Lecture 14

Syntactic Parsing

In the previous section we concluded that the choice for or against a specific type of tree language cannot be made a vacuum. Since regular tree languages can always be translated into strictly local ones, and *vice versa*, we cannot proclaim one of them to be more cognitively real than the other without putting a very elaborate scaffolding of additional assumptions in place. At this point, we do not know enough about cognition to pick a specific set of assumptions over another, and odds are that in order to distinguish between regular and strictly local tree languages, we would need assumptions that are so specific and fine-grained that they could never be fully backed up via empirical evidence.

A more fruitful strategy is to accept this indeterminacy and make good use of it. This can take a negative form by no longer expanding time and resources on fruitless efforts to settle this issue, but also a positive one where we switch between these types of tree languages depending on which one serves our purposes best. Today we will see how strictly local tree languages provide a simple format for designing parsers.

1 Intersection Parsing

On a computational level (in Marr's sense), a parser is device that infers the tree structures that a grammar assigns to a given string. We can make this more precise via the function yd, which maps every tree to its string yield: a parser is device that takes a grammar G and a string $s \in \Sigma^*$ as input and computes $\text{yd}^{-1}(s)$ with respect to G. Formally, this amounts to computing $\text{yd}^{-1}(\{s\} \cap L^s(G))$.

Conceptualizing parsing in terms of intersection is very elegant as it reduces parsing to basic operations on languages that we are already familiar with. In particular, if one can construct a grammar G' whose string language is just s, and that assigns exactly the same trees to s as G, then $\text{yd}^{-1}(\{s\} \cap L^s(G))$ is just the tree language generated by G'. For CFG, such a G' is guaranteed to exist because singleton languages are regular, and CFLs are closed under intersection with regular languages.

Theorem 14.1. Let L_C and L_R be a context-free and a regular string language, respectively. Then $L_C \cap L_R$ is context-free. ⌟

Proof. Our proof relies on three insights:

- L_C is generated by some CFG $G := \{\Sigma_G, S, R\}$.
- L_R can be represented by a deterministic FSA $A := \{\Sigma_A, Q, q_0, F, \Delta\}$.

- The states of A can be incorporated into the alphabet of G, yielding a grammar G_A that executes both G and A simultaneously.

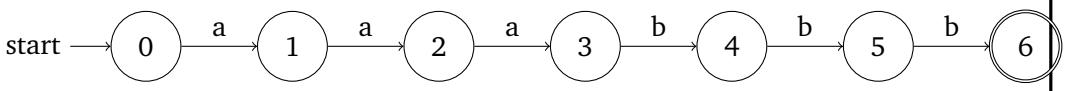
The construction proceeds by prefixing every non-terminal N with the state that A assigns to the leftmost symbol in its substring, and similarly suffixing N by the state of the rightmost symbol. For the sake of simplicity we assume that G is strictly binary branching and ε-free (which does not decrease generative capacity), but the proof is easily adapted to the more general case.

We first add a rule $S \rightarrow {}_{q_0}S_{q_f}$ to G_A for every q_f that is the final state of some accepting run of A over some $s \in L_R$. Given a rewrite rule $X \rightarrow Y\ Z$ and some other rule with ${}_pX_q$ on the righthand side, we add ${}_pX_q \rightarrow {}_pY_u\ {}_uZ_q$ to G_A, where u is some state such that for some run over a string spanned by X that starts with p and ends in q I) the last symbol of the substring spanned by Y ends in u, and II) the first symbol of the substring spanned by Z starts with u. We do the same if B or C are terminal symbols instead, except that they are not prefixed and suffixed with states.

Showing the correctness of the procedure is left as an exercise to the reader. □

Example 14.1

Let G be the CFG with the rewrite rule $S \rightarrow Aa|SB$, $A \rightarrow aa$, $B \rightarrow bbb$ and s the string $aaabbb$, which is recognized by the deterministic automaton A below.



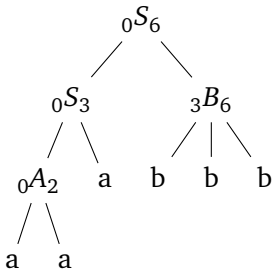
In order to determine what trees G assigns to s, we construct a refined CFG G_A according to the instructions above.

First, we add the rewrite rule $S \rightarrow {}_0S_6$, as 6 is the only final state that A assigns to $aaabbb$ and there are no other strings to consider. Since we now have a refined symbol ${}_0S_6$ we have to take care of all the rewrite rules with S on the left hand side. We add the rule ${}_0S_6 \rightarrow {}_0S_3\ {}_3B_6$ since B spans a substring bbb, which only occurs between the states 3 and 6 in A's run over s. We cannot find any valid state subscripts for A in ${}_0S_6 \rightarrow Aa$ since no string in $L(a)$ allows us to reach state 6 via an a-transition. Consequently, we cannot add a refined rule in this case and move on to the rules for ${}_0S_3$. In this case, the only valid refinement is ${}_0S_3 \rightarrow {}_0A_2a$.

The full set of rewrite rules is given below.

$$\begin{aligned} {}_0S_6 &\rightarrow {}_0S_3\ {}_3B_6 \\ {}_0S_3 &\rightarrow {}_0A_2a \\ {}_0A_2 &\rightarrow aa \\ {}_3B_6 &\rightarrow bbb \end{aligned}$$

It is easy to see that G_A generates only s and assigns it the same tree as G, *modulo* subscripts.



What properties must hold of a CFG so that it only assigns a finite number of trees to a string?

As long as the refined grammar assigns only a finite number of trees to s (which can be guaranteed via two simple restrictions), this set can be generated by simply applying all rewrite rules in all possible orders. So our intersection approach has indeed succeeded at reducing parsing to generation with a refined grammar.

The procedure is stated for CFGs, but of course we can extend it to regular tree languages. First we make the hidden alphabet part of the visible alphabet, which turns the refined strictly 2-local tree grammar into a CFG (we might have to change a few labels to establish a clear distinction between terminal and non-terminal symbols). This CFG is then refined, and once the set of trees for the input string has been determined, it is lifted back into a regular tree language via a projection that removes the subscripts and those parts of the label that were originally part of the hidden alphabet. For strictly k-local tree languages, we simply construct the corresponding strictly 2-local tree grammar and proceed as before. The fact that we can freely translate between these types of tree languages and use CFGs as a baseline for the refinement construction is what makes intersection parsing applicable across the board.

2 Top-Down Parsing

2.1 Definition

Elegant as intersection parsing may be from a mathematical perspective, it is not a realistic model of human sentence processing. Its major shortcoming is that the full input string must be known before one can even get started on the grammar refinement. Human parsing does not work like this, listeners begin inferring structures as soon as they hear the first word, and there is plenty of evidence that they even build structure before the corresponding part of the string has been encountered. These are two essential properties of human parsing: it is *incremental* and *predictive*.

We need a more procedural model in order to capture these properties. One of the simplest and best-known models is top-down parsing. 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 still very close to the idea of CFGs as top-down generators.

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

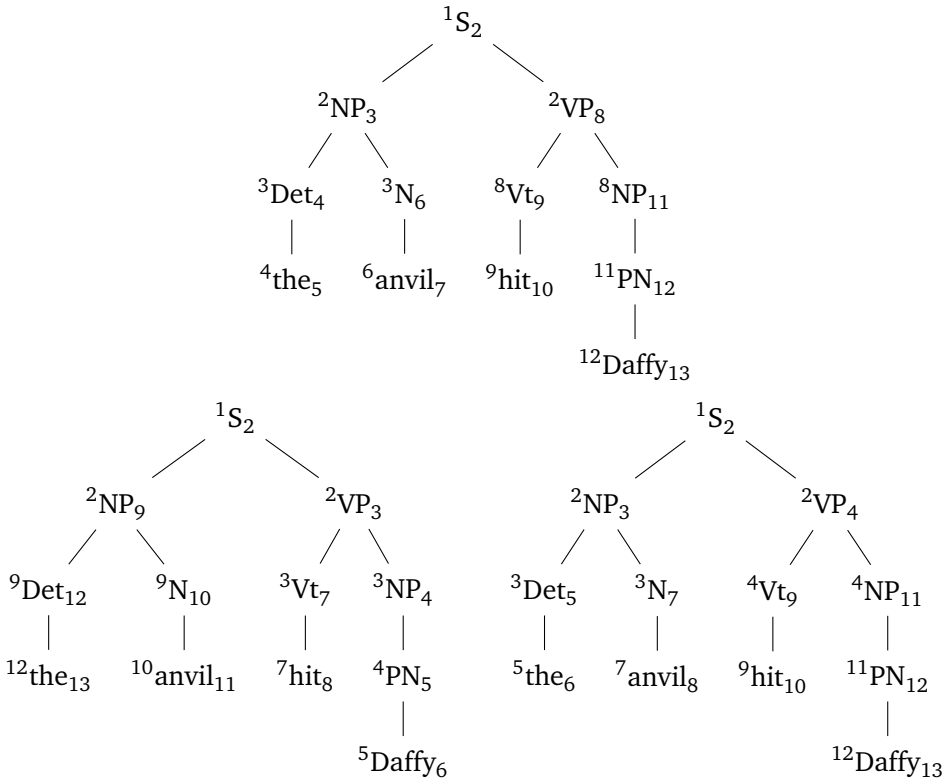
depth-first rewrite some symbol that was introduced during the previous rewriting step

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.



The prototypical top-down parser operates depth-first and left-to-right, and is also known as a *recursive descent parser*. It's behavior can be stated very succinctly 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 made it possible to abstract away from implementation-heavy concepts like data structures and memory management in order to state the basic parsing procedure as clearly as possible. It is also the foundation of *semiring parsing* (Goodman 1999), which provides an abstract perspective that unifies various parsers similar to how our monoid perspective in chapter 6 unified a variety of scanners (in fact, a semiring is an algebraic object that combines two monoids in a particular fashion).

The logical inference rules take the form of *parse items* $[i, \beta]$, where

- $\beta \in \Sigma^*$ is a string of terminal and/or non-terminal symbols, and
- i is a position in the string.

The parser always starts with the *axiom* $[0, S]$, which represents the assumption that the input string can be obtained by rewriting the start symbol S. The parser accepts a string iff it can use its inference rules from the axiom to the *goal* item $[n,]$, where n is the end of the input string. The inference rules are as follows:

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

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

The scan rule removes a terminal symbol from a parse item if it matches the symbol in the input at the corresponding position. The predict rule expands non-terminal nodes according to the grammar.

Example 14.2 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]	axiom
[0,NP VP]	predict(1)
[0,Det N VP]	predict(3)
[0,the N VP]	predict(6)
[1,N VP]	scan
[1,truck VP]	predict(7)
[2,VP]	scan
[2,Vt NP]	predict(5)
[2,hit NP]	predict(10)
[3,NP]	scan
[3,PN]	predict(2)
[3,Daffy]	predict(8)
[4,]	scan

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

Without some kind of additional memory, the parser does not actually keep track of the tree structure and merely acts as a recognizer. But it is fairly simple to assemble the correct tree from the table above as all the needed information is contained in the application of the predict steps.

2.2 Psycholinguistic Predictions

The recursive descent parser is clearly incremental, as it can start parsing before we have seen a single input symbol. This also shows that it is predictive. In fact, there is no such thing as an non-predictive top-down parser seeing how the parser must build a subtree before it can even start scanning the leafs of the subtree. But with a few ancillary assumptions, the recursive descent parser can also explain certain

shortcomings of human sentence processing such as garden-path effects and center-embedding. Let us take a closer look at the former.

Garden path effects were one of the first phenomena to be discussed in the psycholinguistic literature. They 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:

(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.

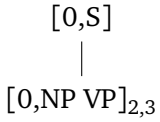
(2) *Lexical ambiguity*
   a. The old train the young.
   b. The old man the boat.
   c. Until the police arrest the drug dealers control the street.
   d. The dog that I had really loved bones.
   e. The man who hunts ducks out on weekends.

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. Let's try to make Frazier's account precise.

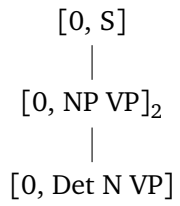
First, we will use prefix trees to represent the parse history, i.e. a record of how the parser built a parse table, rejected it at some point, and moved on to the next one.

Example 14.3 Parse Tables as Trees

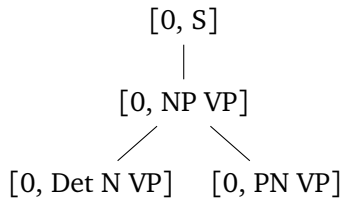
Suppose we are once more using the grammar above to parse *the anvil hit Daffy* with a recursive descent parser. Our parse table starts with [0,S] as usual, from which we can only predict [0,NP VP]. This can be represented as the tree below.



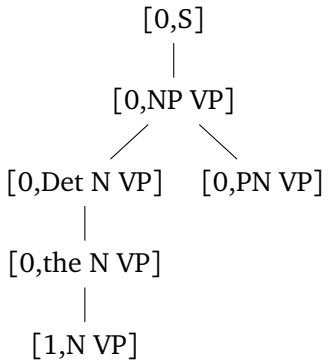
The node [0, NP VP] has subscripts 2 and 3 to indicate that we can use rules 2 and 3 of our CFG to predict new parse items. Suppose the parser uses predict(3) to create the item [0, Det N VP]. Then this item is added as a daughter of [0, NP VP] to the previous tree, and the subscript 3 is also removed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



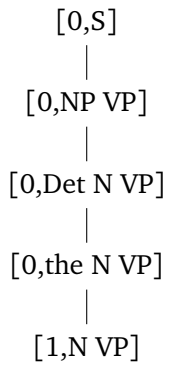
The next step of the parser now depends on how distinct parses are prioritized. We can either expand the table that ends in [0, Det N VP], or go back to the one ending in [0, NP VP] and apply predict(2). Suppose we do the latter, so that [0, PN VP] is added as the second daughter of [0, NP VP], which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



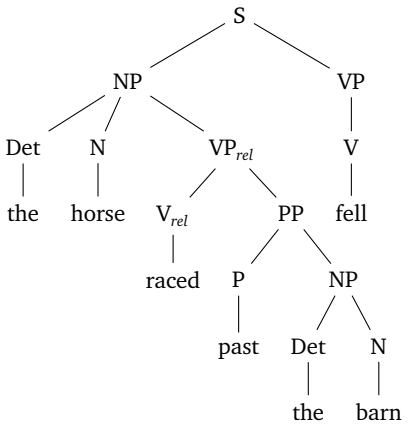
What makes the prefix tree representation of parse tables appealing is that the construction and prioritization of parse tables can be reduced to strategies for tree building. The kind of serial parsing envisioned by Frazier corresponds to a depth-first strategy where the parser always builds a single 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. With such a strategy, garden path effects are readily explained by the amount of attempts it takes to find a successful parse.

Example 14.4 Backtracking in *the horse raced past the barn fell*

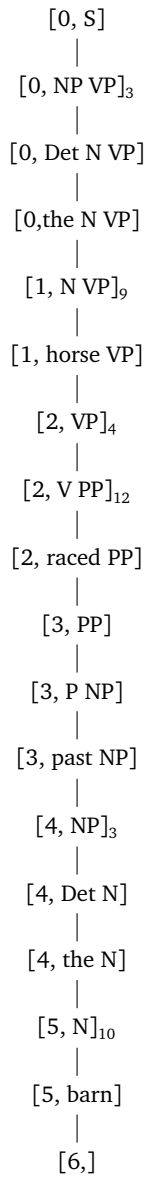
Assume we have the following (massively simplified) grammar:

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

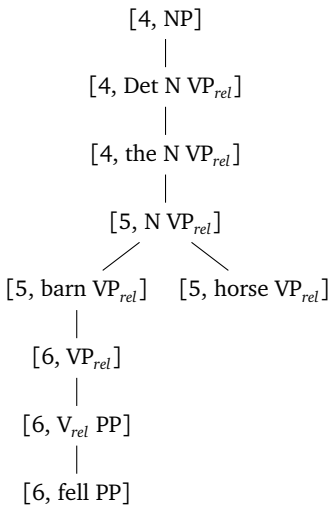
Here's the resulting phrase structure tree for our garden path sentence.



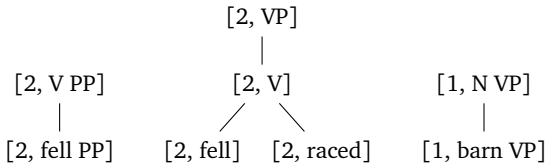
If the parser prefers NP → Det N over NP → Det N VP_{rel} and operates in a recursive descent fashion, the construction of the first parse table results in the prefix tree below.



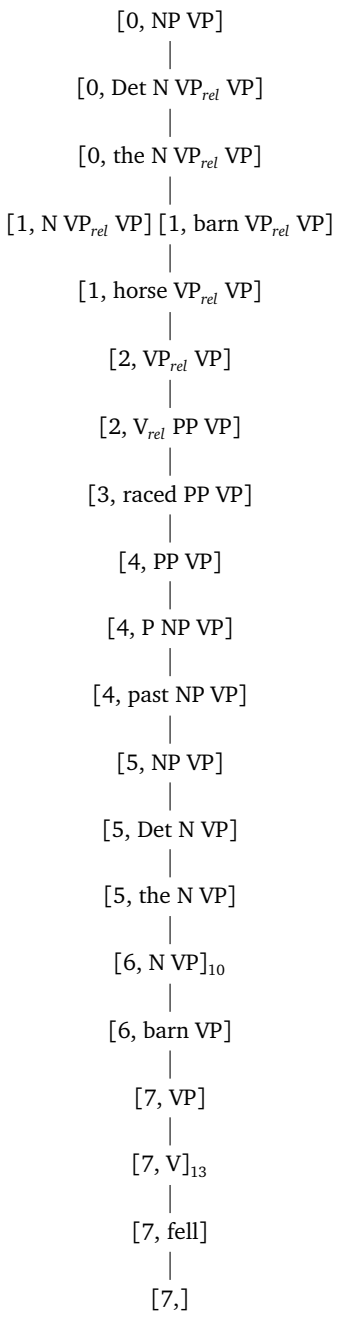
Since this parse does not succeed, the parser needs to backtrack. The closest choice point is $[5,N]_{10}$, which obviously does not fix the problem of integrating *fell* into the structure, as can be verified after a single scan step. The next choice point is $[4,NP]_3$. Here the parser still has the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



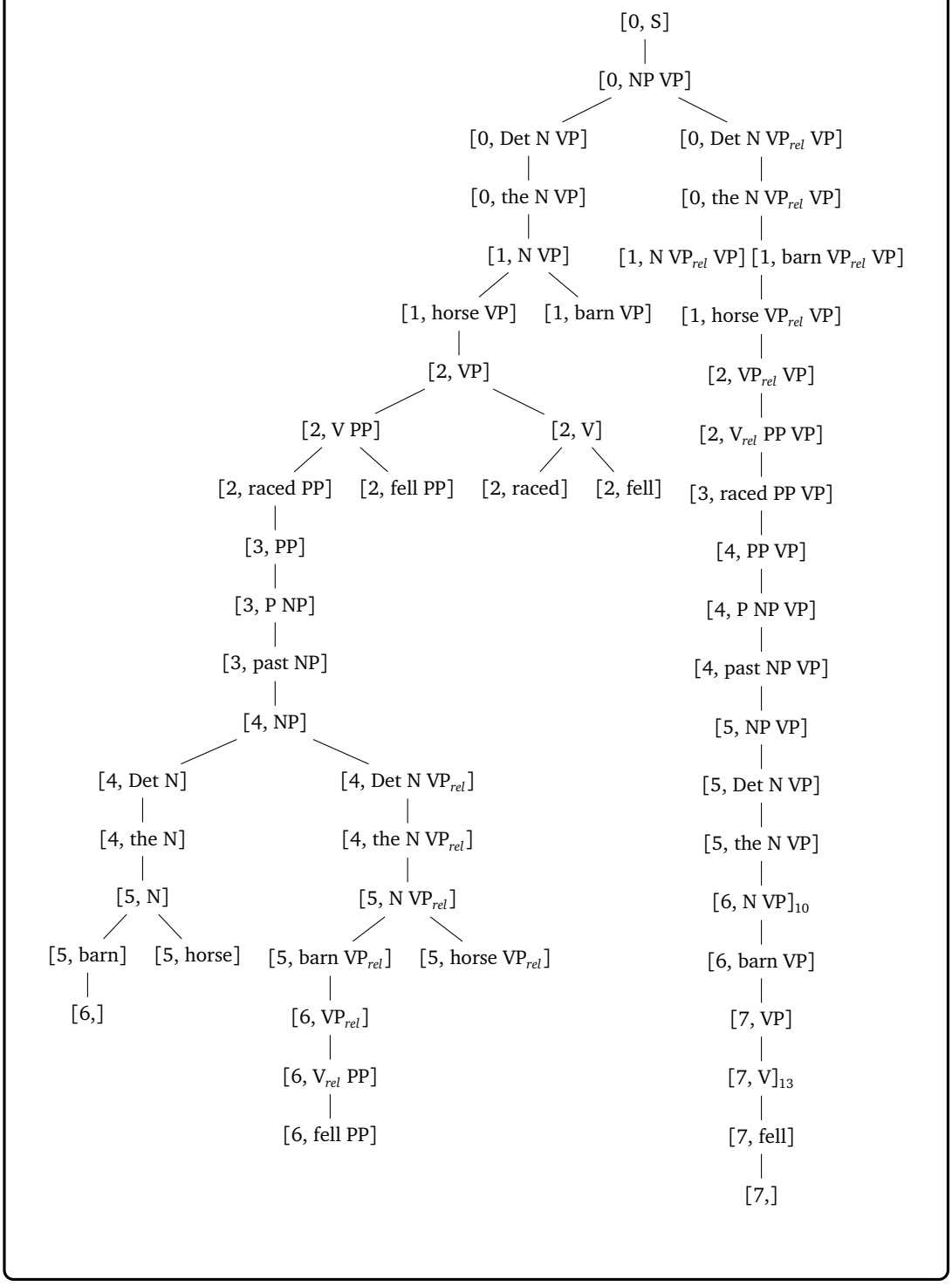
Since this venue didn't yield a successful parse either, the parse backtracks to [2,V PP]$_{12}$, and after this fails, to [2,VP]$_4$. Once again it is not successful, and the same holds once it expands [1,N VP].



Only if the parser backtracks all the way to [0, 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.


[0, S]
[0, NP VP]
[0, Det N VP]
[0, Det N VP$_{rel}$ VP]
[0, the N VP]
[0, the N VP$_{rel}$ VP]
[1, N VP]
[1, N VP$_{rel}$ VP]
[1, barn VP$_{rel}$ VP]
[1, horse VP]
[1, barn VP]
[1, horse VP$_{rel}$ VP]
[2, VP]
[2, VP$_{rel}$ VP]
[2, V PP]
[2, V]
[2, V$_{rel}$ PP VP]
[2, raced PP]
[2, fell PP]
[2, raced]
[2, fell]
[3, raced PP VP]
[3, PP]
[4, PP VP]
[3, P NP]
[4, P NP VP]
[3, past NP]
[4, past NP VP]
[4, NP]
[5, NP VP]
[4, Det N]
[4, Det N VP$_{rel}$]
[5, Det N VP]
[4, the N]
[4, the N VP$_{rel}$]
[5, the N VP]
[5, N]
[5, N VP$_{rel}$]
[6, N VP]$_{10}$
[5, barn]
[5, horse]
[5, barn VP$_{rel}$]
[5, horse VP$_{rel}$]
[6, barn VP]
[6,]
[6, VP$_{rel}$]
[7, VP]
[6, V$_{rel}$ PP]
[7, V]$_{13}$
[6, fell PP]
[7, fell]
[7,]