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.

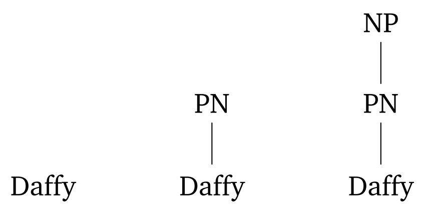
1 Intuition

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.

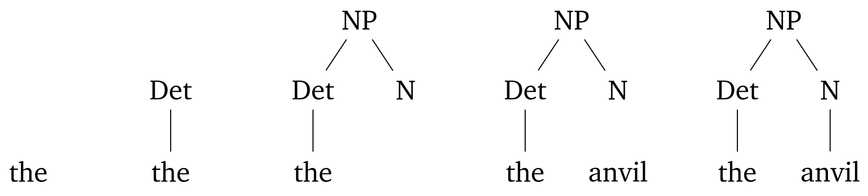
Consider our familiar toy grammar.

1)	S	→	NP VP	6)	Det	→	a | the
2)	NP	→	PN	7)	N	→	car | truck | anvil
3)	NP	→	Det	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.



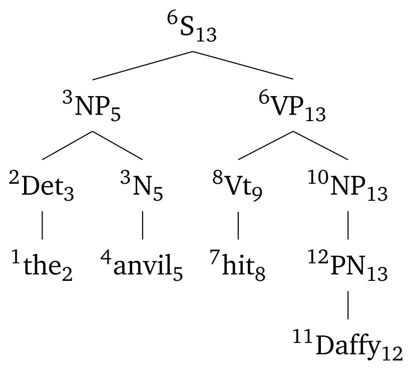
Alternatively, it may involve a mixture of bottom-up reduction and left-corner condition prediction: read *the*, reduce to Det, use the rewrite rule NP → Det N in your top-down prediction, read *anvil*, reduce to N, reduce Det N to NP.



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

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

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

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

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

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

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

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

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,•[$_{\text{NP}}$ N]]	predict(3)
[2,anvil •[$_{\text{NP}}$ N]]	shift
[2,N •[$_{\text{NP}}$ N]]	reduce(7)
[2,•[$_{\text{NP}}$]]	scan
[2,NP •]	complete
[2,•[$_{\text{S}}$ VP]]	predict(1)
[3,hit •[$_{\text{S}}$ VP]]	shift
[3,V •[$_{\text{S}}$ VP]]	reduce(10)
[3,•[$_{\text{VP}}$ NP] [$_{\text{S}}$ VP]]	predict(5)
[4,Daffy •[$_{\text{VP}}$ NP] [$_{\text{S}}$ VP]]	shift
[4,PN •[$_{\text{VP}}$ NP] [$_{\text{S}}$ VP]]	reduce(8)
[4,NP •[$_{\text{VP}}$ NP] [$_{\text{S}}$ VP]]	reduce(2)
[4,•[$_{\text{VP}}$] [$_{\text{S}}$ VP]]	scan
[4, VP •[$_{\text{S}}$ VP]]	complete
[4,•[$_{\text{S}}$]]	scan
[4,S •]	complete

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).

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:

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

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

	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

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.

2.2 Adding Top-Down Filtering

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. An LC parser where left corners are string of length 2 or more is called a *generalized left-corner parser*. It uses the same rules as a standard left-corner parser, except that the prediction rule is slightly modified.

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

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

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

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

3 Left Corner Parsing as Top-Down Parsing

4 Psycholinguistic Adequacy

Exercise 6.3. Show that just like top-down and bottom-up parsers, left-corner parsers struggle with garden path sentences. ⊙

Exercise 6.4. Recall the two structures that were proposed for *John's father's car's exhaust pipe disappeared* in exercises 5.4 and 5.5. Write down the left-corner parse tables for both structures. Based on these parse tables, annotate the trees with subscripts and superscripts in the usual fashion. Determine the payload as well as MaxTen and MaxSum. How does the left-corner parser fare in comparison to the recursive descent and shift reduce parsers? ⊙

Exercise 6.5. Are merely local syntactic coherence effects expected with a left-corner parser? ⊙

References and Further Reading

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