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

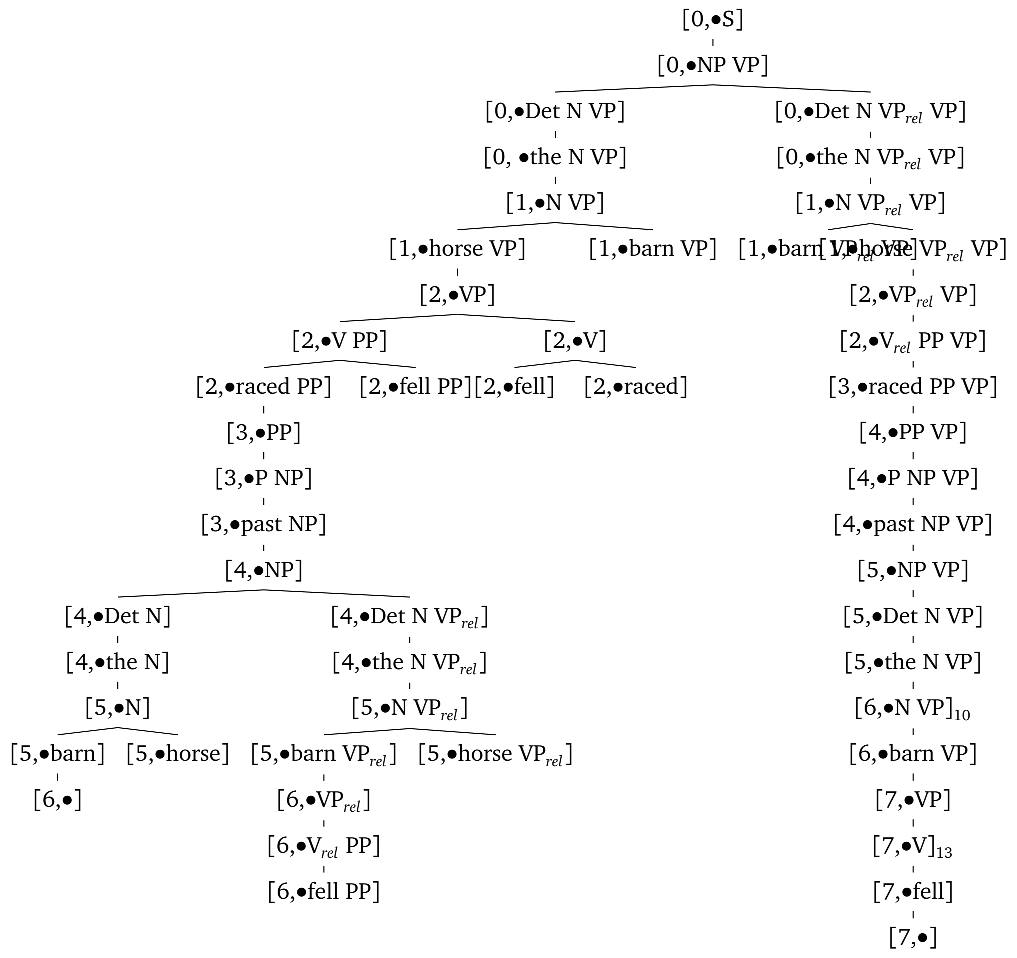
Recall from Chap. 1 that we distinguish between parsers and recognizers. A recognizer only has to determine for a given sentence whether it is well-formed, whereas a parser also has to find the right tree structures.

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 γ] 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.

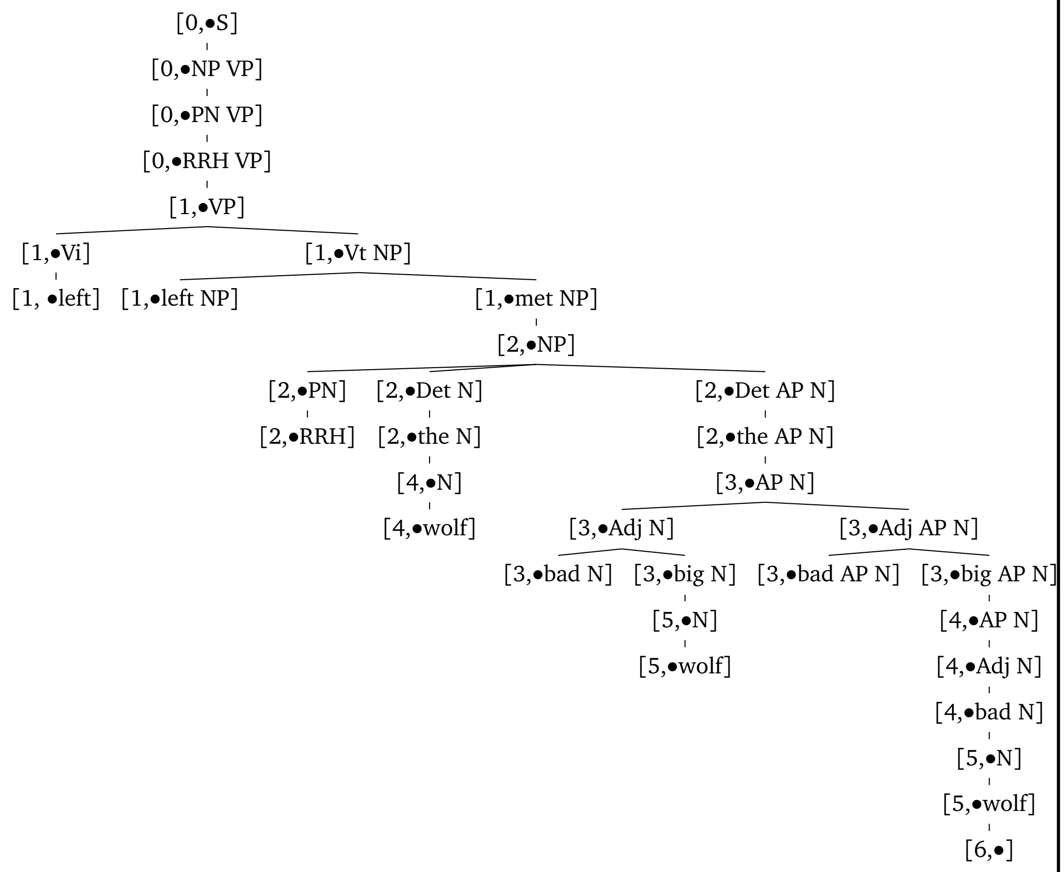
If parse items are discarded immediately after being used in an inference rule, the recognizer also loses the ability to recycle successful partial parses.

Example 7.2 Needless Reexploration of Successful Partial Parses

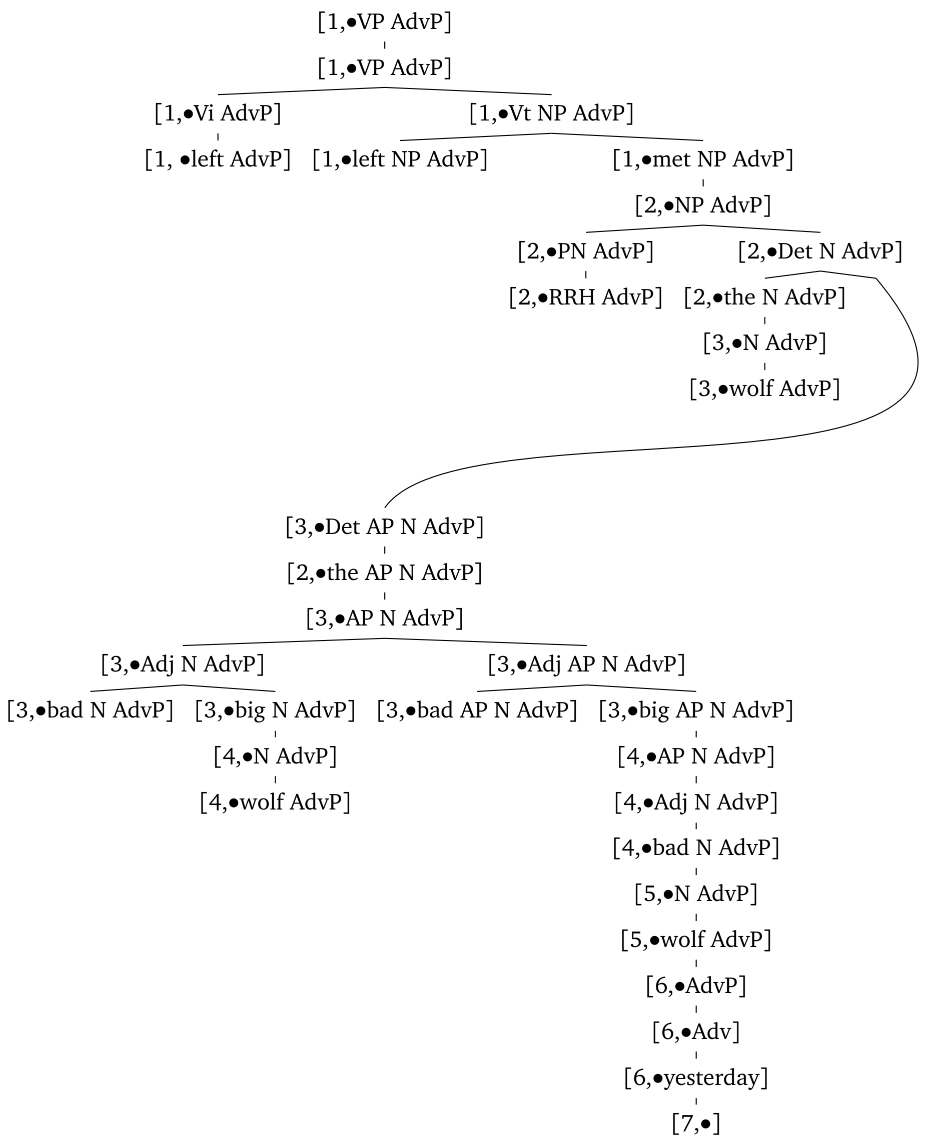
Suppose the sentence *Red Riding Hood met the big, bad wolf yesterday* is to be recognized using the following grammar:

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

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.

	1	2	3	4	5	6	7	8
0								
1	▨							
2	▨	▨						
3	▨	▨	▨					
4	▨	▨	▨	▨				
5	▨	▨	▨	▨	▨			
6	▨	▨	▨	▨	▨	▨		
7	▨	▨	▨	▨	▨	▨	▨	
	the	anvil	hit	the	duck	on	the	head

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

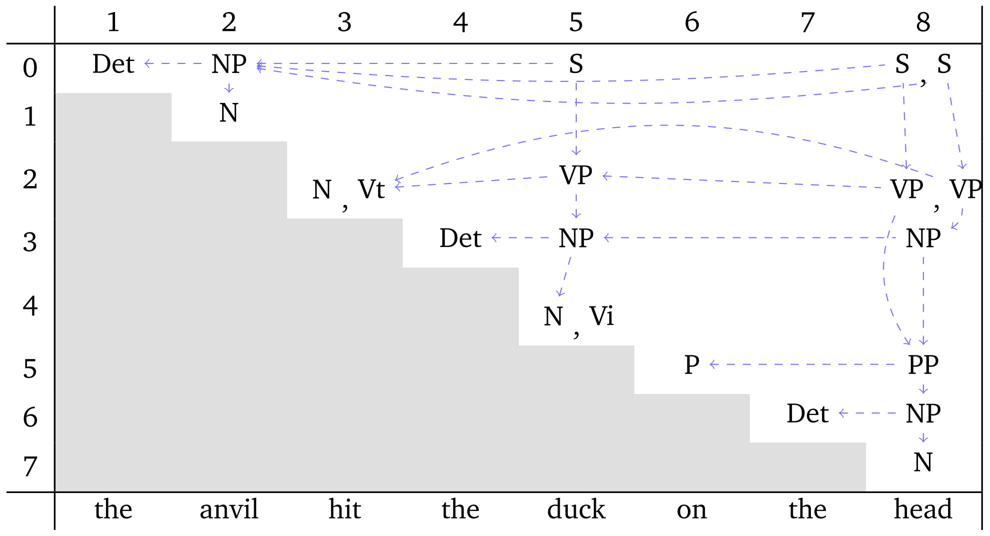
In order to fill in the remaining cells, we use the following algorithm: if cell (i, j) contains category B and cell (j, k) contains category C and $A \to BC$ is a rewrite rule of the grammar, then enter A in cell (i, k).

	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). 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

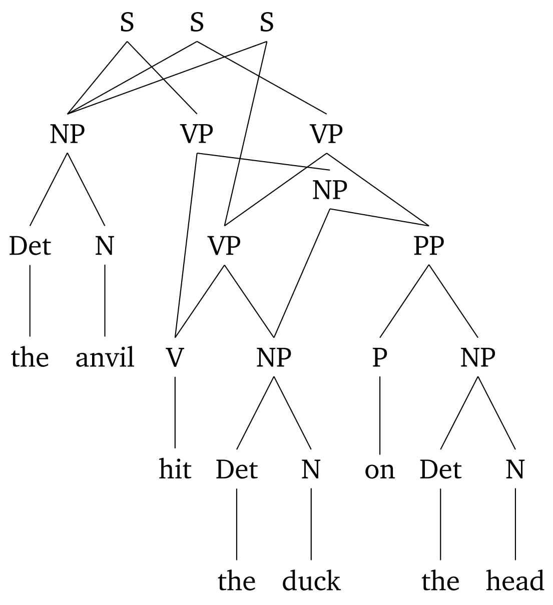


Figure 7.1: Directed acyclic graph representation of a CKY chart with backpointers

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, $(4, 5, \mathrm{N})$ and $(4, 5, \mathrm{V})$. 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 $(\mathrm{VP}, 8)$ 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 terminal 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. 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 one.

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

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

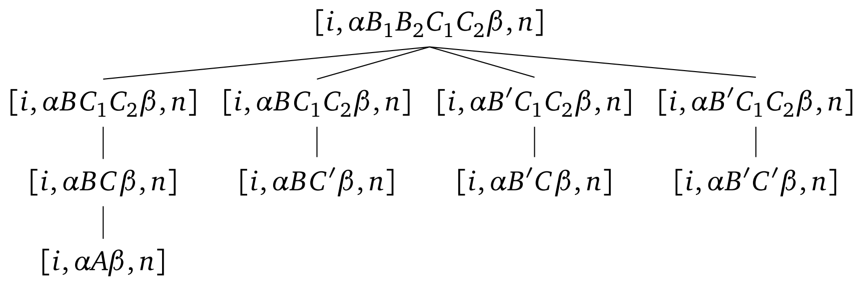

$[i, \alpha B_1 B_2 C_1 C_2 \beta, n]$
$[i, \alpha B C_1 C_2 \beta, n]$
$[i, \alpha B C_1 C_2 \beta, n]$
$[i, \alpha B' C_1 C_2 \beta, n]$
$[i, \alpha B' C_1 C_2 \beta, n]$
$[i, \alpha B C \beta, n]$
$[i, \alpha B C' \beta, n]$
$[i, \alpha B' C \beta, n]$
$[i, \alpha B' C' \beta, n]$
$[i, \alpha A \beta, n]$

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

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. In the standard bottom-up parser, it might take us much longer. As can be seen in Fig. 7.2, it may require 9 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. $\odot$

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. $\odot$

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 $$\frac{[i,B_1,j_1] \qquad [j_1,B_2,j_2] \qquad \cdots \qquad [j_n,B_{n+1},k]}{[i,A,k]}\; A \rightarrow B_1B_2\cdots B_{n+1} \in R$$

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

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.