Life after PCFGs?

In these notes and the next we summarize the main problems with CFGs and PCFGs for natural language processing, and then proceed to see how we can try to fix at least some of these problems.

1 Problems with CFGs

In previous notes we illustrated that CFGs fail in several ways, being unable to represent in a concise way (or even any way), some of the “natural” properties of natural language structures. Further, PCFGs don’t seem to solve the problems associated with making the right kinds of choices to fix ambiguity. Let’s review these issues and see what attempts have been made to solve them. First, we look at problems that arise even with ordinary CFGs.

1.1 CFG problems

While there are many representational problems with CFGs, many of these seem to boil down into four categories:

1. CFGs do not naturally encode the agreement between difference phrases in natural language, leading to duplication of rules

2. CFGs do not naturally encode the displacement of phrases from one part of a sentence to another

3. CFGs do not naturally encode the property that phrases are based on items like verbs, prepositions, adjectives, or nouns.

4. CFGs do not naturally encode the structures associated with conjunctions and coordination.

Let’s look at these problems one at a time.

2 CFGs and features

First, as CGW probably showed you, it seems that CFGs do badly when one has to write a set of rules where there is agreement between, for example, the Subject and the Verb (or between a Determiner and a Noun). This so in English, even though other languages exhibit an ever greater degree of agreement.

The dogs ARE happy
The dog is happy
?The dogs IS happy
THES$ dogs are happy
THIS dog is happy
?THIS dogs are happy
In languages like German, Italian, Spanish, Czech, etc., agreement can be much more systematic. For example, while English has only a residue of agreement, in German, there are three genders (masculine, feminine, neuter) and two numbers (singular, plural) but they combine only into four classes (masculine singular, feminine singular, neuter singular, common plural). The formation of a noun plural may be dependent on the gender of the word, but all other words – determiners, pronouns, adjectives – referring to a plural noun are not affected by its gender. Note that gender is not ‘biological’; it is a purely syntactic phenomenon. English also has a case feature, a historical remnant of Latin: we have, *I saw her* (where *her* is the object of the verb, also called *objective case*); as opposed to, *She saw Mary* (where *she* is the subject of the verb, or *nominative case*).

So what happens with features? Take Subject-Verb agreement. If one uses vanilla CFGs, then one has to laboriously write out 2 or more rules that are near duplicates of each other, one rule for the case where Nouns and Verbs are singular, one where they are plural, one if the ‘person’ feature is so-called “third person” (e.g., she, him). So for example, we have *she GOES to the story* and *we GO to the store*, and so forth. We need at least rules like these to distinguish these possibilities. Note that the nonterminal names are used to distinguish between different ‘states’ in terms of features:

\[
S \rightarrow \text{NP\text{\_\text{singular}}} \text{ VP\text{\_\text{singular}}}
\]
\[
S \rightarrow \text{NP\text{\_\text{plural}}} \text{ VP\text{\_\text{plural}}}
\]

As you have seen, the number of such rules explodes combinatorially when we have to include each new feature. This is bad for two reasons.

1. Since parsing time depends on grammar size, this larger grammar size will slow down parsing (by a factor of the grammar size squared in the worst case).

2. By writing down rules that are nearly redundant, we have failed to capture the right generalization: what we want to say is that the NP Subject agrees with the VP (and Verb) no matter what the features are. Put another way, the grammar would be the same size if we replaced the second of the two rules above with a rule that said the plural Subject NP *does not* agree with the Verb. To capture the right generalization that feature agreement holds in this case (and so blocks a funny rule like this), we want to factor apart the basic phrase structure component (\(S \rightarrow \text{NP VP}\)) from the feature component – a good, modular solution.

So, what we would like to do is to be able to write just a single grammar rule in the place of two (or more), capturing this regularity. For example, something like this, where we have placed in square brackets a feature, value pair, here the NUMBER feature, abbreviated NUM, with its value indicated as a variable by placing a question mark in front of it. The idea is that the value for the number feature is copied up from whatever the number value is for the word that forms the NP, and similarly for the Verb. For example, if the Noun forming the NP is *dogs*, then the feature value for NBR would be, say, \(|\text{PLU}|\), standing for “plural”; similarly for the verb *are*. By making the variable value \(?X\) the same for both the NP and the VP, we have imposed the constraint that whatever those values turn out to be, they must be the same.

\[
S \rightarrow \text{NP[\text{NUM }?X]} \text{ VP[\text{NUM }?X]}
\]

Of course, other languages (and also English) will have different feature-value combinations. In English, agreement includes subject-verb agreement on person, number, gender, animacy, abstractness, and quantity. (For example, you ran across the feature of “quantity” when doing the CGW exercise: some nouns are count nouns, that is, objects that can be parcelled out and counted, as with *book*. Other nouns are mass nouns, that are measured in terms of quantities of “stuff” like *water*, so one can say *Water is refreshing* but not *Book is refreshing.*) Case marking is another form of agreement that surfaces both in word forms
and syntactically. Noun forms such as epithets, pronouns, or anaphora – words that refer to earlier phrases in or across sentences – may be required to agree or disagree with other noun forms in person, number, gender, and so forth. For instance: Reagan, the fool, believed he could appoint justices himself. Typically such agreement can occur over an unbounded number of words or phrases. This occurs in all languages, along with the phenomenon of ambiguity. Syntactic homonyms are common, like block as a noun or a verb.

Descriptively adequate theories of human language must therefore describe these two phenomena of agreement and ambiguity, and all major linguistic theories do so, using three devices:

1. Distinctive features to represent the dimensions of agreement
2. An agreement enforcement mechanism
3. Provision for lexical and structural (syntactic) ambiguity

While different theories work out the details of these three devices in different ways, one can abstract away from these variations in order to model just agreement.

To formalize this, we can extend the notion of CFGs to what we can call Agreement Grammars or AGs. We do this by adding nonterminals that are sets of features and imposing an agreement condition on the CFG derivation relation. (We should note that as with CFGs, AGs are still too simple to model natural languages, and they also can describe infinitely many non-natural languages. However, they are a step in the right direction.)

We first recall the definition of a standard CFG we have presented earlier, along with the notion of derivation in a CFG. A CFG $G$ is a 4-tuple:

$$G = (N, V, R, S)$$

where $N$ is a finite set of nonterminal symbols or phrases; $V$ is a finite set of terminal symbols; $S$ is a distinguished start symbol; and $R$ is a finite set of rules of the form $A \rightarrow \gamma$, where $A \in N$ and $\gamma \in (N \cup V)^*$. If $R$ contains a rule $A \rightarrow \gamma$, then for any $\alpha, \beta \in (N \cup V)^*$, we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$ and say that $\alpha A \beta$ derives $\alpha \gamma \beta$ with respect to $G$. We let $\Rightarrow^*$ be the reflexive, transitive closure of $\Rightarrow$ (that is, the application of the derives relation 0 or more times, until no more applications are possible). The language $L(G)$ generated by a CFG $G$ is the set of all terminal strings that can be derived from the start symbol with respect to $G$, i.e.,

$$L(G) = \{s : s \in V^* \& S \Rightarrow^* s\}$$

We now extend CFGs to obtain agreement grammars by adding nonterminals that are sets of features and imposing an agreement condition on the derivation relation.

A feature is defined as a [feature-name, feature-value] pair. For example, [Per 1] is a possible feature, designating “first person.” Some features may be designated agreement features, required to match other features, as we describe below. For instance, an AG nonterminal labeling the first person pronoun $I$ (and so a Noun) could be written this way: it is of the category N, it is not plural, and it is first person: \{[CAT N], [PLU -], [Per 1]\}. The singular verb sleeps could be labeled with the AG nonterminal features \{[CAT V], [PLU -], [PER 3]\}.

More formally, we define the set of nonterminals in an AG in the following way. The set of nonterminals in an agreement grammar is characterized by a specification triple $< F, A, p >$, where $F$ is a finite set of feature names and $A$ is a finite set of feature values. $\rho$ is a function from feature names to permissible feature values, that is, $\rho : F \rightarrow 2^A$ (the possible subsets of feature values associated with categories). This triple specifies a finite set of nonterminals $N$, where a nonterminal may also be thought of as a partial function from feature-names to feature-values:

$$N = \{C \in A^{(F)} : \forall f \in \text{DOM}(C)[C(f) \in \rho(f)]\}$$

Here, $Y^{(X)}$ denotes the set of all partial functions from $X$ to $Y$. $\text{DOM}(C)$ is the domain of $C$, that is, the set $\{x : \exists y[(x, y) \in C]\}$. We say that a category $C'$ extends a category $C$ (written $C' \supseteq C$) if and
only if \( \forall f \in \text{DOM}(C), |C'(f) = C(f)| \), that is, \( C' \) is a superset of \( C \). For example, the feature category \([\text{PER} 1], [\text{NUM} 1]\) extends the category \([\text{PER} 1]\) but \([\text{PER} 2], [\text{NUM} 1]\) does not because of the mis-matched feature value for \( \text{PER} \).

As a concrete example to illustrate all this terminology, suppose that we have the set \( F \) of 3 feature names, \( \text{CAT}, \text{PLU}, \text{PER} \) (phrase category, plurality, and person). We further have that the possible phrase category names are \( S, \text{VP}, \text{NP}, \text{V}, \text{N} \); that the possible values for the \( \text{PER} \) (person) feature are 1, 2, 3, corresponding to 1st person, 2nd person, and 3rd person; and the possible values for the feature \( \text{PLU} \) (for plural), are either + (meaning that something is plural), or − (meaning that it is not plural, i.e., that it is singular). Then the function \( \rho \) may be defined as follows.

\[
\rho(\text{CAT}) = \{ S, \text{VP}, \text{NP}, \text{V}, \text{N} \} \\
\rho(\text{PER}) = \{ 1, 2, 3 \} \\
\rho(\text{PLU}) = \{ +, - \}
\]

Using this feature machinery, we could encode the plural noun \textit{guys} as having the features: \([\text{CAT N}], [\text{PLU} +] \). (So \textit{guys} is not marked for the person feature.) As a verb, \textit{sleep} could have the features, \([\text{CAT V}], [\text{PLU} +] \) or the alternative feature set where it is not plural, \([\text{CAT V}], [\text{PLU} -], [\text{PER} 1] \).

**Definition.** A feature grammar \((AG)\) is a 5-tuple:

\[ G = (\langle F, A, \rho \rangle, V, F_A, R, S) \]

whose first element specifies a finite set \( N \) of syntactic categories (now feature based); \( V \) is a finite set of terminal symbols; \( F_A \) is a finite set of agreement feature names, \( F_A \subseteq F \); \( S \) is a distinguished start symbol; and \( R \) is a finite set of the usual context-free rules, each rule taking one of the forms:

1. \( C \rightarrow w \), where \( C \in N \) and \( w \in V \);
2. \( C_0 \rightarrow C_1 \ldots C_n \), where each \( C_i \in N \).

In this formulation at least, no so-called null or epsilon productions are permitted: each rule must have at least one non-null element on its right-hand side.

To complete the definition, we must modify the derives relation to incorporate agreement. We say that a production \( C_0' \rightarrow C_1' \ldots C_n' \) extends a rule \( C_0 \rightarrow C_1 \ldots C_n \) if and only if \( C_i' \) extends \( C_i \) for every \( i \), and further, that the mother’s agreement features appear on every daughter:

1. \( \forall i, 0 \leq i \leq n, [C_i' \supseteq C_i] \) and
2. \( \forall f \in (\text{DOM}(C_0') \cap F_A), \forall i, 1 \leq i \leq n, [(f \in \text{DOM}(C_i')) \& (C_i'(f) = C_i(f))] \)

This last condition, the agreement convention, ensures that all agreement features on the mother are also found on all the daughters.

We may now define the language generated by an agreement grammar. If \( R \) contains a rule \( A \rightarrow \gamma \) with an extension \( A' \rightarrow \gamma' \), then for any \( \alpha, \beta \in (N \cup V)^* \), we write \( \alpha A' \beta \Rightarrow \alpha \gamma' \beta \). Let \( \Rightarrow \) be the reflexive, transitive closure of \( \Rightarrow \) in the given \( AG \) grammar \( G \). The language \( L(G) \) generated by \( G \) contains all the terminal strings that can be derived from any extension of the start category:

\[ L(G) = \{ s : s \in V^* \text{ and } \exists S', S' \supseteq S, \& S' \Rightarrow^* s \} \]

**Example continued.** To continue with our example, besides the feature names and values and function we gave earlier, let us specify the terminal vocabulary of the example agreement grammar as:

\[ V = \{ I, \text{guys}, \text{John}, \text{sleep}, \text{sleeps} \} \]
The grammar contains the following 9 rules:

\[
\begin{align*}
\text{CAT } S & \rightarrow \text{CAT } NP \mid \text{CAT } VP \\
\text{CAT } VP & \rightarrow \text{CAT } V \\
\text{CAT } NP & \rightarrow \text{CAT } N \\
\text{CAT } NP, [\text{PLU} -], [\text{PER } 1] & \rightarrow I \\
\text{CAT } N, [\text{PLU} +] & \rightarrow \text{guys} \\
\text{CAT } NP, [\text{PLU} -], [\text{PER } 3] & \rightarrow \text{John} \\
\text{CAT } V, [\text{PLU} +] & \rightarrow \text{sleep} \\
\text{CAT } V, [\text{PLU} -], [\text{PER } 1] & \rightarrow \text{sleep} \\
\text{CAT } V, [\text{PLU} -], [\text{PER } 3] & \rightarrow \text{sleeps}
\end{align*}
\]

This example grammar generates exactly the following sentences.

(a) I sleep \(= [\text{CAT } S], [\text{PER } 1], [\text{PLU} -1]\)
(b) guys sleep \(= [\text{CAT } S], [\text{PLU} +1]\)
(c) John sleeps \(= [\text{CAT } S], [\text{PER } 3], [\text{PLU} -1]\)
(d) I sleep \(= [\text{CAT } S], [\text{PER } 1], [\text{PLU} -1]\)

In practice, rather than generating sentences “top down” as above, often feature grammars are first parsed using just the “vanilla” CFG that underlies them, and then passing variable values up the parse tree to see if the features can match where the agreement convention can be enforced. In this case, we can use feature names with variable values, e.g., \([\text{PER } ?X]\) to indicate a value that must be filled in from some lexical entry. Then, the agreement convention will force the values of a single variable value like \(?X\) appearing within a single rule to all have the same value, i.e., to “agree.”

Let’s see how this works out in the context of a simple, concrete example. We begin with a “vanilla” CFG:

\[
\begin{align*}
S & \rightarrow \text{NP } \text{VP} \\
\text{NP} & \rightarrow \text{Name} \\
\text{NP} & \rightarrow \text{DT } \text{N} \\
\text{VP} & \rightarrow \text{VB } \text{NP} \\
\text{VB} & \rightarrow \text{solve} \\
\text{VB} & \rightarrow \text{solves} \\
\text{DT} & \rightarrow \text{this} \\
\text{DT} & \rightarrow \text{these} \\
\text{N} & \rightarrow \text{crime} \\
\text{N} & \rightarrow \text{crimes} \\
\text{Name} & \rightarrow \text{John}
\end{align*}
\]

As stated, the grammar will wind up generating sentences where the VP and the NP agree (e.g., John solves the case) as well as disagree (John solves the case), along with sentences where the determiner and Noun agree (John solves this case) and disagree (John solves these case). Also note that if the Subject NP is plural, as in these cases, then it must agree with a plural verb form: These cases solve... – though we have not elaborated this grammar enough to bring out such examples.

Let us now rewrite the grammar using the variable-style feature value format, such that we do not need to duplicate the rules in order to enforce feature agreement. Here is one way to do it, just checking for the feature plural:
When a sentence such as John solves these crimes is parsed, we can first just use the original rules, which will leave us with a set of features attached to the parse tree nodes with unvalued variable names. We can then pass up values from the words at the bottom. The singular feature on John will make the Subject NP have the value [PLU -]. The feature value on solves will make the V, and so the VP, have the feature value [PLU -]. Finally, the S rule will ensure that these two feature values are the same, because the variable ?x on both the NP and the VP must be the same (and this becomes the feature value for the whole S). Similarly, the Object NP will get the feature value [PLU +] because of these and crimes, which must also agree.

If we have more than one agree feature, say, the person feature (1st, 2nd, or third), then one can simply attach these features to rules as a set:

\[
S [PLU ?x], [PER ?y] \rightarrow NP [PLU ?x], [PER ?y] VP [PLU ?x], [PER ?y]
\]

As an abbreviation, we might write down the person, number, and gender features as a kind of wholesale agreement feature, AGR, associated with this first rule:

\[
S [AGR ?x] \rightarrow NP [AGR ?x] VP [AGR ?x]
\]

That is, the AGR feature is itself nested and can act like a macro that expands out into two or more actual feature values. This is a very satisfying abbreviation. However, here one must be a bit careful, as the next paragraph explains, because the cost of doing this kind of computation can become expensive, and unnecessary.

Note that some linguistic theories encourage a more elaborated form of feature than this. So far, what we have described is a “flat” set of feature-value pairs that can be associated with any nonterminal node. It is an open question whether one needs more complex features than this. If one opts for a more complex feature structure, one that is, for example, hierarchical, or even cyclical, then the corresponding representation is the same as a directed acyclic graph (DAG), a feature structure. If this is so, then checking whether two feature structures agree or not is computationally more complex than simply checking whether two phrases agree in terms of person or number. For “flat” feature checking, we only have to move through two (or more) linear arrays of feature-value pairs. However, since feature structures are DAGs, checking whether two DAGs “agree” or not amounts to what is called feature unification. One can envision unification as though one were laying one DAG on top of the other, and making sure that where they overlap, the DAGs have the same values at graph edges and at the two graphs’ leaves. In the worst case, this problem can be exponential-time complete, which is very hard indeed; though many useful subcases can still be solved in polynomial time. A
complete discussion of linguistic theories that adopt such full-fledged unification is beyond the scope of these
notes. Such grammars go by the name of unification grammar; lexical-functional grammar (LFG), and head-
driven phrase structure grammar. Please see me if you would like additional references on the complexity of
these theories, or, you can consult the book by Berwick, Barton, and Ristad, Computational Complexity and
Natural Language (MIT Press). In any case, for our purposes, “flat” feature-value agreement will suffice.

Here is a worked-out example of the “slash” nonterminal analysis, using a very simple feature grammar
from NLTK. Note the minor notational differences from the format used earlier. The symbols TV and
IV denote transitive and intransitive verbs. The way that feature checking is done is on-line; that is, we
percolate features up from the lexical entries to nonterminals, as soon as we find in a bottom-up way (like
the CKY algorithm) a complete phrase (nonterminal) of some type that needs features to be filled in. A
good question to ask is when we should halt parsing, if the program finds a feature clash. For example, the
dogs likes children is not well-formed, because the plural feature of dogs does not agree with the singular
feature on the verb likes (it should be like). A grammar-checker would continue, and then inform us about
the mistake.

% start S
S -> NP[NUM=?n] VP[NUM=?n]
# NP expansion productions
NP[NUM=?n] -> N[NUM=?n]
NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]
# VP expansion productions

Det[NUM=sg] -> ’this’ | ’every’
Det[NUM=pl] -> ’these’ | ’all’
Det -> ’the’ | ’some’ | ’several’
PropN[NUM=sg] -> ’Kim’ | ’Jody’
N[NUM=sg] -> ’dog’ | ’girl’ | ’car’ | ’child’
Figure 2: The feature-value parse tree for *Kim likes children*.

If we now parse a simple example sentence like *Kim likes children*, we get the parse tree shown in Figure 2.

We can examine in detail the way this successful parse proceeds, and how the features are percolated up the tree. That gives us the trace below. The operations “Predict - Combine” refer to the bottom-up construction of each subtree, e.g., the first three rules show how the singular feature on *Kim* are passed up to the PropN (proper name) nonterminal, then to the NP, then to the S level, as the value ?n sg (i.e., a n(umber) feature with the value sg, singular. The parser constructs the rest of the sentence bottom up, both VP and Object NP, it percolates the plural feature on the verb up to the VP (the next to last operation), to finally discover that this value agrees with the singular feature on the Subject NP:

Feature Bottom Up Predict Combine Rule:
| [-] . . | [0:1] PropN[NUM='sg'] -> 'Kim' *
Feature Bottom Up Predict Combine Rule:
| [-] . . | [0:1] NP[NUM='sg'] -> PropN[NUM='sg'] *
Feature Bottom Up Predict Combine Rule:
Feature Bottom Up Predict Combine Rule:
| . [----- . | [1:2] TV[NUM='sg', TENSE='pres'] -> 'likes' *
Feature Bottom Up Predict Combine Rule:
Feature Bottom Up Predict Combine Rule:
| . [----- ] [2:3] N[NUM='pl'] -> 'children' *
Feature Bottom Up Predict Combine Rule:
| . [----- ] [2:3] NP[NUM='pl'] -> N[NUM='pl'] *
Feature Bottom Up Predict Combine Rule:
Feature Single Edge Fundamental Rule:
| . [---------] [1:3] VP[NUM='sg', TENSE='pres'] -> TV[NUM='sg', TENSE='pres'] NP[] *
Feature Single Edge Fundamental Rule:
| | [0:3] S[] -> NP[NUM='sg'] VP[NUM='sg'] *
Case | Masc | Fem | Neut | Plural
--- | --- | --- | --- | ---
Nom | der | die | das | die
Gen | des | der | des | der
Dat | dem | der | dem | den
Acc | den | die | das | die

Table 1: Morphological Paradigm for the German definite Article

We can compare this successful parse with the trace of an unsuccessful parse on, *the dogs likes children*. Note when the mis-match in the Subject-Verb features is discovered: after the VP has been assembled, the last line in the trace, the VP has the feature for singular, sg. Now the rule to combine the Subject NP and the VP fails (it is not listed in the trace), because of the feature conflict between the plural feature for the NP *the dogs* and the singular feature for the VP *likes*. In the current system used here, nothing is done after this point, so no parse tree is output, but one could imagine taking other action.

As one final flourish to illustrate how important features can be when one turns to a morphologically rich language, such as German. German has a very rich agreement system, where definite articles vary with Case, Number, and Gender, as shown in Table 2.

Subjects in German take the nominative case, and most verbs govern their objects in the accusative case. However, there are exceptions like *helfen* that govern the dative case:
The feature *objcase* is used to specify the case that a verb governs on its object. Thus our grammar will now have a rich feature set. Here is the beginning of the rule set for Determiners:

```
# Singular determiners
# masc
Det[CASE=nom, AGR=[GND=masc, PER=3, NUM=sg]] -> 'der'
Det[CASE=dat, AGR=[GND=masc, PER=3, NUM=sg]] -> 'dem'
Det[CASE=acc, AGR=[GND=masc, PER=3, NUM=sg]] -> 'den'
...
# fem
Det[CASE=nom, AGR=[GND=fem, PER=3, NUM=sg]] -> 'die'
Det[CASE=dat, AGR=[GND=fem, PER=3, NUM=sg]] -> 'der'
Det[CASE=acc, AGR=[GND=fem, PER=3, NUM=sg]] -> 'die'
# Plural determiners
Det[CASE=nom, AGR=[PER=3, NUM=pl]] -> 'die'
...
N[AGR=[GND=fem, PER=3, NUM=sg]] -> 'Katze'
N[AGR=[GND=fem, PER=3, NUM=pl]] -> 'Katzen'
```

Similarly, the system for NPs and verbs must include a case feature (here as before TV stands for a “transitive verb”):

```
S -> NP[CASE=nom, AGR=?a] VP[AGR=?a]
VP[AGR=?a] -> IV[AGR=?a]
...
TV[OBJCASE=dat, AGR=[NUM=sg, PER=1]] -> 'folge' | 'helfe'
TV[OBJCASE=dat, AGR=[NUM=sg, PER=2]] -> 'folgst' | 'hilfst'
...
```

The next example, in Figure 3, illustrates the parse tree we get for this grammar, given a sentence containing a verb which governs dative case, *ich folge den Katzen*, or “I follow the cats.”

In developing grammars, excluding ungrammatical word sequences is often as challenging than parsing grammatical ones. In order to get an idea where and why a sequence fails to parse, setting the trace parameter can be crucial. Consider the following parse failure: we try the sentence, *ich folge den Katze*. This will not parse with our grammar, why not? The last two lines in the trace show that *den* is recognized as admitting two possible categories: *Det[AGR=[GND=’masc’, NUM=’sg’, PER=3], CASE=’acc’]* and *Det[AGR=[NUM=’pl’, PER=3], CASE=’dat’]*. We know from the grammar in that *Katze* has category *N[AGR=[GND=’fem’, NUM=’sg’, PER=3]]*. Thus there is no binding for the variable *?a* in the rule *NP[CASE=?c, AGR=?a] -> Det[CASE=?c, AGR=?a] N[CASE=?c, AGR=?a]* which will satisfy these constraints, since the AGR value of *Katze* will not unify with either of the AGR values of *den*, that is, with either *[GND=’masc’, NUM=’sg’, PER=3]* or *[NUM=’pl’, PER=3]*.
Figure 3: A feature-based grammar parsing a German sentence, *ich folge den Katzen*, showing how the dative case on the Object agrees with the dative ending on the Verb.

Feature Bottom Up Predict Combine Rule:
| [---] . . . | [0:1] PRO[AGR=[NUM='sg', PER=1], CASE='nom'] -> 'ich' *

Feature Bottom Up Predict Combine Rule:
| [---] . . . | [0:1] NP[AGR=[NUM='sg', PER=1], CASE='nom'] -> PRO[AGR=[NUM='sg', PER=1], CASE='nom'] *

Feature Bottom Up Predict Combine Rule:
| [---] . . . | [0:1] S[] -> NP[AGR=?a, CASE='nom'] * VP[AGR=?a] {?a: [NUM='sg', PER=1]}*

Feature Bottom Up Predict Combine Rule:
| . . [---] . | [1:2] TV[AGR=[NUM='sg', PER=1], OBJCASE='dat'] -> 'folge' *

Feature Bottom Up Predict Combine Rule:
  {?a: [NUM='sg', PER=1], ?c: 'dat'}

Feature Bottom Up Predict Combine Rule:
| . . [---] . | [2:3] Det[AGR=[GND='masc', NUM='sg', PER=3], CASE='acc'] -> 'den' *

Feature Bottom Up Predict Combine Rule:
| . . [---] . | [2:3] Det[AGR=[NUM='pl', PER=3], CASE='dat'] -> 'den' *

Feature Bottom Up Predict Combine Rule:
  {?a: [NUM='pl', PER=3], ?c: 'dat'}

Feature Bottom Up Predict Combine Rule:
  {?a: [GND='masc', NUM='sg', PER=3], ?c: 'acc'}

Feature Bottom Up Predict Combine Rule:
| . . . [---] . | [3:4] N[AGR=[GND='fem', NUM='sg', PER=3]] -> 'Katze' *

3 The problem of “displaced” phrases

A very common phenomenon in human language is that of *displaced phrases*: that is, an NP, a Prepositional Phrase (PP), a Verb Phrase (VP), indeed any kind of phrase, may be moved from its “canonical” position in a sentence. Here are some examples. We have indicated the canonical position for the displaced phrase as a sequence of X’s.
And there are many more different kinds of examples like these. Much, perhaps most, variation within and across languages, apart from lexical variation, seems to be due to displacements like this, and so far, we have no way to represent them succinctly. The notion that there is a core sentence of sentences, and then various manipulations that ring all the changes on them, is what is behind the idea of modern transformational generative grammar. So the idea is, once one has all the basic verb-predicate structures, all the rest follow from a set of simple rules that relate one structure to another. If it works, the result is an enormous savings in space (as well as ease in learning).

Moreover, the possible displacements found in human language all follow very particular constraints. Displacement cannot be just to any location. First, all the locations where NPs, PPs, Ss, and so forth might wind up, what are called “landing sites”, must also be places where they might otherwise be found without such displacement. Second, we have already seen that there is nothing in human language that could specify that a displacement be to exactly the third position in a sentence. Additionally, displacements must obey a particular constraint that is known in linguistics as c-command. The basic result is that the displaced phrase must c-command the position from which it is displaced. But what is c-command? It has a simple verbal description, but it is even easier to depict graphically, and also intuitively. Intuitively, c-command is the notion of variable scope, as used in human languages (as opposed to programming languages). We can define it this way with respect to hierarchical structures as produced by a parser:

**Definition** A phrase X c-commands a phrase or position Y if and only if the first branching node that dominates X also dominates Y.

For example, consider the sentence, *the guy ate the ice-cream*, and refer to figure 4. Here we have $S =$ the Subject NP *the guy*. The first branching node above this NP is the S node. The S node dominates the Object NP *ice-cream*. Therefore, the Subject NP c-commands the Object NP. Similarly, the Subject NP c-commands the verb *ate*, and the VP. However, obviously the c-command relation is not symmetric, because the Object NP does not c-command the Subject NP: the first branching node above *the ice-cream* is the V$^P$, and the V$^P$ does not dominate the Subject NP. Thus, all displacements have to be somewhere “above” their original locations, in a strict hierarchical sense. The next figure, Figure 5, shows that a displaced phrase like *this guy*, in *this guy I want to solve the problem*, occupies a position where the first branching node above it is called a “Complement Phrase” (or CP), and CP again dominates the position where *this guy* must have originated, as the Subject of the embedded sentence, *this guy to solve the problem*. The same is true for all the other sentences listed above. We will see that c-command plays a key role not only in this context, but also with respect to how pronouns get their antecedents (as in a sentence such as, *Obama thinks that he is too sharp to lose*, where *Obama* and *he* can be, but need not be, the same person.

In general, c-command appears to be the way that the environment frames for variables in general are used in human language (now thinking of the programming language sense). In each case, either an empty phrase, a pronoun, or, in one other case we haven’t mentioned so far, quantifiers like *every* and *some*, c-command plays a big role. It may be considered to be one of the primitive “building blocks” for human language. We’ll talk more about this in later installments of these notes.

Further, again following the scope idea, in order for sentences with displacements to be properly interpreted – that is, so that one can properly interpret the arguments to verbs that act as predicates – there must be some way to link a displaced phrase back to its canonical location. For example, in the question *What did John say that Mary ate* then the question is asking: *For which x, did John say that Mary ate x?*. This can be carried out in the representation in a few different ways; in current linguistic theory, one simply puts
an $x_i$ at the canonical position, and then the same index $i$ is attached to the displaced phrase, to indicate that it refers to the same thing. In CFGs, one has to use a different trick, as we explain below. So in our question sentence, we would have something like this (omitting the hierarchical structure for now):

$$\textit{For What}_i \textit{ did John say that Mary ate } x_i.$$

The $x_i$ is not pronounced, of course, though it is present when the sentence must be interpreted internally, by the brain or by a computer.

Finally, one can ask how far one can displace a phrase. The answer seems to be remarkably simple and follows the spirit of adjacency that we mentioned earlier as one of the central constraints on human language: one can only displace a phrase to the next higher (ie, adjacent, hierarchically) phrase. You might then ask, what about a question like the one just above, where it appears that what can be displaced arbitrarily far from its canonical position after the verb: \textit{What did John ... that Mary ate XXX}. In this case, it can be shown by linguistic arguments that in fact the wh-phrase \textit{What} has been successively displaced to intermediate points along the way to its finding landing site, something like this, \textit{For what$-i$ did John say $x_i$ that Mary liked $x_i$}. Importantly, each single displacement is completely local, just one adjacent level up as we go. (We will note in passing that this means that parsing can be entirely local, at least with respect to this aspect – even though displacements may \textit{appear} to be arbitrarily far, in fact they only go as far as the next higher phrase.

The constraints on displacement are in fact more subtle than this, and can vary across languages. For example, consider the contrast between the following two sentences, where we have indicated by “XXX” the ‘gap’ which is paired with the filler \textit{How many cars} and then \textit{How many mechanics}:

How many cars did they say that the mechanics fixed XXX? (OK)

How many mechanics did they say that XXX fixed the cars? (not so OK – but better if that is deleted)

In Italian, both of these are fine; as is the second if that is dropped. A question is how all speakers of English have learned this difference. Other examples that violated the “one phrase” constraint on displacement are, however, very bad. In these cases, the attempted displacement fails because what cannot move “in one fell swoop” over two S(entence) boundaries, the first marked by who likes and the second by do you wonder:

What do you wonder who likes XXX? (Meaning: for which X, do you wonder who likes X?)

Importantly, however, that although the original Penn Tree Bank was annotated with relevant information about this (see Figure 6), parsers trained on this dataset did not really make use of this information. This has been remedied in the last few years by a third method of representing displaced phrases, which we’ll discuss later in the course.

In fact, the Penn Tree Bank (PTB) annotated corpus did attempt to represent these so-called “empty categories” (unpronounced phrases) in its representation, though these are generally ignored. For example, Figure 6 shows how one sentence that has its topic displaced out to the front, \textit{The SEC will probably vote on the proposal early next June, he said}, is annotated in the PTB. Note the presence of an empty phrase

![Figure 4: A graphical picture of c-command. The NP node in red c-commands the VP, V, and NP the ice-cream.](image-url)
Figure 5: Displaced phrases must c-command the canonical positions from which they are displaced. Here, in the sentence *This student I want to solve the problem, this student* has been displaced from its position as the Subject of *to solve*, as indicated by the empty element with an index $i$. The displaced NP with index $i$ c-commands the empty element’s position.

labeled $T$ that is below the $S$ node at the end of the sentence. It has the index 1, which is linked to the $S$ at the front, which is labeled $S$-TPC-1 (TPC stands for “topic”). The annotation $T$ means “trace” which is an older linguistics term for an unpronounced, displaced phrase. So we see that the PTB has represented this structure in an appropriate way, so that interpretation is made transparent. (There is a second phrase with the label -NONE-, but this is not a true empty phrase; it is simply stating that in English, one can omit the word “that” in an embedded sentence, e.g., *I know that Bill runs, I know Bill runs.*)

How can we represent sentence with displacements using CFGs? Note that there are three parts to such a representation: (1) We have to have a CFG rule that generates the displaced phrase in a non-canonical position; and (2) we have to have some way of “linking up” the displaced phase with the canonical position from which it is displaced; (3) we must ensure that the displaced phrase is not pronounced twice – in the “lower” position (its canonical position of origin), the phrase is “silent.”

Here’s how this has been done in at least one way, in a theory that was first called *generalized phrase structure grammar* (GPSG), and then later on, adopted by a linguistic theory that embraces both features and this GPSG mechanism, known as “HPSG.” The key idea is due to Prof. Gil Harman, in 1963 (but GPSG was not developed until 1979).

1. Add new nonterminal rules that map NPs, Ss, PPs, etc. to the empty string, $\epsilon$. This corresponds to an expansion of a phrase that is then “unpronounced” or silent. For example, for NP we will have the rule, $NP \rightarrow \epsilon$. Informally, we will call the position of the empty phrase a “gap.” (In fact, this is not exactly the $NP$ expansion rule we want, and we’ll have to modify it slightly as described below.)

2. Add new nonterminal rules corresponding to the introduction of displaced phrase, along with new nonterminal names to keep track of the “link” between the displaced phrase and the place where the empty phrase is. This is done by taking a regular nonterminal, say, $S$ or $VP$, and then creating an additional new non-terminal that has a “slash” in it, with a non-terminal name after the slash that stands for the kind of phrase that has been displaced, and so will eventually expand out as the empty string $\epsilon$ *somewhere below in the expansion of the regular nonterminal*. For example, given the “vanilla” rule with nonterminals, $S \rightarrow NP \ VP$, then we can create a “slash” nonterminal: $VP/NP$, which denotes the fact that somewhere below in the subtree headed by $VP$, the rule $NP \rightarrow \epsilon$ must be applied. Of course, it must be the case that there is an NP that gets “paired up” with this unpronounced NP (the
Figure 6: An example of how the PTB annotates displaced phrases. Here the sentence is a topicalized form, *The SEC will probably vote on the proposal early next June*, *he said*. Note the empty phrase under the final *S*, labeled as a *T* indexed with a 1. This means it is linked to the *S* which has been displaced to the front, which also has the index 1, along with a topic annotation, *−TPC*.

“filler” for the “gap). So the new, additional rule for expanding *S* looks like this: *S* → NP VP/NP. Here, it is tacitly assumed that the first NP (the Subject) is in fact the “filler” for the “gap” that is represented by the slashed nonterminal VP/NP. **Important**: note that the new nonterminal name, VP/NP is an atomic symbol, just like NP or VP. The purpose of the slash form is to keep track – remember – that an NP has been displaced, and that whatever rules expand out VP/NP, then somewhere below there must be a matching epsilon-expanding rule.

3. Therefore, whenever a “slash rule” introduces the possibility of a displaced phrase, one must not only add the initial slashed nonterminal and rule, e.g., VP/NP, one must also add a chain of such rules that terminate in the expansion of NP as ε. For example, given that we have added a new rule *S* → NP VP/NP, one must also add rules that expand VP/NP with the slash “passed on” beneath the VP. For example, if we have the vanilla rule expanding VP as, say, VP NP, then we must also add a new rule and slash nonterminal, VP/NP → VB NP/NP, indicating that the NP “gap” (denoted by the slash) has been passed down to the NP on the right. Finally, we have the rule that terminates the ‘chain’ of slashed rule expansions as ε. For example, in our example sentence, *this student I want to solve the problem*, where this student has been displaced from its position as the subject of to solve, we require rules that look something like the following, resulting in the parse structure that is displayed in Figure 5. It should be readily apparent that the sequence of new “slash: nonterminals for a chain

4. It is even possible to incorporate the idea of feature values to work with the “filler and gap” idea. The method is to introduce a new feature, named GAP, which has as its value the name of the phrase that is being displaced. Then it is this GAP feature that must be given a value (by the displaced NP, S, PP, etc.), and we link displaced phrases (and fillers) to their gaps by means of this feature. This turns out to be very tricky to implement; see Figure 7. The reason is that once one posits the possibility of a
GAP feature, then it is hard to keep it under control. (It must be able to handle even sentences with displacement to the right, as in I need, and he wanted, a book about economics.)

We can illustrate one way to handle “slash” categories with a GAP feature, as follows. In this example grammar, the feature type is used as a feature whose value holds the type of phrase that is found. We also use a feature to mark whether an NP is a wh NP or not (i.e., a phrase like which book or who is a wh NP), because we have to know whether the sentence is a question or not. Finally, the grammar handles inversion of the auxiliary verb and the Subject NP in questions by means of one more feature, called INV. Figure 7 displays the parse of the with a filler and a gap, such as Who do you like, where there is a gap after like and the filler is who? See Figure 7 for a picture of the parse. Note how the feature value for “slash” is given the value NP corresponding to the phrase who, so that this “slash value” is passed down the tree until it reaches the point where the rule NP/NP → ϵ is used (expanding to nothing). Also, the feature INV has the value true, which means that the sentence is an inverted question form, so the auxiliary verb do must be inverted with the Subject NP, you.

The simple grammar for this is listed below, and you can examine it for how the slash feature is used to handle fillers and gaps. Note that the next to last line is the rule expanding NP/NP to an empty symbol. We would have to extend the slash feature to hold other kinds of phrases, in a more realistic grammar.

% start S
S[-INV] -> NP VP
S[-INV]/?x -> NP VP/?x
S[-INV] -> NP S/NP
S[-INV] -> Adv[+NEG] S[+INV]
S[+INV] -> V[+AUX] NP VP
S[+INV]/?x -> V[+AUX] NP VP/?x
SBar -> Comp S[-INV]
SBar/?x -> Comp S[-INV]/?x
VP -> V[SUBCAT=intrans, -AUX]
VP -> V[SUBCAT=trans, -AUX] NP
VP/?x -> V[SUBCAT=trans, -AUX] NP/?x
VP -> V[SUBCAT=clause, -AUX] SBar
VP/?x -> V[SUBCAT=clause, -AUX] SBar/?x
VP -> V[+AUX] VP
VP/?x -> V[+AUX] VP/?x
There are other problems, though, with the using a feature value “pass around” the information about a displaced phrase. Consider example sentences where there are two displaced phrases, as mentioned earlier, where SSS denotes the place from which these sonatas has been displaced, and VVV denotes the place from which which violins is displaced:

Which violins are these sonatas too difficult to play SSS on VVV

Meaning: For wh-x, x= violins, and is it too difficult for (someone) to play these sonatas on these violins

Furthermore, examples like these illustrate that there is a third type of “unpronounced” element in some sentences, corresponding to an (arbitrary) person or thing that we can call PRO (for Pronoun, but unpronounced). PRO can specify someone that is referred to earlier in the sentence, if it is in right structural relationship (you should know this by now: the earlier element must c-command the PRO), or even earlier in the discourse. So for example, we have I promised Bill PRO to leave, where PRO is the understood Subject of the embedded sentences. We’ll return to this point later, but you see that language is full of empty (unpronounced) items, which of course cause difficulties for both parsing and learning.

Returning to the violins-sonatas example with two displaced phrases, in order to represent this in an augmented CFG system, we must use two “slash categories”: one for the first phrase, which violins and one for the second, these sonatas. Thus at some point we must have a new nonterminal that encodes both of these in its augmented name, something like VP/wh-NP/NP, where the 2nd NP in this list after the slash stands for these sonatas, and the first, which violins. One must also complicate the way that the rules rewriting NPs or Wh-NPs are discharged as empty elements: in our example, we need to discharge the rule writing NP as an empty string so that it corresponds to these sonatas, leaving us with just VP/Wh-NP. In this case at least, the addition and then discharge of slashed elements proceeds in a first-in, last-out (pushdown stack) order, but one can see how greatly complicated the rule system must become. One could also encode this kind of example by using multiple “slash” features, but the same issues arise.

If the reader is interested in a real challenge, the final kind of syntactic structure that poses a difficulty for almost every approach is conjunctions like and or or. The problem is that one can conjoin any two or more phrases of the same type, but representing this is a problem. In the PTB, they used a phrase type called CONJ, for conjunction, but this is not really quite right.

It should be evident that unpronounced elements need careful treatment within a CFG framework.

4 CFGs and lexicalized, head-based CFGs

To complete our discussion, and to foreshadow the better use of PCFGs, we introduce the notion of a CFG that incorporates what is called lexical “head” information. What is this about? As mentioned in the notes from lectures 5-6, nearly 40 years ago, it was determined that phrase name information in human languages is completely redundant with information in the lexicon. Thus, a VP is completely determined by the fact that it is grounded on a Verb. Moreover, the dictionary information is more specific. It is this specific information, as we shall see, that should be used in PCFGs (but is not). This view, introduced by Chomsky in 1965 and then more completely in 1970 (in Remarks on Nominalization) is sometimes called lexicalization.
In 1995, Carl deMarcken wrote the first paper on designed a head-centric CFG for use in learning CFGs from data (presented at a meeting of the Association for Computational Linguistics at MIT). Subsequently, this idea was picked up and developed for parsing with PCFGs, as we shall see. Here is the central idea. Consider sentence fragments like the following.

walk on ice
walk to ice (?)
give to Mary a book
give with Mary a book (?)
take from Mary a toy

Clearly, the particular details of a verb sometimes dictate what kind of Prepositional Phrase (PP) might follow – with, to, from, etc. Yet none of this information is present in our use of phrase names like VP, VB, or PP. Similarly, as we have seen, the kind of verb dictates what might follow: we have solve a puzzle but not solve an ice-cream. There is no way to distinguish the more fine-grained information illustrated by the examples above.

What to do? One answer is to make the phrase names (the nonterminals of the CFG), more fine-grained according to the lexical information on which they are based. We already know there is linguistic evidence for this. So for example, the VP for walk on water, is based on the Verb, walk. de Marcken suggested changing the representation of this VP to the following format: VP(walk), where walk is the head. Clearly, this introduces a large number of word-specific CFG rules, a point that we will need to address later, as well as estimation problems for PCFGs. For now, though, we note only that DeMarcken did this for every ordinary CFG rule, resulting in a new CFG that is obviously equivalent both in terms of strings and structure that it generates to the original CFG. So for example, we replace a rule like:

\[ S \rightarrow \text{NP VP} \]

with:

\[ S(\text{gave}) \rightarrow \text{NP(John) VP(gave)} \]

More generally, we will do the following. For each rule in the form,

\[ X \rightarrow Y_1 Y_2 \ldots Y_n \]

there is an index \( h \in \{1 \ldots n\} \) that specifies the head of a rule. (We will need to use linguistic knowledge to define the head of each phrase – there are some differences conventionally adopted that are at odds with the linguistic formulation, in fact.) For example, in our \( S \) expansion rule above, the head of \( S \) is typically taken to be the VP, hence \( h = 2 \) in this case, or gave. In turn, the head of the VP came from the verb. (In a more linguistically sound account, the \( S \) phrase is actually a type of “inflected phrase” or IP.) An example tree with all the heads marked in bold font is given in Figure 8. (We shall see that it is sometimes difficult to figure out what the head of a phrase should be.) You can see from the figure how the head information is passed up the tree from below.

The notion of a head in linguistic theory is more straightforward when all structures are binary branching. In this case, we pick one or the other of a pair of elements in a structure as the head. With the PTB however, there are sometimes long strings of “flat” structures, or ternary or higher-degree branchings, which make the job more difficult. The PTB does not annotate heads (though it should). For example, in a long stretch before a Noun, as in the big red MIT award screw, screw is the head. More generally, English is a language that is called head-first, because heads occur at the immediate left of a phrase. This rule seems to be violated in NPs. However, there is good reason to believe that the DT is the head of an NP. In this case, the lexical information about the Noun would not be available higher up in the tree. We can see from examples like
these that the real motivation behind heads is to make lexical information available in a parse, not simply to abide by proper linguistic analysis. In the next installment, we shall see how to improve PCFG parsing by using head information.