullet \beta, j]$ of the Earley parser. We also adopt the axiom of the Earley parser: $[0, S' \rightarrow \bullet S, 0]$.

The rewrite rules also undergo some major modifications. Remember that the reduce rule can be emulated by predict followed by complete, so we drop it. This leaves us with shift, scan, predict and complete. We recombine shift and scan into the more standard scan rule as it is familiar from the recursive-descent and Earley parsers. We then split predict into two rules depending on whether it is triggered by a terminal symbol or a non-terminal. Complete is just the standard complete rule of the Earley parser. So the full set of rewrite rules looks as follows:

**Scan**
$$\frac{[i, A \rightarrow \alpha \bullet a\beta, j]}{[i, A \rightarrow \alpha a \bullet \beta, j+1]}\ w_j = a$$

**Predict Non-Terminal**
$$\frac{[i, C \rightarrow \gamma \bullet E\delta, j] \qquad [j, A \rightarrow \alpha\bullet, k]}{[j, B \rightarrow A \bullet \beta, k]}\ B \in \mathrm{lc}(E)$$

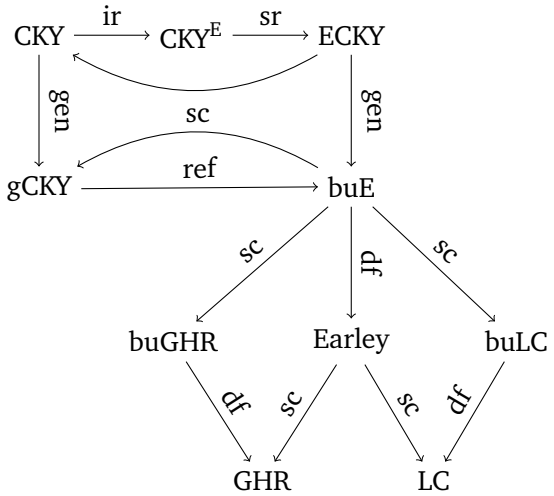
**Predict Terminal** $$\frac{[i, C \to \gamma \bullet E\delta, j]}{[j, B \to a \bullet \beta, k]}\ a = w_j, B \in \mathrm{lc}(E)$$

**Complete** $$\frac{[i, A \to \alpha \bullet B\beta, j] \qquad [j, B \to \gamma, k]}{[i, A \to \alpha B \bullet \beta, k]}$$

*Exercise 9.5.* Verify for yourself that this parser still builds trees in the same fashion as our original left-corner parser with top-down filtering. You might want to start by constructing the parse for our usual example sentence *The anvil hit Daffy*, and possibly *John's father's car's exhaust pipe disappeared.* ⊙

With these changes in the parsing schema it is very easy to recognize the LC parser with top-down filtering as a step contraction of the Earley parser. To prove it, we have to show that the Earley parser is a refinement of the LC parser. Every LC parse item is also an Earley item, so Earley is an item refinement of LC. In addition, Earley is also a step refinement. The scan and complete rule are identical between the two. As we establish next, Predict Non-Terminal and Predict-Terminal correspond to sequences of Earley predict and completion steps.

Consider the LC parser's Predict Non-Terminal rule. It allows the LC parser to infer $[j, B \to A \bullet \beta, k]$ provided $[i, C \to \gamma \bullet E\delta, j]$ and $[j, A \to \alpha\bullet, k]$ are valid. Now suppose that we are given $[i, C \to \gamma \bullet E\delta, j]$ in the Earley parser. Since $B$ must be a left-corner of $E$, some finite sequence of predict steps infers $[j, B \to \bullet A\beta, j]$ from $[i, C \to \gamma \bullet E\delta, j]$. From this we infer $[j, A \to \bullet\alpha, j]$, which is eventually completed as $[j, A \to \alpha\bullet, k]$. Another complete step then produces $[j, B \to A \bullet \beta, k]$. A similar argument also works for Predict Terminal.




*Exercise 9.6.* Show that buE $\xrightarrow{sc}$ buGHR and buGHR $\xrightarrow{df}$ GHR hold. ⊙

*Exercise 9.7.* The diagram also shows relations for the GHR parsing schema and a bottom-up version without top-down filtering. Give a definition of buGHR, then prove the relevant relations. ⊙

*Exercise 9.8.* How could the generalized left-corner parser — and by extension the recursive-descent and shift-reduce parsers — be fit into the diagram? ⊙

## 4 Psycholinguistic Connections

I frankly don't know. That would be an interesting issue to explore.

## Lecture 10

# Parsing With Probabilities

Lecture 11

# Generalizing Parsers via Monoids and Semirings

# Lecture 12

# Moving Beyond Context-Free Grammars

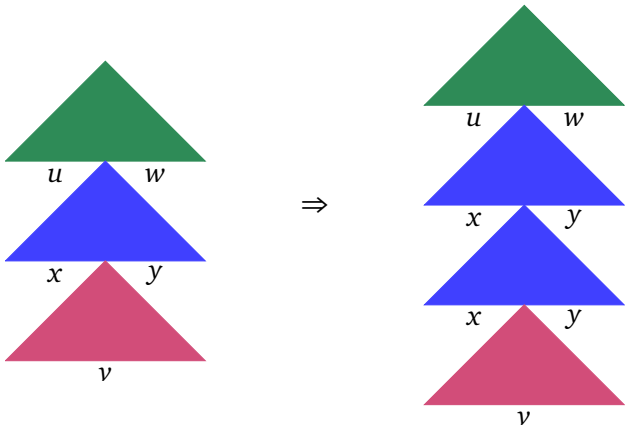
## 1 Not All Natural Languages are Context-Free

### 1.1 Context-Free Pumping Lemma

**Theorem 12.1 (CFL Pumping Lemma).** If $L$ is a context-free string language, then there is some constant $k \geq 0$ such that

- $uxvyw \in L$, and
- $|uxvyw| \geq k$

jointly imply $ux^n vy^n w \in L$ for all $n \geq 1$. □




**Lemma 12.2.** None of the following languages are context-free:

- $a_1^n a_2^n \cdots a_i^n$, for any fixed choice of $i \geq 3$.
- $\{ww \mid w \in \Sigma^*, |\Sigma| \geq 2\}$,
- $a^{2^n}$ □

## 1.2 Mildly Context-Sensitive Languages

*Definition 12.3 (Mild Context-Sensitivity).* A class of string languages is *mildly context-sensitive* iff

1. it properly includes the class of context-free languages,
2. each language can be generated by a grammar that allows for parsing in polynomial time,
3. each language is of constant growth,
4. each language can be generated by a grammar with only a bounded number of crossing dependencies.

*Exercise 12.1.* Show that $a^n b^n c^n$ is of constant growth, whereas $a^{2^n}$ is not. ⊙

This definition is not particularly elegant as it puts restrictions on the languages as well as the grammars that generate them. A more unified definition in terms of just one of the two would be more appealing. In addition, the clause about crossing dependencies is very vague. Fortunately it can be made more precise via the concept of *semilinearity*.

## 1.3 Semilinear Languages

In order to define semilinearity, we first need to introduce the *Parikh map*. Intuitively, the Parikh map counts for each symbol of the alphabet how often it occurs in a given string. This is modeled mathematically as a function that takes a string as input and returns a vector of integers — the *Parikh vector* — such that the $i$th integer is the number of occurrences of the $i$th symbol of the alphabet (so we have to posit some arbitrary linear order for our alphabet symbols).

*Definition 12.4 (Vector addition).* Let $\vec{u} := \langle u_1, \ldots, u_k \rangle$ and $\vec{v} := \langle v_1, \ldots, v_k \rangle$ be vectors in $\mathbb{N}^k$. Then their sum $\vec{u} + \vec{v}$ is the vector $\langle u_1 + v_1, u_2 + v_2, \ldots, u_k + v_k \rangle$.

*Definition 12.5 (Parikh Map).* The *Parikh vector* of a string is the function $p : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ such that

$$p(a_1 a_2 \cdots a_n) := \begin{cases} 0^{i-1} \cdot \langle 1 \rangle \cdot 0^{k-i} & \text{if } n = 1 \text{ and } a_1 \text{ is the } i\text{-th symbol of } \Sigma, \\ p(a_1) + p(a_2 \cdots a_n) & \text{otherwise} \end{cases}$$

The *Parikh map* $\hat{p}$ associates a language $L$ with the set $\{p(w) \mid w \in L\}$. Two languages $L$ and $L'$ are *letter equivalent* iff $\hat{p}(L) = \hat{p}(L')$.

**Example 12.1 Parikh vectors and maps**

Consider the string language $\{a, b\}^*$, which contains all possible strings over $a$ and $b$. The Parikh map computes the following Parikh vectors for this language.

| String | Parikh Vector |
|---|---|
| $\varepsilon$ | $\langle 0,0\rangle$ |
| $a$ | $\langle 1,0\rangle$ |
| $b$ | $\langle 0,1\rangle$ |
| $aa$ | $\langle 2,0\rangle$ |
| $ab$ | $\langle 1,1\rangle$ |
| $ba$ | $\langle 1,1\rangle$ |
| $bb$ | $\langle 0,2\rangle$ |
| $\vdots$ | $\vdots$ |

The set of all Parikh vectors is $\{\langle m,n\rangle \mid m,n \in \mathbb{N}\}$, which is identical to $\mathbb{N}\times\mathbb{N}$. In a certain sense, then, we can think of $\{a,b\}^*$ as the set of all vectors in the first quadrant of the Cartesian plane. This is a simplification, though, since $\{a,b\}^*$ contains many strings that map to the same Parikh vector because the Parikh map completely ignores linear order.

*Exercise 12.2.* Define a string language that is distinct from $\{a,b\}^*$ but letter-equivalent to it. ⊙

*Exercise 12.3.* Define a string language as in the previous exercise, with the additional requirement that no proper subset of that language is letter equivalent to $\{a,b\}^*$. ⊙

The set $\mathbb{N}\times\mathbb{N}$ is obviously infinite, but nonetheless it can be generated from a finite number of vectors. Any vector of that set is a directed arrow in the first quadrant of that Cartesian plane. Wherever that vector points, we can get to that point by only moving up and to the right. For example, the location pointed to by the vector $(2,1)$ can be reached by starting at $(0,0)$ and taking two steps to the right, immediately followed by one step up. This can be written as $(2,1)=(0,0)+(1,0)+(1,0)+(0,1)=(0,0)+2\cdot(1,0)+1\cdot(0,1)$. Every vector of $\mathbb{N}\times\mathbb{N}$ can be described in this fashion, we just have to change the multipliers accordingly. Consequently, $\mathbb{N}\times\mathbb{N}$ is equivalent to the set $\{(0,0)+t_1\cdot(1,0)+t_2\cdot(0,1) \mid t_1,t_2\in\mathbb{N}\}$. Sets of this form are called *linear*.

---

*Definition 12.6 (Linear Set).* A set $S\subseteq\mathbb{N}^k$ is *linear* iff it is of the form $\{u_0+t_1u_1+\cdots+t_nu_n | t_1,\ldots,t_n\in\mathbb{N}\}$ for some fixed base vectors $u_0,u_1,\ldots,u_n$, $n\geq 0$.

---

*Exercise 12.4.* Show that the Parikh map yields a linear set for $a^nb^nc^n$, $n\geq 1$. ⊙

Many interesting languages are not linear under the Parikh map. However, they are still semilinear

---

*Definition 12.7 (Semilinearity).* A set is *semilinear* iff it is the union of a finite number of linear sets. A language is semilinear iff its image under the Parikh map is semilinear.

---

### 1.4 Semilinearity and Constant Growth

**Theorem 12.8.** Every semilinear set is of constant-growth. □

**Corollary 12.9.** $a^{2^n}$ is not semilinear. □

*Exercise 12.5.* Give an example of a language that is of constant-growth but not semilinear. ⊙

### 1.5 Semilinearity = Regular Backbone + String Permutation

**Theorem 12.10 (Parikh's Theorem).** For every semilinear language $L$ there is a regular language $R$ such that $\hat{p}(L) = \hat{p}(R)$. □

*Exercise 12.6.* The inverse of Parikh's Theorem also holds: for every regular language $R$ there is a semilinear language $L$ such that $\hat{p}(L) = \hat{p}(R)$. Explain why this is (trivially) the case. ⊙

## 2 Minimalist Grammars

# Lecture 13

# 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.

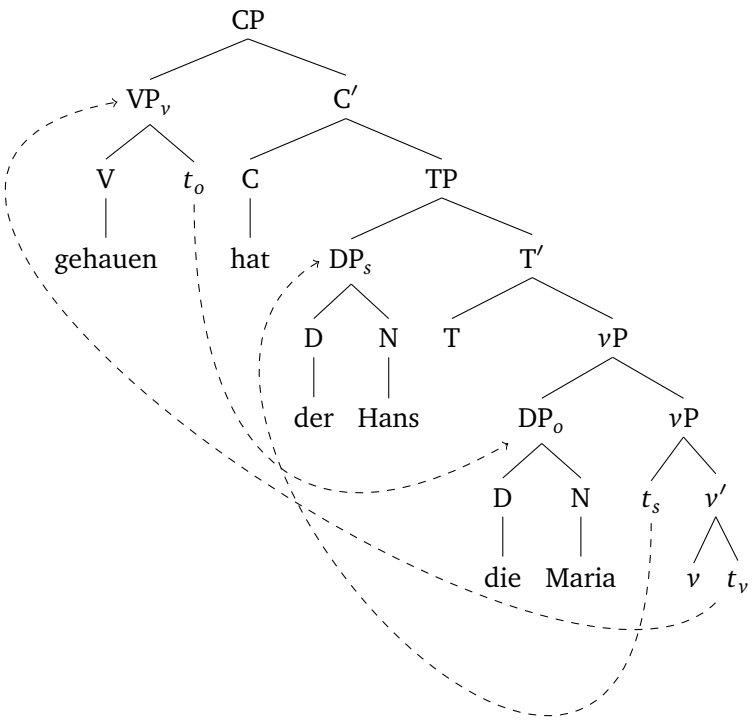
## 1 MG Derivations are Context-Free

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.

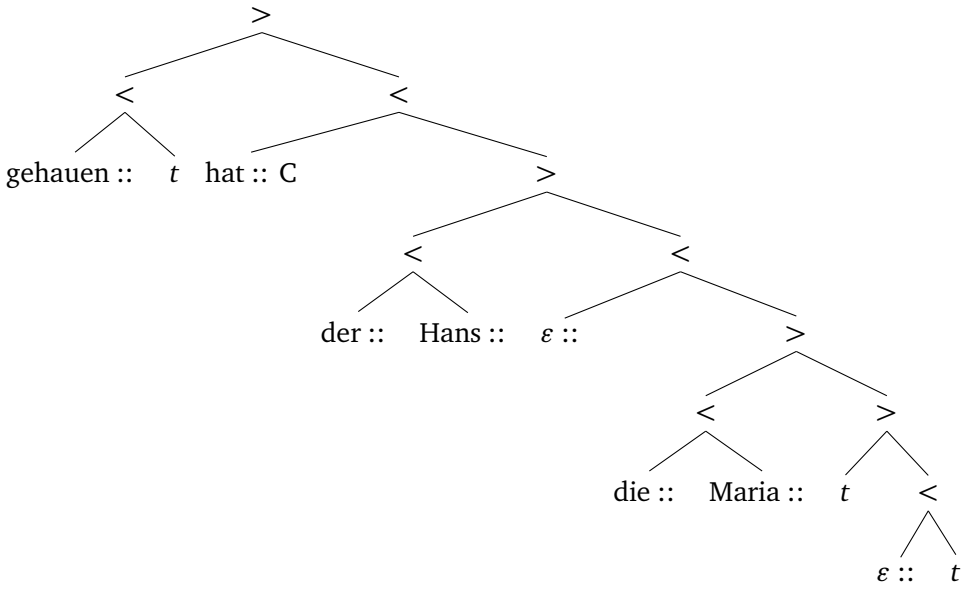
Consider the following sentence from Bavarian German, which displays topicalization of the V-head.

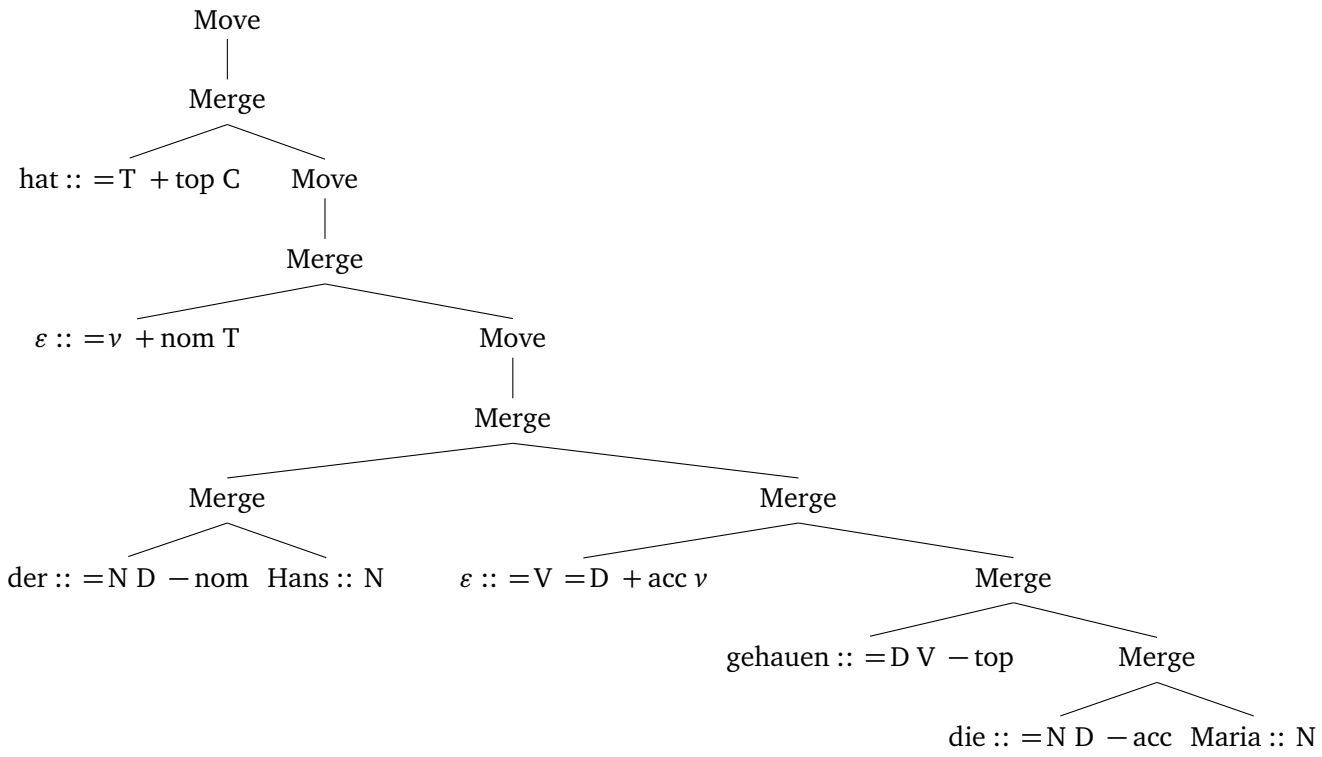
(1) Gehauen hat der Hans die Maria (, nicht geküsst).
beaten has the Hans the Maria (, not kissed).
'Hans beat Mary (he didn't kiss her).'

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).

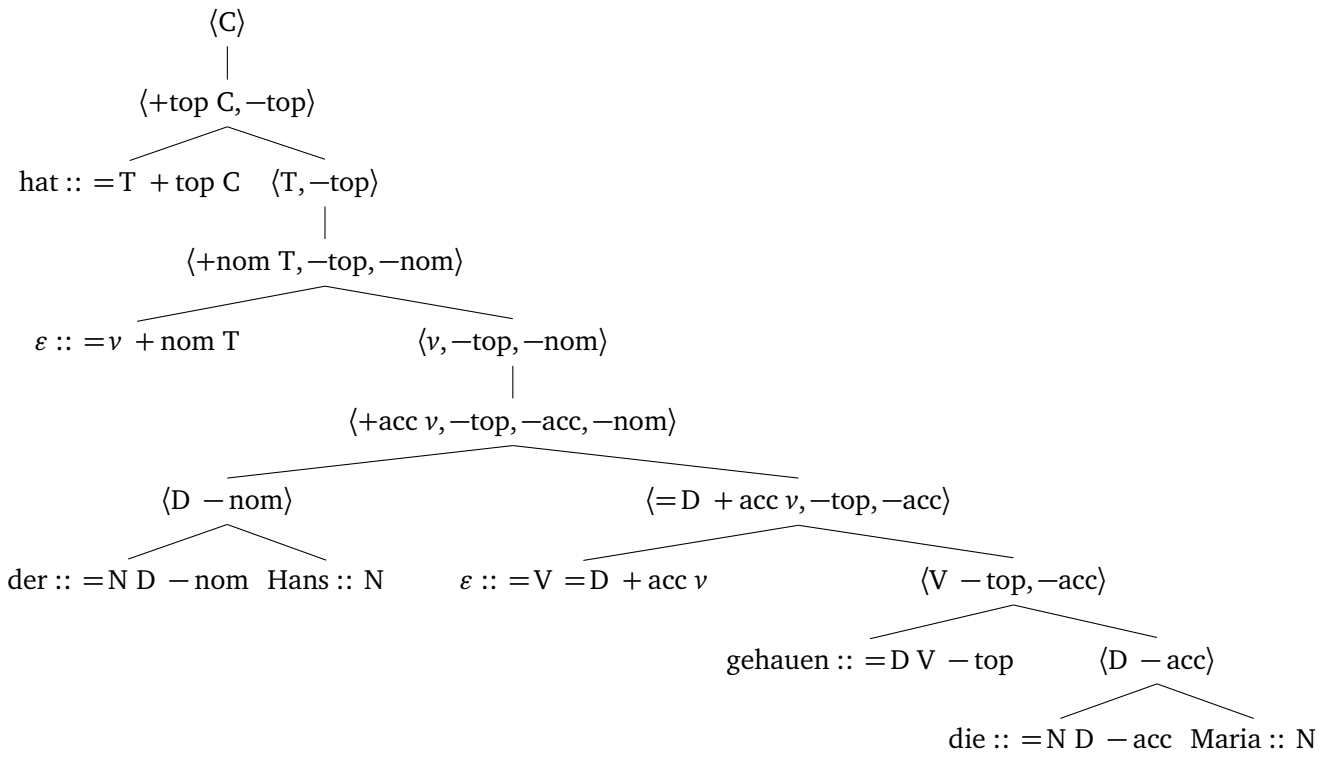



We can directly replicate this analysis with MGs.






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 $\alpha$ 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 $[n, \bullet]$.

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

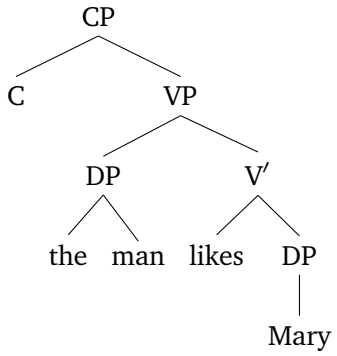
$$\textbf{Merge Right} \qquad \frac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i, \bullet \langle =\mathrm{F}_r\ \alpha \rangle \langle \mathrm{F} \rangle \beta, j]} \; \mathrm{F} \text{ a feature name of } G$$

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

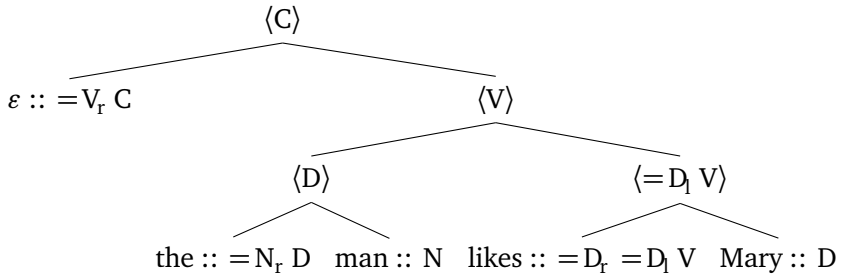
**LI (empty)** $\dfrac{[i, \bullet \langle \alpha \rangle \beta, j]}{[i, \bullet \beta, j]}\ w_i :: \alpha \in Lex_G, w_i = \varepsilon$

**Example 13.1** MG Top-Down Parse of *The man likes Mary*

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



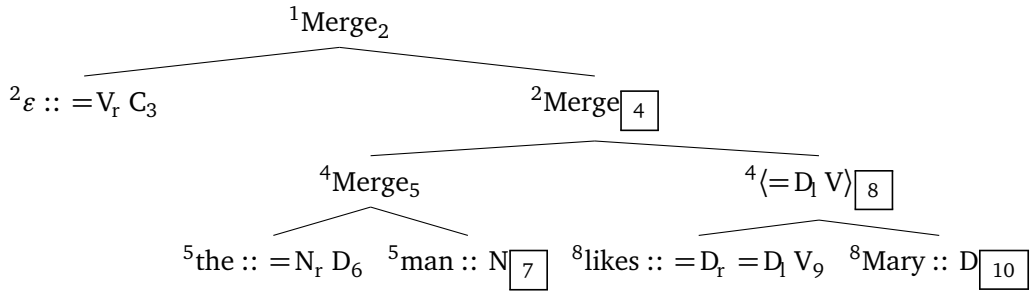

The feature-annotated Minimalist derivation tree for this sentence looks as follows.




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 $\beta$ differs minimally from $\alpha$,
- 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 $\beta_i$ 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 $\beta_i$ and $\beta_j$ are distinct,
- if $\delta = \varepsilon$, then each $\beta_i$ is identical to exactly one $\gamma_j$, and *vice versa*,
- otherwise, there is some $\beta_i$ such that $\beta_i = \delta$ and the previous condition holds for all $\beta_j$ 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 $\beta_i$ 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 $\beta_i$ 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 $\beta_i$ 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 $\phi$ of a parse item, $\phi_{/[f_1 \cdots f_n]}$ is the result of removing the placeholders $[f_1], \ldots, [f_n]$ from $\phi$.

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 $\phi$ or $\psi$. 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 $\beta_i$, 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.

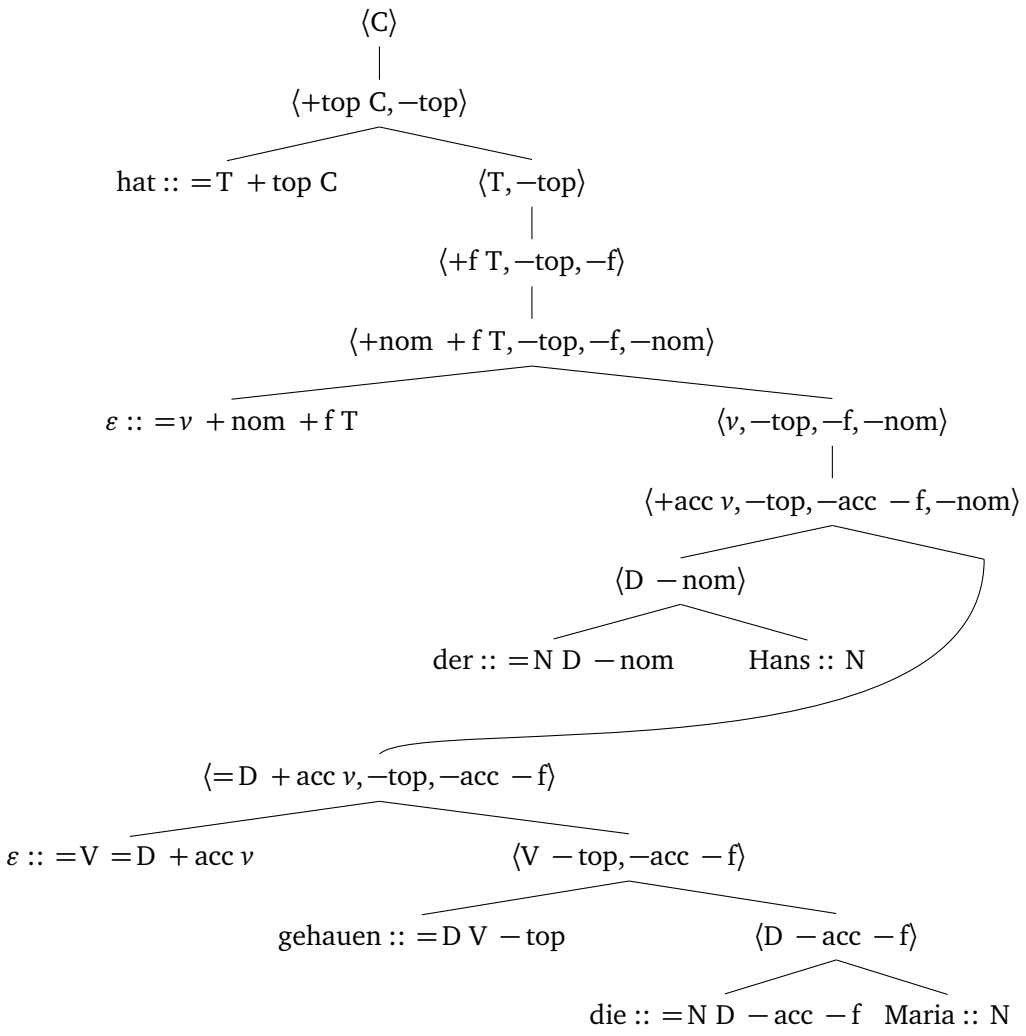
**Shift** $$\frac{[i, \phi \bullet x\psi]}{[i, \phi x \bullet \psi]}\ \phi \text{ not the empty string, } x \text{ a placeholder or an LI}$$

Example 13.2 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.

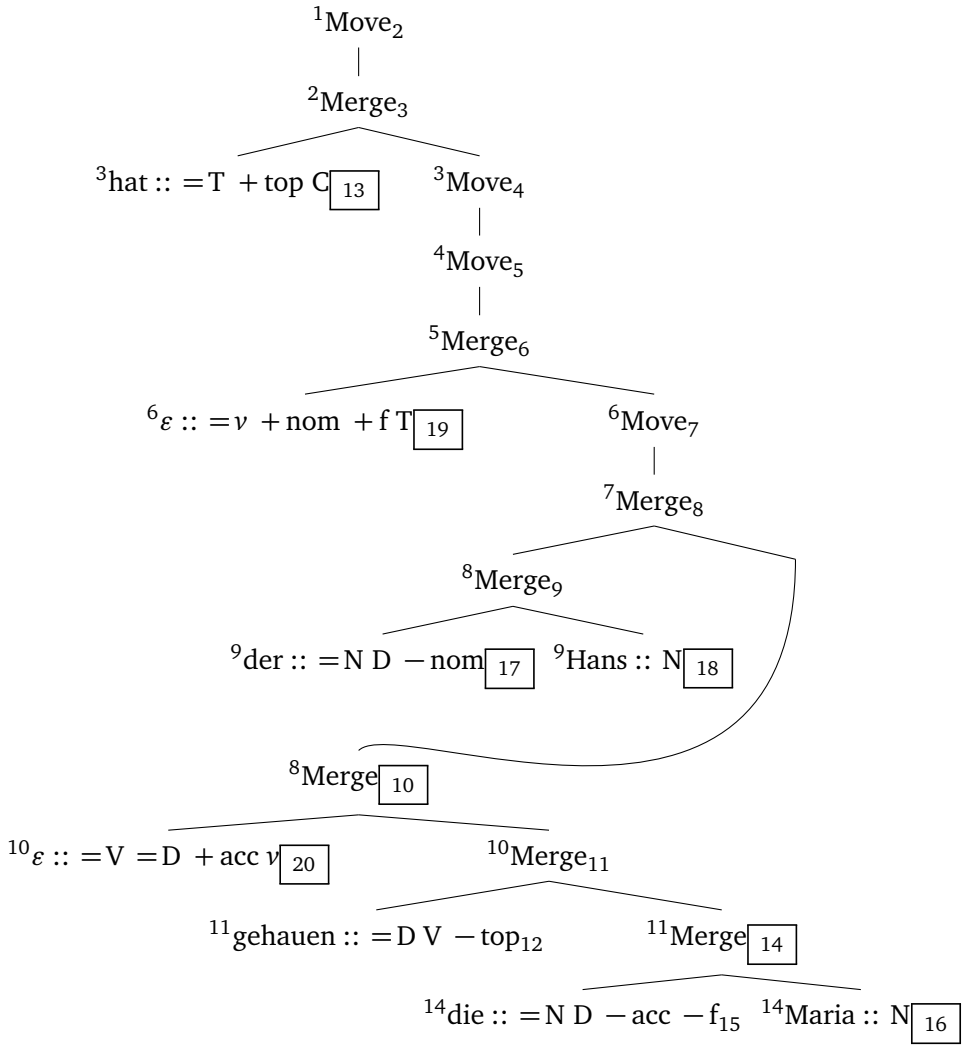



The parser infers the derivation with the following steps.

| **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] | |

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

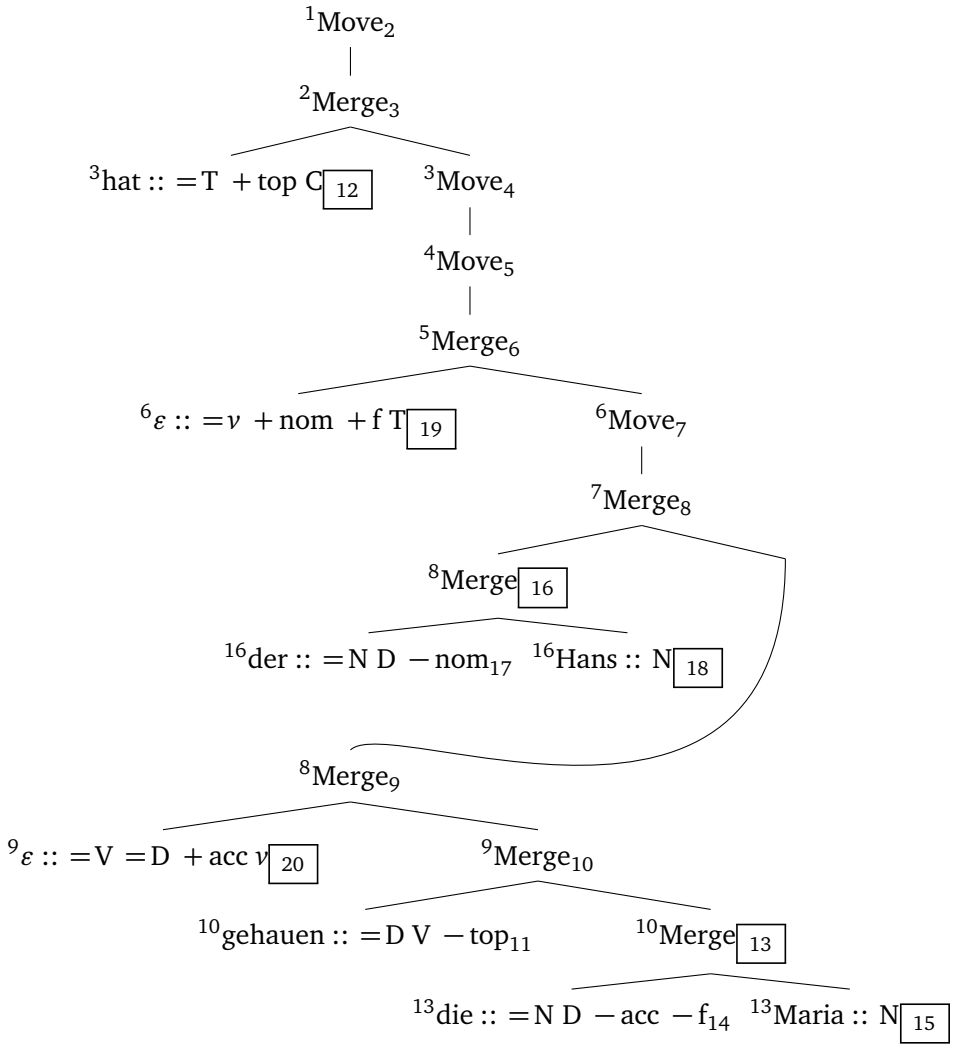
If we translate the MG into a CFG, the set of valid tuples corresponds to the set non-terminals of the CFG. So with a little bit of processing, the rules can be limited to only use non-terminals of this CFG. It would be more appealing, however, if we could

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 (2013).

Stabler's (2013) 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 13.3 A More Efficient Traversal of the Derivation Tree**

Using the search strategy outlined in Stabler (2013), 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 13.1.* 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 13.2.* Following up on the previous exercise, assume that we actually have a C-T-*v*-V spine where the subject starts in Spec,*v*P 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 13.3.* 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,*v*P 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. URL http://thomasgraf.net/doc/papers/Graf13Thesis.pdf.

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. 2013. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5:611–633. URL https://dx.doi.org/10.1111/tops.12031.

# Lecture 14

# 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 (2013) 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 $[_{\text{SC}}$ that the employee$_i$ $[_{\text{RC}}$ who the manager hired $t_i]$ stole office supplies] worried the executive.
  b. The executive$_i$ $[_{\text{RC}}$ who the fact $[_{\text{SC}}$ that the employee stole offices supplies] worried $t_i]$ hired the manager.

(2) a. The reporter$_i$ $[_{\text{RC}}$ who $t_i$ attacked the senator] admitted the error.
  b. The reporter$_i$ $[_{\text{RC}}$ 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$_{\text{Lex}}$** 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$_{\text{Lex}}$** 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. 14.1–14.4. 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.

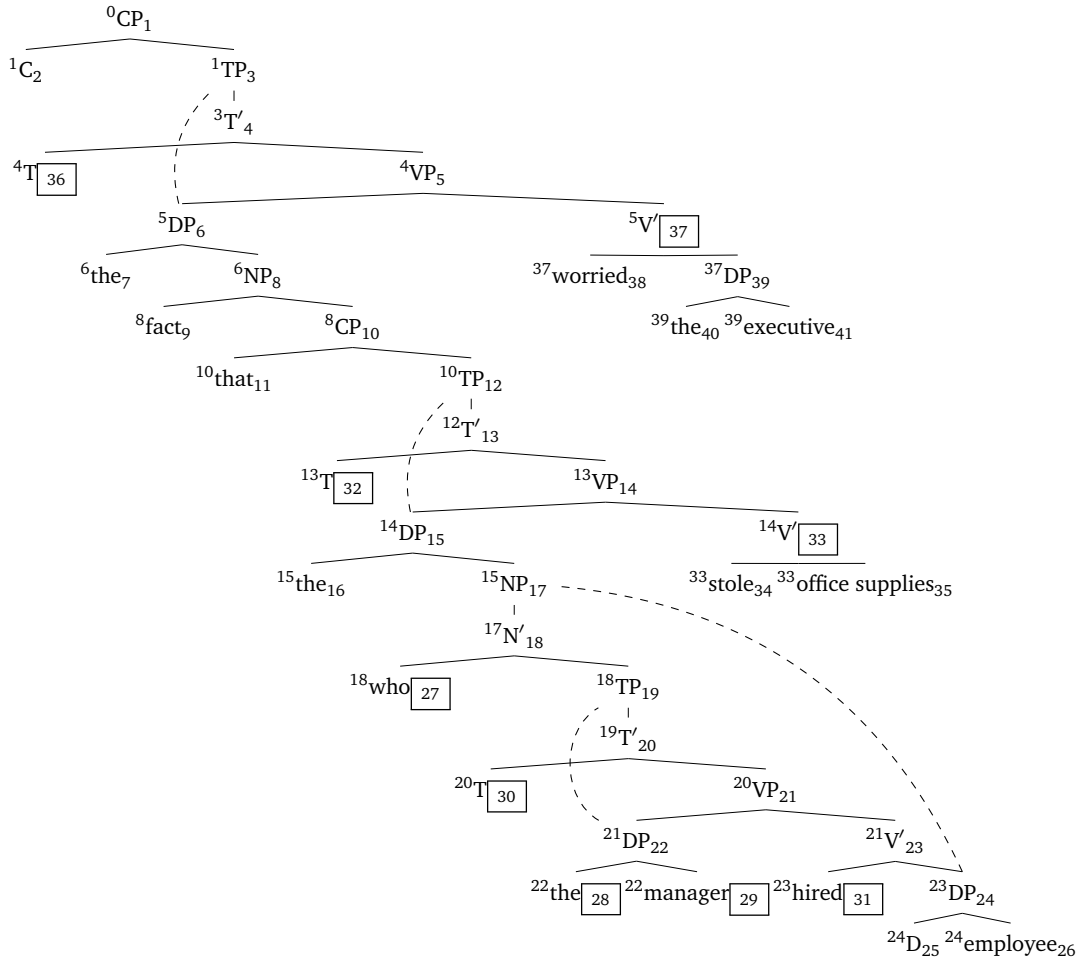



Figure 14.1: Merge-eager parse of sentential complement with embedded relative clause; **Max** = 32/32, $\textbf{Max}_{\textbf{Lex}}$ = 32/9, **Payload** = 9/6, $\textbf{Payload}_{\textbf{Lex}}$ = 7/4

As can be seen in Tab. 14.1, 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 $\textbf{Payload}_{\textbf{Lex}}$

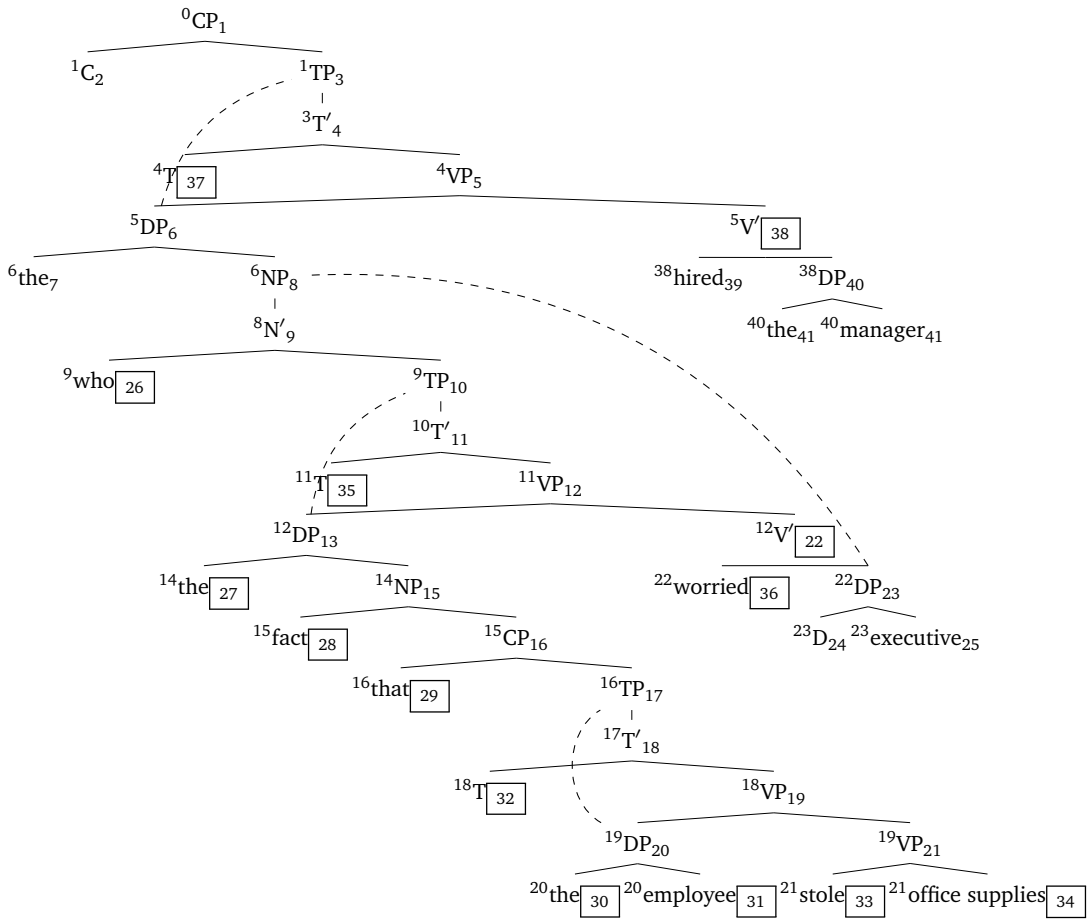



Figure 14.2: Merge-eager parse of relative clause containing a sentential complement; **Max** = 33/33, $\textbf{Max}_{\textbf{Lex}}$ = 33/17, **Payload** = 14/11, $\textbf{Payload}_{\textbf{Lex}}$ = 12/9

| | **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 14.1: Overview of processing predictions for SC/RC and RC/SC

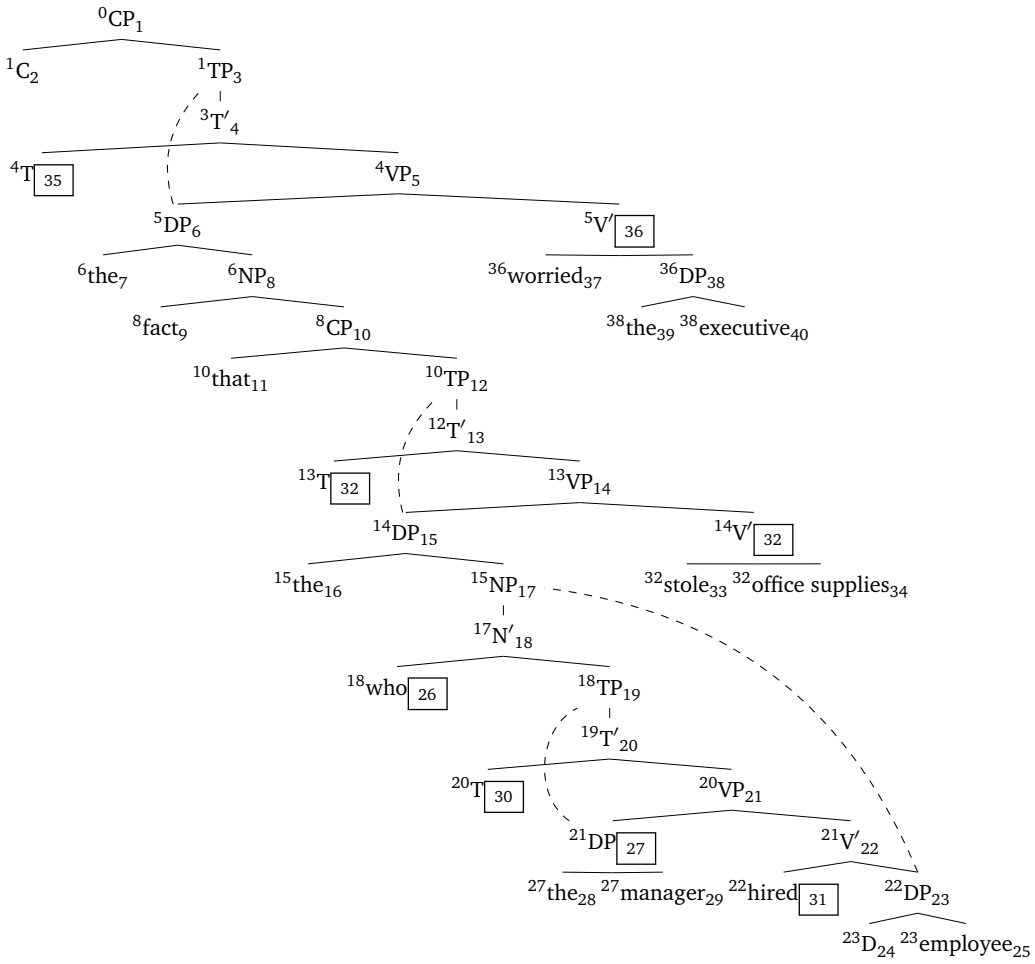



Figure 14.3: Move-eager parse of sentential complement with embedded relative clause; **Max** $= 31/31$, $\textbf{Max}_{\textbf{Lex}} = 31/8$, **Payload** $= 8/5$, $\textbf{Payload}_{\textbf{Lex}} = 5/2$

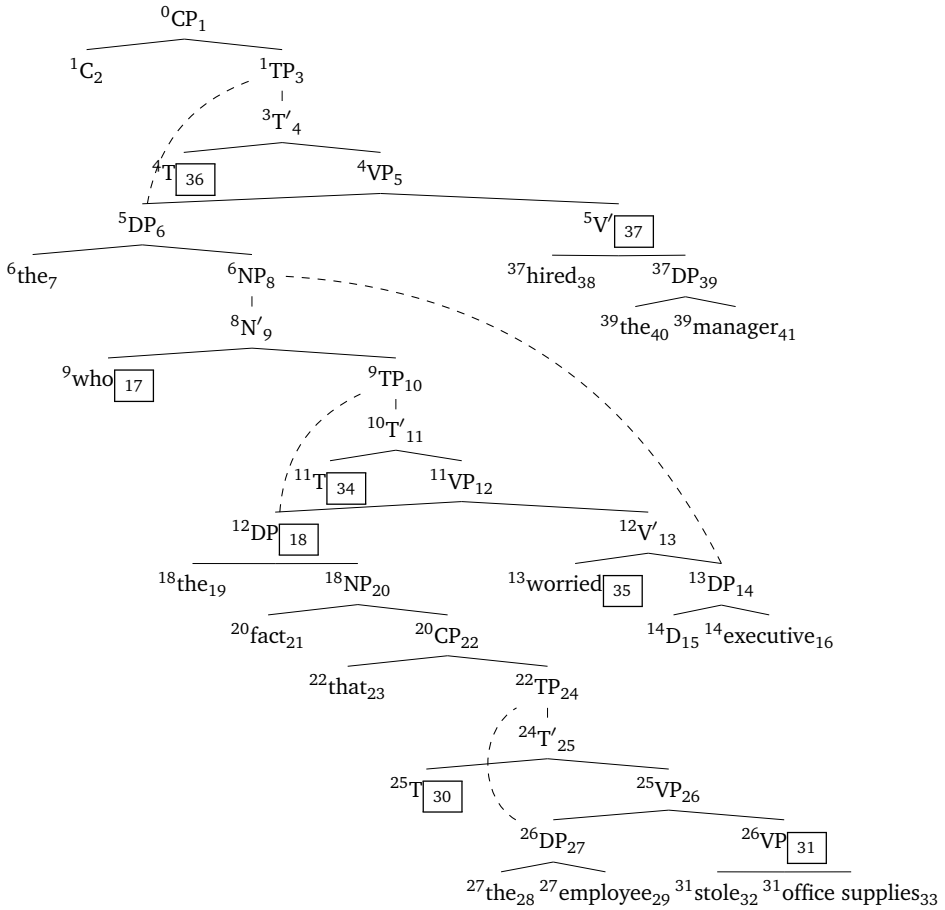



Figure 14.4: Move-eager parse of relative clause containing a sentential complement; **Max** $= 32/32$, $\textbf{Max}_{\textbf{Lex}} = 32/22$, **Payload** $= 8/5$, $\textbf{Payload}_{\textbf{Lex}} = 5/2$

(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 14.2: Overview of processing predictions for SC/RC and RC/SC

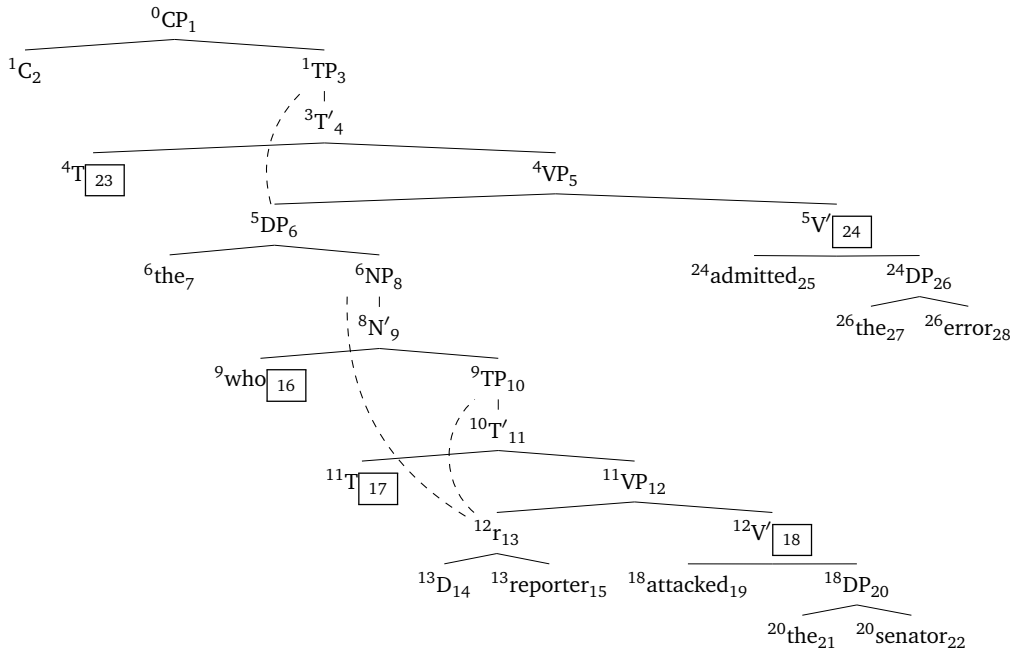



Figure 14.5: Merge-eager and Move-eager parse of relative clause with subject gap; **Max** = 19/19, **Max$_{\text{Lex}}$** = 19/7, **Payload** = 5/3, **Payload$_{\text{Lex}}$** = 3/1

## 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.

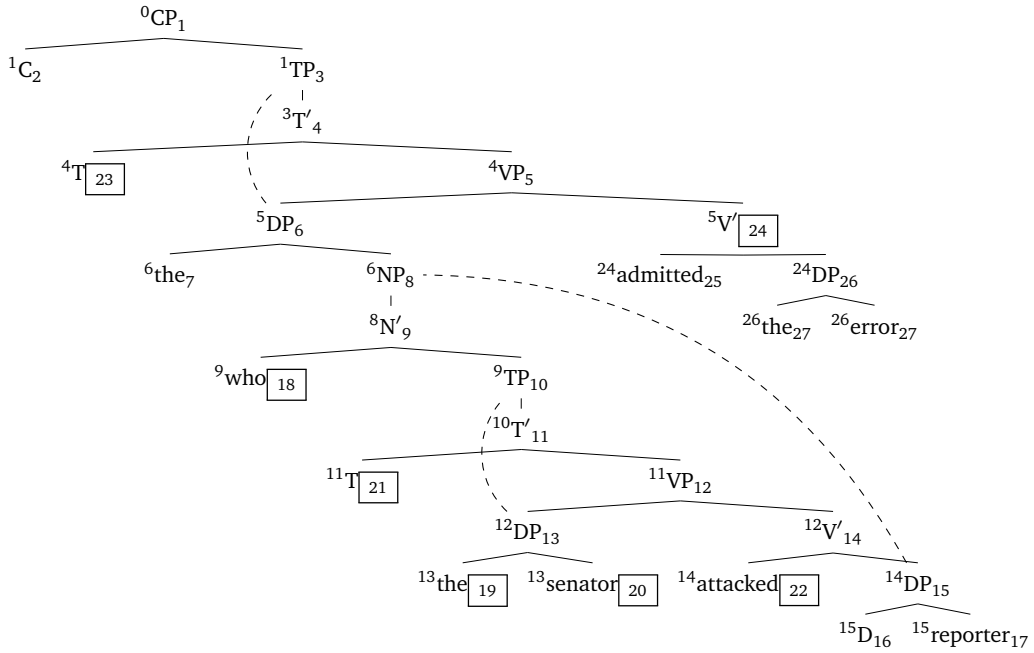



Figure 14.6: Merge-eager parse of relative clause with object gap; **Max** = 19/19, $\textbf{Max}_{\textbf{Lex}}$ = 19/9, **Payload** = 7/5, $\textbf{Payload}_{\textbf{Lex}}$ = 6/4

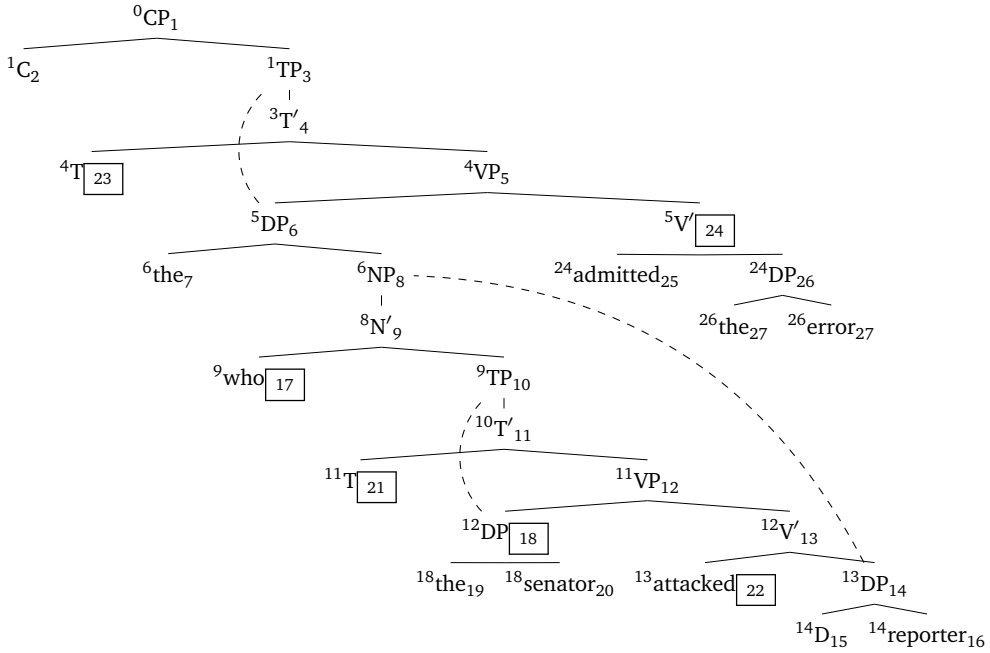



Figure 14.7: Move-eager parse of relative clause with object gap; **Max** = 19/19, $\textbf{Max}_{\textbf{Lex}}$ = 19/9, **Payload** = 6/4, $\textbf{Payload}_{\textbf{Lex}}$ = 4/2

Stabler, Edward P. 2013. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5:611–633. URL https://dx.doi.org/10.1111/tops.12031.

## Lecture 15

# (Quasi-)Deterministic Parsing

## Lecture 16

# Partial Parsing