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.

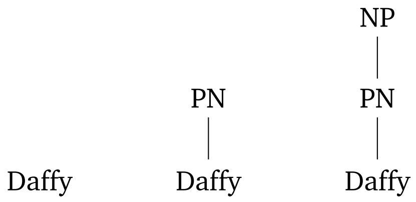
1 Intuition

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 γ once it has reached Y in a bottom-up fashion.

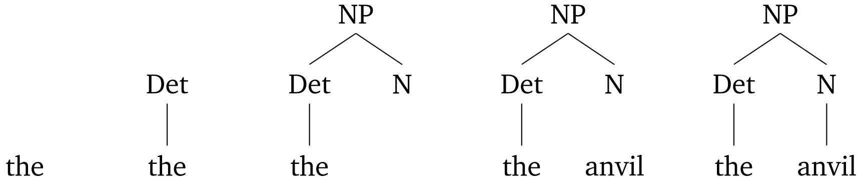
Consider our familiar toy grammar.

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

Rather than conjecturing S → NP VP right away, 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.



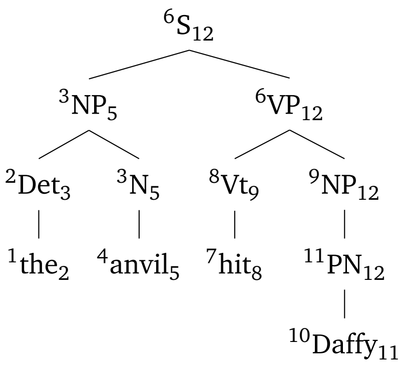
Alternatively, it may involve a mixture of bottom-up reduction and left-corner condition prediction: read *the*, reduce to Det, use the rewrite rule NP → Det N in your top-down prediction, read *anvil*, reduce to N, reduce Det N to NP.



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 $p_1, \dots, p_n$ need to be built according to the left-corner prediction using some l_i, and
- which phrase is built from $l_i, p_i, \dots, p_n$

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

- i is the current position in the string,
- α is the list of identified unused nodes (derived via bottom-up reduction), and
- β is a list of labeled lists of phrases to be built (the top-down predictions).

For instance, the item $[1, \bullet[_{\mathrm{NP}}\ \mathrm{N}]]$ 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.

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

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

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

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

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

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 β a list that is labeled with the conjectured mother of N and contains the conjectured sisters of N. The completion rule, finally, states that once we 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	$\frac{[i, \alpha \bullet a\beta]}{[i+1, \alpha \bullet \beta]}$		$\frac{[i, \alpha N \bullet [_M\ N\gamma]\ \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}$
Shift		$\frac{[\alpha \bullet \beta, j]}{[\alpha a \bullet \beta, j+1]}$	$\frac{[i, \alpha \bullet \beta]}{[i+1, \alpha a \bullet \beta]}$
Predict	$\frac{[i, \alpha \bullet N\beta]}{[i, \alpha \bullet \gamma\beta]}$		$\frac{[i, \alpha N \bullet \beta]}{[i, \alpha \bullet [_M\ \gamma]\ \beta]}$
Reduce		$\frac{[\alpha\gamma \bullet \beta, j]}{[\alpha N \bullet \beta, j]}$	$\frac{[i, \alpha\gamma \bullet \beta]}{[i, \alpha N \bullet \beta]}$
Complete			$\frac{[i, \alpha \bullet [_M\ ]\ \beta]}{[i, \alpha M \bullet \beta]}$

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 $Y_1, \dots, Y_n$ 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 $\mathrm{lc}(Y)$ for the set $\{X \mid \langle X, Y\rangle \in \mathrm{lc}\}$ 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.

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

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 ⋆ at the appropriate position in the rewrite rule. For example, NP → Det A N might be written NP →Det A ⋆ 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 $G^{\mathrm{lc}} := \left\langle N', T, S, R' \right\rangle$ such that

$$\begin{array}{ll}
A \rightarrow \; a \; A\text{-}a & \text{for all } A \in N \text{ and } a \in T \\
A \rightarrow \; A\text{-}B & \text{for all } A \in N \text{ and } B \rightarrow \varepsilon \in R \\
A\text{-}X \rightarrow \; \beta \; A\text{-}B & \text{for all } A \in N \text{ and } B \rightarrow X\beta \in R \\
A\text{-}A \rightarrow \varepsilon & \text{for all } A \in N
\end{array}$$

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-$X \to \beta\ A$-B for all non-terminals A and every B such that $B \to X\ \beta$. 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-$A \rightarrow \varepsilon$ for all non-terminals A of the original grammar.

S-S	→	ε	Det-Det	→	ε
NP-NP	→	ε	N-N	→	ε
VP-VP	→	ε	PN-PN	→	ε
			Vi-Vi	→	ε
			Vt-Vt	→	ε

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 (Grune and Jacobs 2008:cf.). 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

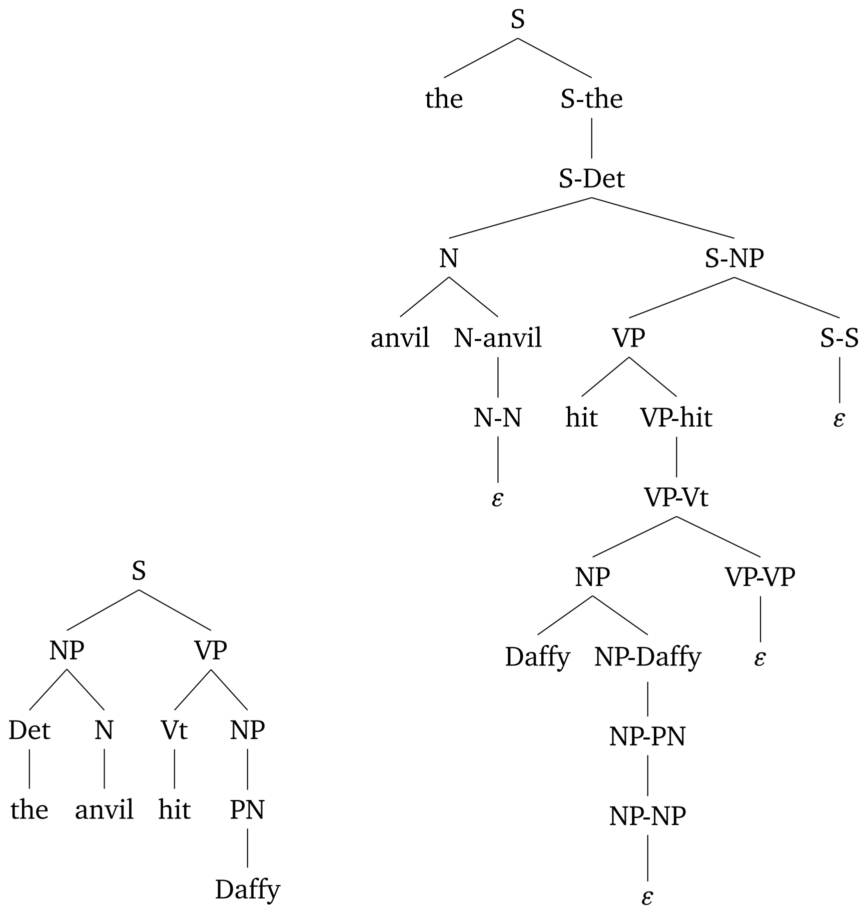


Figure 6.1: Parse tree for *The anvil hit Daffy* with original and transformed grammar

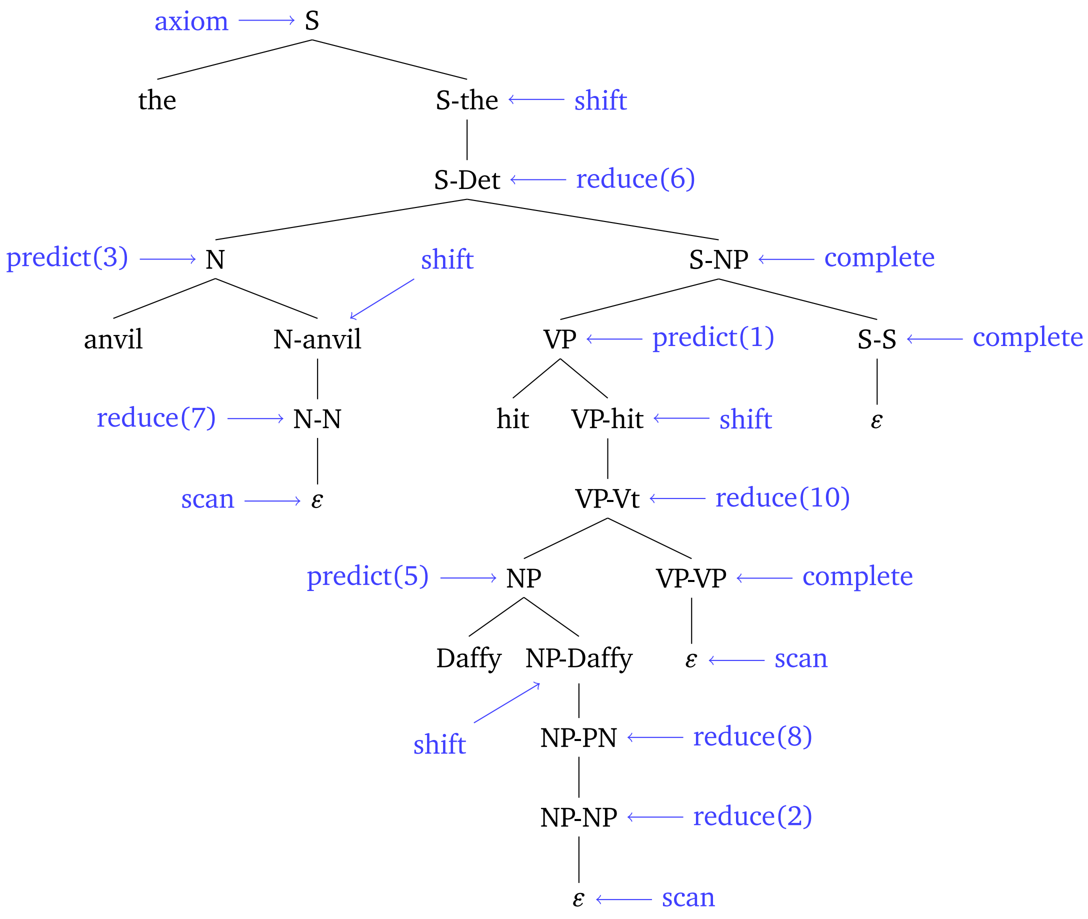


Figure 6.2: Recursive descent traversal of the parse tree closely mirrors the steps of an LC parser

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. ⊙

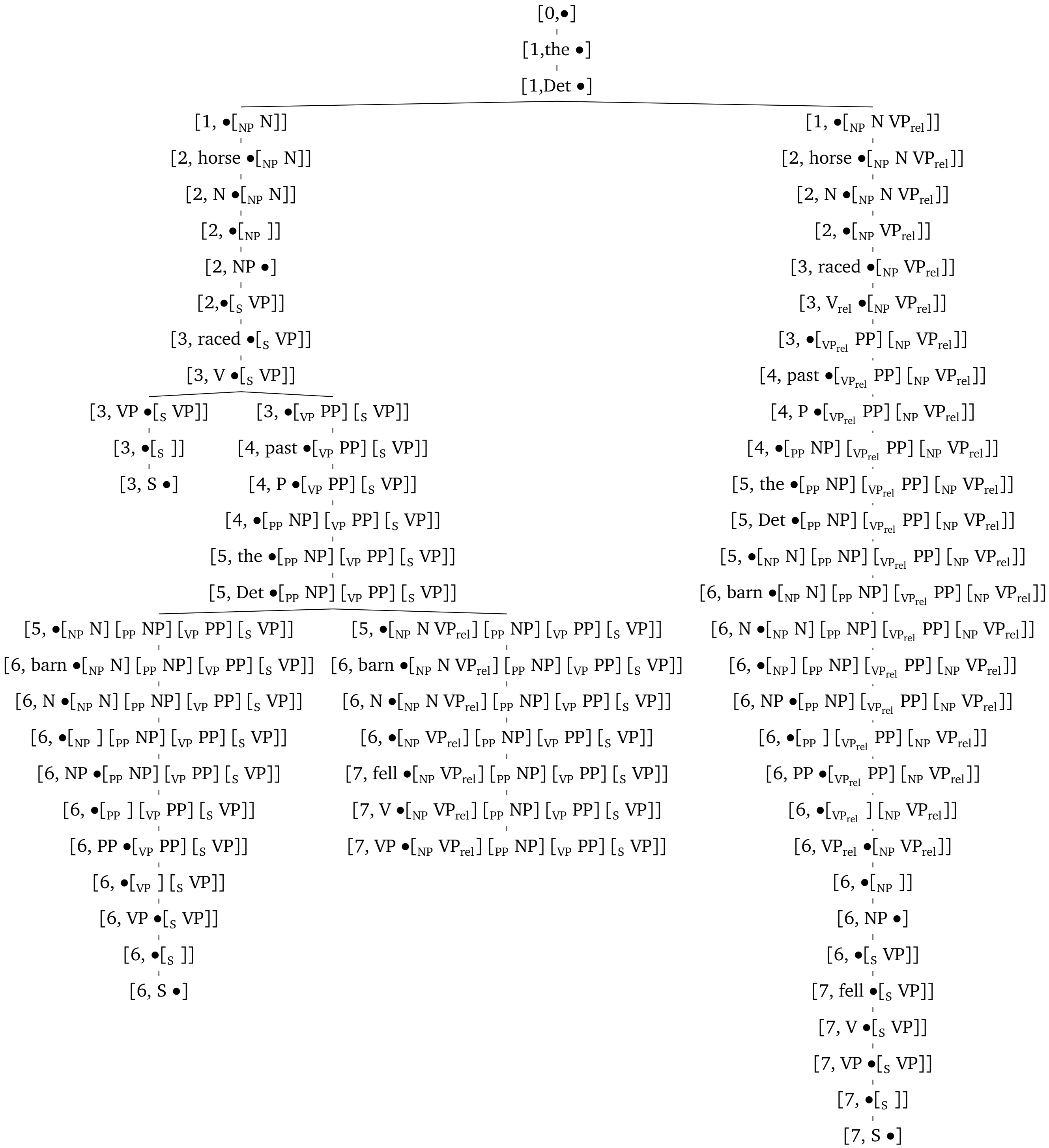

[0,•]
[1,the •]
[1,Det •]
[1, •[$_{NP}$ N]]
[2, horse •[$_{NP}$ N]]
[2, N •[$_{NP}$ N]]
[2, •[$_{NP}$]]
[2, NP •]
[2,•[$_{S}$ VP]]
[3, raced •[$_{S}$ VP]]
[3, V •[$_{S}$ VP]]
[3, VP •[$_{S}$ VP]]
[3, •[$_{S}$]]
[3, S •]
[3, •[$_{VP}$ PP] [$_{S}$ VP]]
[4, past •[$_{VP}$ PP] [$_{S}$ VP]]
[4, P •[$_{VP}$ PP] [$_{S}$ VP]]
[4, •[$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[5, the •[$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[5, Det •[$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[5, •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, barn •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, N •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, •[$_{NP}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, NP •[$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, •[$_{PP}$] [$_{VP}$ PP] [$_{S}$ VP]]
[6, PP •[$_{VP}$ PP] [$_{S}$ VP]]
[6, •[$_{VP}$] [$_{S}$ VP]]
[6, VP •[$_{S}$ VP]]
[6, •[$_{S}$]]
[6, S •]
[5, •[$_{NP}$ N VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, barn •[$_{NP}$ N VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, N •[$_{NP}$ N VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[6, •[$_{NP}$ VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[7, fell •[$_{NP}$ VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[7, V •[$_{NP}$ VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[7, VP •[$_{NP}$ VP$_{rel}$] [$_{PP}$ NP] [$_{VP}$ PP] [$_{S}$ VP]]
[1, •[$_{NP}$ N VP$_{rel}$]]
[2, horse •[$_{NP}$ N VP$_{rel}$]]
[2, N •[$_{NP}$ N VP$_{rel}$]]
[2, •[$_{NP}$ VP$_{rel}$]]
[3, raced •[$_{NP}$ VP$_{rel}$]]
[3, V$_{rel}$ •[$_{NP}$ VP$_{rel}$]]
[3, •[$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[4, past •[$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[4, P •[$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[4, •[$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[5, the •[$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[5, Det •[$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[5, •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, barn •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, N •[$_{NP}$ N] [$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, •[$_{NP}$] [$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, NP •[$_{PP}$ NP] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, •[$_{PP}$] [$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, PP •[$_{VP_{rel}}$ PP] [$_{NP}$ VP$_{rel}$]]
[6, •[$_{VP_{rel}}$] [$_{NP}$ VP$_{rel}$]]
[6, VP$_{rel}$ •[$_{NP}$ VP$_{rel}$]]
[6, •[$_{NP}$]]
[6, NP •]
[6, •[$_{S}$ VP]]
[7, fell •[$_{S}$ VP]]
[7, V •[$_{S}$ VP]]
[7, VP •[$_{S}$ VP]]
[7, •[$_{S}$]]
[7, S •]

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:

$$\begin{array}{rcl@{\qquad}rcl}
\text{S} & \rightarrow & \text{DP VP} & \text{Poss} & \rightarrow & \text{'s} \\
\text{DP} & \rightarrow & \text{DP D}' & \text{N} & \rightarrow & \text{father} \mid \text{car} \mid \text{exhaust pipe} \\
\text{D}' & \rightarrow & \text{Poss NP} & \text{PN} & \rightarrow & \text{John} \\
\text{NP} & \rightarrow & \text{N} \mid \text{PN} & \text{V} & \rightarrow & \text{disappeared} \\
\text{VP} & \rightarrow & \text{V} & & &
\end{array}$$

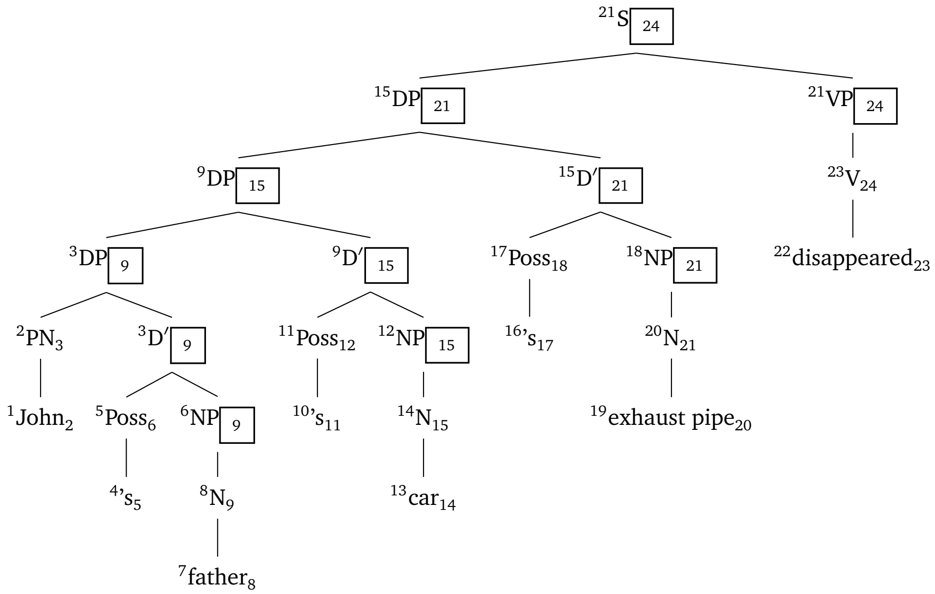
parse item	**inference rule**
$[0, \bullet]$	axiom
$[1, \text{John} \bullet]$	shift
$[1, \text{PN} \bullet]$	reduce
$[1, \text{DP} \bullet]$	reduce
$[1, \bullet[_{\text{DP}}\ \text{D}']]$	predict
$[2, \text{'s} \bullet[_{\text{DP}}\ \text{D}']]$	shift
$[2, \text{Poss} \bullet[_{\text{DP}}\ \text{D}']]$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

$[7, \text{exhaust pipe} \bullet[_{\text{D}'}\ \text{NP}]\ [_{\text{DP}}\ \text{D}']]$	shift
$[7, \text{N} \bullet[_{\text{D}'}\ \text{NP}]\ [_{\text{DP}}\ \text{D}']]$	reduce
$[7, \text{NP} \bullet[_{\text{D}'}\ \text{NP}]\ [_{\text{DP}}\ \text{D}']]$	reduce
$[7, \bullet[_{\text{D}'}\ ]\ [_{\text{DP}}\ \text{D}']]$	scan
$[7, \text{D}' \bullet[_{\text{DP}}\ \text{D}']]$	complete
$[7, \bullet[_{\text{DP}}\ ]]$	scan
$[7, \text{DP} \bullet]$	complete
$[7, \bullet[_{\text{S}}\ \text{VP}]]$	predict
$[8, \text{disappeared} \bullet[_{\text{S}}\ \text{VP}]]$	shift
$[8, \text{VP} \bullet[_{\text{S}}\ \text{VP}]]$	reduce
$[8, \bullet[_{\text{S}}\ ]]$	scan
$[9, \text{S} \bullet]$	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? ⊙

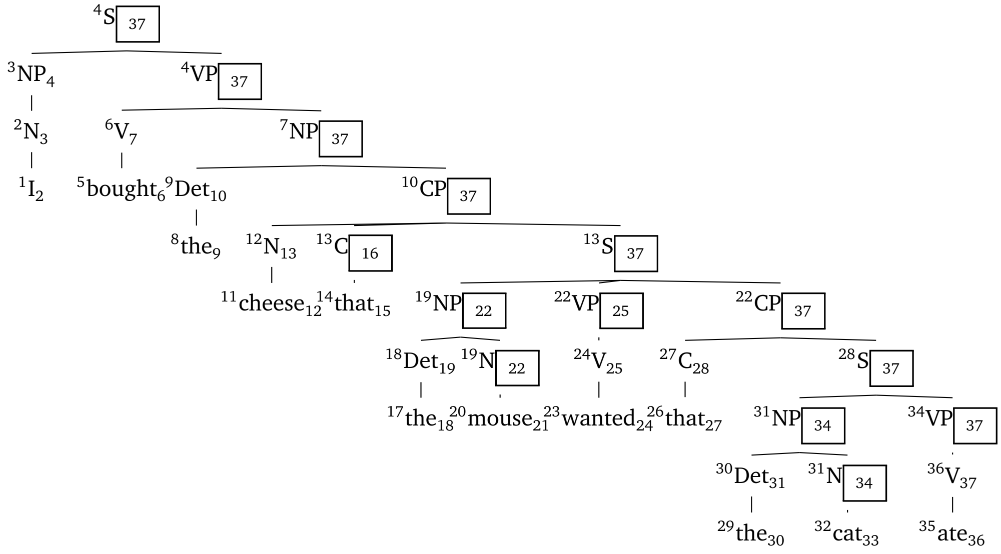
Since the LC parser completely finishes a possessive DP before starting the next higher one, it behaves similar to the shift reduce parser for left embeddings, with memory load remaining mostly unaffected by the number of embeddings. This can also be seen in the annotated parse tree, which may have a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



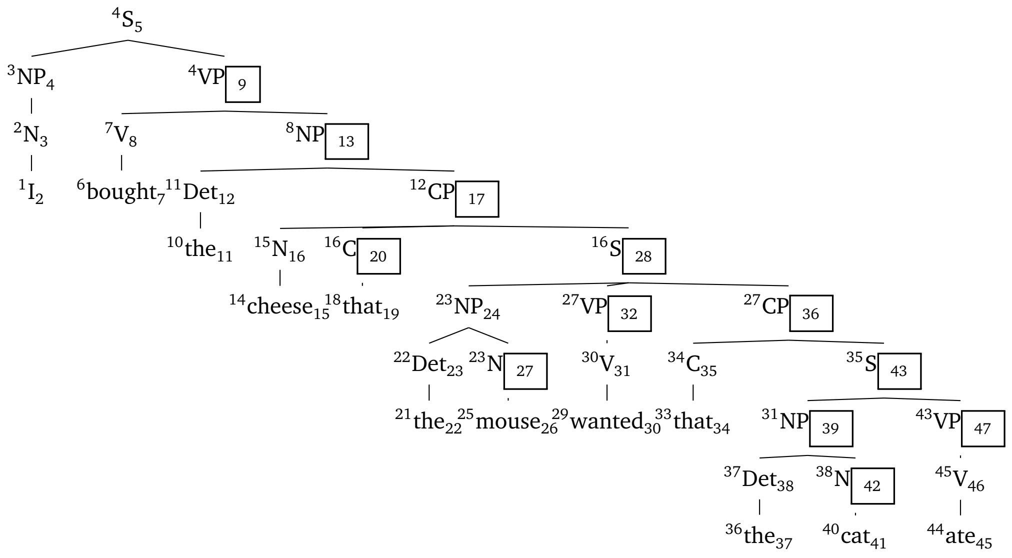
The payload of right embedding is enormous with an LC parser, and tenure also increases with every level of embedding.

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.

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

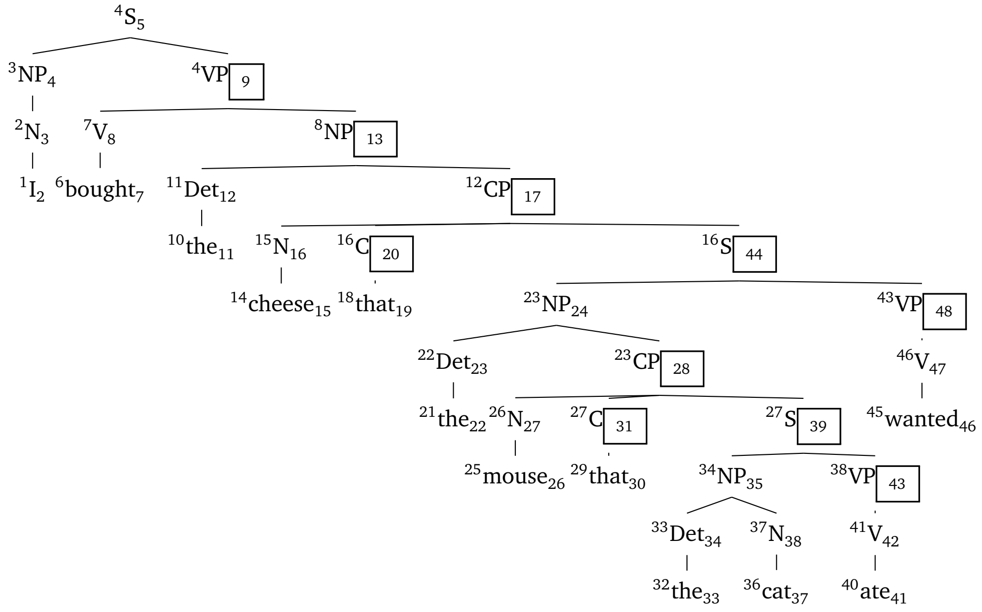
Resnik (1992) studies this type of LC parser in detail and calls it *arc-eager* (the intuition being that if there are conjectured nodes than can be linked by an arc, the parser establishes that arc immediately). Resnik concludes that an arc-eager LC parser predicts both left embedding and right embedding 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.