Chapter 9

Feature Based Grammar

9.1 Introduction

The framework of context-free grammars that we presented in Chapter 7 describes syntactic constituents with the help of a limited set of category labels. These atomic labels are adequate for talking about the gross structure of sentences. But when we start to make finer grammatical distinctions it becomes awkward to enrich the set of categories in a systematic manner. In this chapter we will address this challenge by decomposing categories using features (somewhat similar to the key-value pairs of Python dictionaries).

We will start off by looking at the phenomenon of syntactic agreement; we will show how agreement constraints can be expressed elegantly using features, and illustrate how their use in a simple grammar. Feature structures are a general data structure for representing information of any kind; we will briefly look at them from a more formal point of view, and explain how they are made available in NLTK. In the final part of the chapter, we demonstrate that the additional expressiveness of features opens out a wide spectrum of possibilities for describing sophisticated aspects of linguistic structure.

9.2 Decomposing Lingustic Categories

9.2.1 Syntactic Agreement

Consider the following contrasts:

(1a) this dog

(1b) *these dog

(2a) these dogs

(2b) *this dog

In English, nouns are usually morphologically marked as being singular or plural. The form of the demonstrative also varies in a similar way; there is a singular form *this* and a plural form *these*. Examples (1) and (2) show that there are constraints on the realization of demonstratives and nouns within a noun phrase: either both are singular or both are plural. A similar kind of constraint is observed with subjects and predicates:

(3a) the dog runs

(3b) *the dog run

(4a) the dogs run

(4b) *the dogs runs

Here again, we can see that morphological properties of the verb co-vary with morphological properties of the subject noun phrase; this co-variance is usually termed **agreement**. The element which determines the agreement, here the subject noun phrase, is called the agreement **controller**, while the element whose form is determined by agreement, here the verb, is called the **target**. If we look further at verb agreement in English, we will see that present tense verbs typically have two inflected forms: one for third person singular, and another for every other combination of person and number:

	singular	**plural**
1st per	*I run*	*we run*
2nd per	*you run*	*you run*
3rd per	*he/she/it runs*	*they run*

Table 9.1: Agreement Paradigm for English Regular Verbs

We can make the role of morphological properties a bit more explicit as illustrated in (5) and (6). These representations indicate that the verb agrees with its subject in person and number. (We use "3" as an abbreviation for 3rd person, "SG" for singular and "PL" for plural.)

	the	dog	run-s
(5a)		dog.3.SG	run-3.SG

	the	dog-s	run
(6a)		dog.3.PL	run-3.PL

Despite the undoubted interest of agreement as a topic in its own right, we have introduced it here for another reason: we want to look at what happens when we try encode agreement constraints in a context-free grammar. Suppose we take as our starting point the very simple CFG in (7).

(7)
```
S → NP VP
NP → DET N
VP → V

DET → 'this'
N → 'dog'
V → 'runs'
```

(7) allows us to generate the sentence *this dog runs*; however, what we really want to do is also generate *these dogs run* while blocking unwanted strings such as **this dogs run* and **these dog runs*. The most straightforward approach is to add new non-terminals and productions to the grammar which reflect our number distinctions and agreement constraints (we ignore person for the time being):

(8)
```
S_SG → NP_SG VP_SG
S_PL → NP_PL VP_PL
NP_SG → DET_SG N_SG
NP_PL → DET_PL N_PL
VP_SG → V_SG
VP_PL → V_PL

DET_SG → 'this'
DET_PL → 'these'
N_SG → 'dog'
N_PL → 'dogs'
V_SG → 'runs'
V_PL → 'run'
```

It should be clear that this grammar will do the required task, but only at the cost of duplicating our previous set of rules. Rule multiplication is of course more severe if we add in person agreement constraints.

9.2.2 Using Attributes and Constraints

We spoke informally of linguistic categories having *properties*; for example, that a verb has the property of being plural. Let's try to make this more explicit:

(9)
```
N[NUM pl]
```

In (9), we have introduced some new notation which says that the category N has a **feature** called NUM (short for 'number') and that the value of this feature is *pl* (short for 'plural'). We can add similar annotations to other categories, and use them in lexical entries:

(10)
```
DET[NUM sg] → 'this'
DET[NUM pl]  → 'these'
N[NUM sg] → 'dog'
N[NUM pl] → 'dogs'
V[NUM sg] → 'runs'
V[NUM pl] → 'run'
```

Does this help at all? So far, it looks just like a slightly more verbose alternative to what was specified in (8). Things become more interesting when we allow *variables* over feature values, and use these to state constraints. This is illustrated in (11).

(11a)
```
S → NP[NUM ?n] VP[NUM ?n]
```

(11b)
```
NP[NUM ?n] → DET[NUM ?n] N[NUM ?n]
```

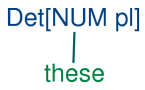
(11c)
```
VP[NUM ?n] → V[NUM ?n]
```

We are using '?*n*' as a variable over values of NUM; it can be instantiated either to *sg* or *pl*. Its scope is limited to individual rules. That is, within (11a), for example, ?*n* must be instantiated to the same constant value; we can read the rule as saying that whatever value NP takes for the feature NUM, VP must take the same value.

In order to understand how these feature constraints work, it's helpful to think about how one would go about building a tree. Lexical rules will admit the following local trees (trees of depth one):

(12a)

(12b)

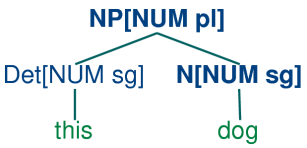


(13a)

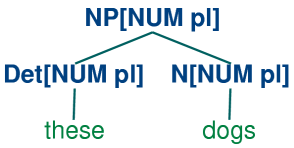
(13b)

Now (11b) says that whatever the NUM values of N and DET are, they have to be the same. Consequently, (11b) will permit (12a) and (13a) to be combined into an NP as shown in (14a) and it will also allow (12b) and (13b) to be combined, as in (14b). By contrast, (15a) and (15b) are prohibited because the roots of their constituent local trees differ in their values for the NUM feature.

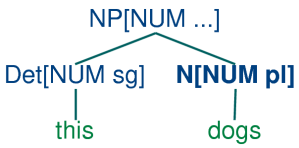
(14a)



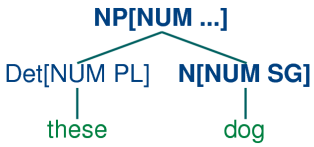
(14b)



(15a)

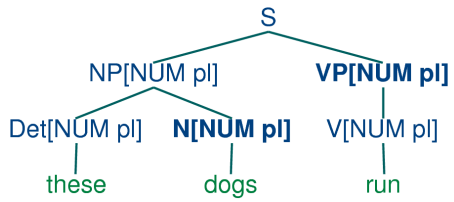


(15b)



Rule (11c) can be thought of as saying that the NUM value of the head verb has to be the same as the NUM value of the VP mother. Combined with (11a), we derive the consequence that if the NUM value of the subject head noun is *pl*, then so is the NUM value of the VP's head verb.

(16)



Grammar (17) illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones. As you will see, the format of feature specifications in these productions inserts a `'=` between the feature and its value. In our exposition, we will stick to our earlier convention of just leaving space between the feature and value, except when we are directly referring to the NLTK grammar formalism.

(17)

```
% start S
############################
# Grammar Rules
############################

# S expansion rules
S -> NP[NUM=?n] VP[NUM=?n]

# NP expansion rules
NP[NUM=?n] -> N[NUM=?n]
NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]

# VP expansion rules
VP[TENSE=?t, NUM=?n] -> IV[TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> TV[TENSE=?t, NUM=?n] NP

############################
# Lexical Rules
############################

Det[NUM=sg] -> 'this' | 'every'
Det[NUM=pl] -> 'these' | 'all'
Det -> 'the' | 'some'

PropN[NUM=sg]-> 'Kim' | 'Jody'

N[NUM=sg] -> 'dog' | 'girl' | 'car' | 'child'
N[NUM=pl] -> 'dogs' | 'girls' | 'cars' | 'children'

IV[TENSE=pres,  NUM=sg] -> 'disappears' | 'walks'
TV[TENSE=pres, NUM=sg] -> 'sees' | 'likes'

IV[TENSE=pres,  NUM=pl] -> 'disappear' | 'walk'
TV[TENSE=pres, NUM=pl] -> 'see' | 'like'
```

```
IV[TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
TV[TENSE=past, NUM=?n] -> 'saw' | 'liked'
```

You will notice that a feature annotation on a syntactic category can contain more than one specification; for example, V[TENSE *pres*, NUM *pl*]. In general, there is no upper bound on the number of features we specify as part of our syntactic categories.

A second point is that we have used feature variables in lexical entries as well as grammatical rules. For example, *the* has been assigned the category DET[NUM ?*n*]. Why is this? Well, you know that the definite article *the* can combine with both singular and plural nouns. One way of describing this would be to add two lexical entries to the grammar, one each for the singular and plural versions of *the*. However, a more elegant solution is to leave the NUM value **underspecified** and letting it agree in number with whatever noun it combines with.

A final detail about (17) is the statement `%start S`. This a "directive" which tells the parser to take S as the start symbol for the grammar.

In general, when we are trying to develop even a very small grammar, it is convenient to put the rules in a file where they can be edited, tested and revised. Assuming we have saved (17) as a file named `'feat0.cfg'`, the function `GrammarFile.read_file()` allows us to read the grammar into NLTK, ready for use in parsing.

```
>>> from nltk_lite.parse import GrammarFile
>>> from pprint import pprint
>>> g = GrammarFile.read_file('feat0.cfg')
```

We can inspect the rules and the lexicon using the commands `print g.earley_grammar()` and `pprint(g.earley_lexicon())`.

Next, we can tokenize a sentence and use the `get_parse_list()` function to invoke the Earley chart parser.

It is important to observe that the parser works directly with the underspecified productions given by the grammar. That is, the Predictor rule does not attempt to compile out all admissible feature combinations before trying to expand the non-terminals on the lefthand side of a production. However, when the Scanner matches an input word against a lexical rule that has been predicted, the new edge will typically contain fully specified features; e.g., the edge [PropN[NUM = *sg*] → 'Kim', (0, 1)]. Recall from Chapter 7 that the Fundamental (or Completer) Rule in standard CFGs is used to combine an incomplete edge that's expecting a nonterminal *B* with a following, complete edge whose left hand side matches *B*. In our current setting, rather than checking for a complete match, we test whether the expected category *B* will **unify** with the lefthand side *B'* of a following complete edge. We will explain in more detail in Section 9.3 how unification works; for the moment, it is enough to know that as a result of unification, any variable values of features in *B* will be instantiated by constant values in the corresponding feature structure in *B'*, and these instantiated values will be used in the new edge added by the Completer. This instantiation can be seen, for example, in the edge [NP[NUM *sg*] → PropN[NUM *sg*] •, (0, 1)] in 9.1, where the feature NUM has been assigned the value *sg*.

Finally, we can inspect the resulting parse trees (in this case, a single one).

```
>>> for tree in trees: print tree
...
([INIT]:
  (Start:
    (S:
      (NP[NUM=sg]: (PropN[NUM=sg]: 'Kim'))
```

Listing 1 Trace of Feature-Based Chart Parser

```
>>> from nltk_lite import tokenize
>>> sent = 'Kim likes children'
>>> tokens = list(tokenize.whitespace(sent))
>>> tokens
['Kim', 'likes', 'children']
>>> cp = g.earley_parser(trace=10)
>>> trees = cp.get_parse_list(tokens)
          |.K.l.c.|

Predictor |> . . .| S -> * NP[NUM=?n] VP[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * N[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * PropN[NUM=?n]
Predictor |> . . .| NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n]
Predictor |> . . .| NP[NUM=pl] -> * N[NUM=pl]
Scanner   |[-] . .| [0:1] 'Kim'
Completer |[-] . .| NP[NUM=sg] -> PropN[NUM=sg] *
Completer |[-> . .| S -> NP[NUM=sg] * VP[NUM=sg]
Predictor |. > . .| VP[NUM=?n, TENSE=?t] -> * IV[NUM=?n, TENSE=?t]
Predictor |. > . .| VP[NUM=?n, TENSE=?t] -> * TV[NUM=?n, TENSE=?t] NP
Scanner   |. [-] .| [1:2] 'likes'
Completer |. [-> .| VP[NUM=sg, TENSE=pres] -> TV[NUM=sg, TENSE=pres] * NP
Predictor |. . > .| NP[NUM=?n] -> * N[NUM=?n]
Predictor |. . > .| NP[NUM=?n] -> * PropN[NUM=?n]
Predictor |. . > .| NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n]
Predictor |. . > .| NP[NUM=pl] -> * N[NUM=pl]
Scanner   |. . [-]| [2:3] 'children'
Completer |. . [-]| NP[NUM=pl] -> N[NUM=pl] *
Completer |. [---]| VP[NUM=sg, TENSE=pres] -> TV[NUM=sg, TENSE=pres] NP *
Completer |[=====]| S -> NP[NUM=sg] VP[NUM=sg] *
Completer |[=====]| [INIT] -> S *
```

```
(VP[NUM=sg, TENSE=pres]:
  (TV[NUM=sg, TENSE=pres]: 'likes')
  (NP[NUM=pl]: (N[NUM=pl]: 'children'))))))
```

9.2.3 Terminology

So far, we have only seen feature values like *sg* and *pl*. These simple values are usually called **atomic** — that is, they can't be decomposed into subparts. A special case of atomic values are **boolean** values, that is, values which just specify whether a property is true or false of a category. For example, we might want to distinguish **auxiliary** verbs such as *can*, *may*, *will* and *do* with the boolean feature AUX. Then our lexicon for verbs could include entries such as the following:

(18)
```
V[TENSE pres, AUX +] → 'can'
V[TENSE pres, AUX +] → 'may'

V[TENSE pres, AUX –] → 'walks'
V[TENSE pres, AUX –] → 'likes'
```

A frequently used abbreviation for boolean features allows the value to be prepended to the feature:

(19)
```
V[TENSE pres, +AUX] → 'can'
V[TENSE pres, –AUX] → 'walks'
```

We have spoken informally of attaching 'feature annotations' to syntactic categories. A more general approach is to treat the whole category — that is, the non-terminal symbol plus the annotation — as a bundle of features. Consider, for example, the object we have written as (20).

(20)
```
N[NUM sg]
```

The syntactic category N, as we have seen before, provides part of speech information. This information can itself be captured as a feature value pair, using POS to represent "part of speech":

(21)
```
[POS N, NUM sg]
```

In fact, we regard (21) as our "official" representation of a feature-based linguistic category, and (20) as a convenient abbreviation. A bundle of feature-value pairs is called a **feature structure** or an **attribute value matrix** (AVM). A feature structure which contains a specification for the feature POS is a **linguistic category**.

In addition to atomic-valued features, we allow features whose values are themselves feature structures. For example, we might want to group together agreement features (e.g., person, number and gender) as a distinguished part of a category, as shown in (22).

(22)
$$\begin{bmatrix} \text{POS} & N \\ \text{AGR} & \begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & pl \\ \text{GND} & fem \end{bmatrix} \end{bmatrix}$$

In this case, we say that the feature AGR has a **complex** value.

There is no particular significance to the *order* of features in a feature structure. So (22) is equivalent to (22).

(23)
$$\begin{bmatrix} & \begin{bmatrix} \text{NUM} & pl \\ \text{PER} & 3 \\ \text{GND} & fem \end{bmatrix} \\ \text{AGR} & \\ \text{POS} & N \end{bmatrix}$$

Once we have the possibility of using features like AGR, we can refactor a grammar like (17) so that agreement features are bundled together. A tiny grammar illustrating this point is shown in (24).

(24)
```
S → NP[AGR ?n] VP[AGR ?n]
NP[AGR ?n] → PropN[AGR ?n]
VP[TENSE ?t, AGR ?n] → Cop[TENSE ?t, AGR ?n] Adj
Cop[TENSE pres, AGR [NUM sg, PER 3]] → 'is'
PropN[AGR [NUM sg, PER 3]] → 'Kim'
Adj → 'happy'
```

9.2.4 Exercises

1. ☼ What constraints are required to correctly parse strings like *I am happy* and *she is happy* but not **you is happy* or **they am happy*? Implement two solutions for the present tense paradigm of the verb *be* in English, first taking Grammar (8) as your starting point, and then taking Grammar (24) as the starting point.

2. ☼ Develop a variant of grammar (17) which uses a COUNT to make the distinctions shown below:

   (25a) The boy sings.

   (25b) *Boy sings.

   (26a) The boys sing.

   (26b) Boys sing.

   (27a) The boys sing.

   (27b) Boys sing.

   (28a) The water is precious.

   (28b) Water is precious.

3. ◑ Develop a feature-based grammar that will correctly describe the following Spanish noun phrases:

   (29a)

un	cuadro	hermos-o
INDEF.SG.MASC	picture	beautiful-SG.MASC

   'a beautiful picture'

(29b)	un-os INDEF-PL.MASC	cuadro-s picture-PL	hermos-os beautiful-PL.MASC
	'beautiful pictures'		
(29c)	un-a INDEF.SG.FEM	cortina curtain	hermos-a beautiful-SG.FEM
	'a beautiful curtain'		
(29d)	un-as INDEF.PL.FEM	cortina-s curtain	hermos-as beautiful-PL.FEM
	'beautiful curtains'		

4. ◑ Develop a wrapper for the `earley_parser` so that a trace is only printed if the input string fails to parse.

9.3 Computing with Feature Structures

In this section, we will show how feature structures can be constructed and manipulated in NLTK. We will also discuss the fundamental operation of unification, which allows us to combine the information contained in two different feature structures.

9.3.1 Feature Structures in NLTK

Feature structures in NLTK are declared with the `FeatureStructure()` constructor. Atomic feature values can be strings or integers.

```
>>> from nltk_lite.featurestructure import *
>>> fs1 = FeatureStructure(TENSE='past', NUM='sg')
>>> print fs1
[ NUM   = 'sg'   ]
[ TENSE = 'past' ]
```

We can think of a feature structure as being like a Python dictionary, and access its values by indexing in the usual way.

```
>>> fs1 = FeatureStructure(PER=3, NUM='pl', GND='fem')
>>> print fs1['GND']
fem
```

However, we cannot use this syntax to *assign* values to features:

```
>>> fs1['CASE'] = 'acc'
Traceback (most recent call last):
...
KeyError: 'CASE'
```

We can also define feature structures which have complex values, as discussed earlier.

```
>>> fs2 = FeatureStructure(POS='N', AGR=fs1)
>>> print fs2
[       [ GND = 'fem' ] ]
[ AGR = [ NUM  = 'pl'  ] ]
[       [ PER  = 3     ] ]
[                        ]
[ POS = 'N'              ]
>>> print fs2['AGR']
[ GND = 'fem' ]
[ NUM  = 'pl'  ]
[ PER  = 3     ]
>>> print fs2['AGR']['PER']
3
```

An alternative method of specifying feature structures in NLTK is to use the `parse` method of `FeatureStructure`. This gives us the facility to use square bracket notation for embedding one feature structure within another.

```
>>> FeatureStructure.parse("[POS='N', AGR=[PER=3, NUM='pl', GND='fem']]")
[AGR=[GND='fem', NUM='pl', PER=3], POS='N']
```

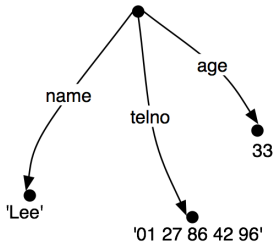
9.3.2 Feature Structures as Graphs

Feature structures are not inherently tied to linguistic objects; they are general purpose structures for representing knowledge. For example, we could encode information about a person in a feature structure:

```
>>> person01 = FeatureStructure(name='Lee', telno='01 27 86 42 96', age=33)
```

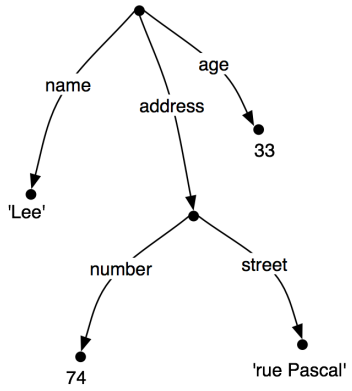
(30)
$$\begin{bmatrix} \text{NAME} & \textit{'Lee'} \\ \text{TELNO} & \textit{01 27 86 42 96} \\ \text{AGE} & \textit{33} \end{bmatrix}$$

It is sometimes helpful to view feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (31) is equivalent to the AVM (30).

(31)


The feature names appear as labels on the directed arcs, and feature values appear as labels on the nodes which are pointed to by the arcs.

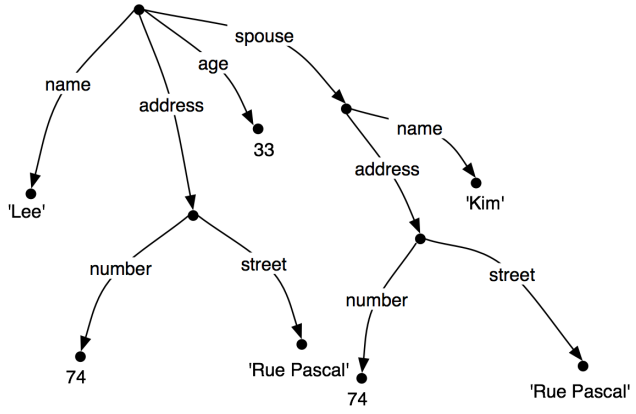
Just as before, feature values can be complex:



(32)

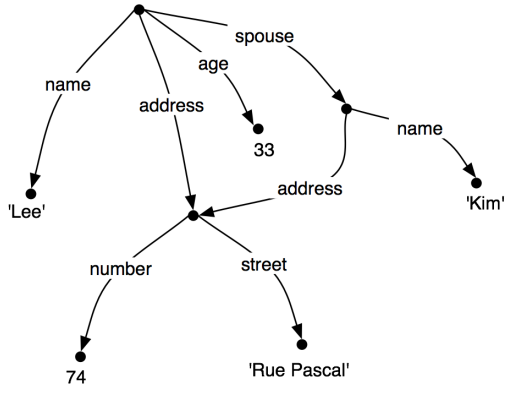
When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths in NLTK as tuples. Thus, `('address', 'street')` is a feature path whose value in (32) is the string 'rue Pascal'.

Now let's consider a situation where Lee has a spouse named "Kim", and Kim's address is the same as Lee's. We might represent this as (33).



(33)

However, rather than repeating the address information in the feature structure, we can "share" the same sub-graph between different arcs:



(34)

In other words, the value of the path `('ADDRESS')` in (34) is identical to the value of the path `('SPOUSE', 'ADDRESS')`. DAGs such as (34) are said to involve **structure sharing** or **reentrancy**. When two paths have the same value, they are said to be **equivalent**.

There are a number of notations for representing reentrancy in matrix-style representations of feature structures. In NLTK, we adopt the following convention: the first occurrence of a shared feature structure is prefixed with an integer in parentheses, such as `(1)`, and any subsequent reference to that structure uses the notation `->(1)`, as shown below.

```
>>> fs=FeatureStructure.parse("""[NAME='Lee', ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
...                               SPOUSE=[NAME='Kim', ADDRESS->(1)]]""")
>>> print fs
[ ADDRESS = (1) [ NUMBER = 74           ] ]
[               [ STREET = 'rue Pascal' ] ]
[                                         ]
[ NAME    = 'Lee'                         ]
[                                         ]
[ SPOUSE  = [ ADDRESS -> (1)  ]           ]
[           [ NAME    = 'Kim' ]           ]
```

This is similar to more conventional displays of AVMs, as shown in (35).

(35)
$$\begin{bmatrix} \text{ADDRESS} & \boxed{1}\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

The bracketed integer is sometimes called a **tag** or a **coindex**. The choice of integer is not significant. There can be any number of tags within a single feature structure.

```
>>> fs1 = FeatureStructure.parse("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
```

(36)
$$\begin{bmatrix} \text{A} & \text{'a'} \\ \text{B} & \boxed{1}\begin{bmatrix} \text{C} & \text{'c'} \end{bmatrix} \\ \text{D} & \boxed{1} \\ \text{E} & \boxed{1} \end{bmatrix}$$

9.3.3 Subsumption and Unification

It is standard to think of feature structures as providing **partial information** about some object, in the sense that we can order feature structures according to how general they are. For example, (37a) is more general (less specific) than (37b), which in turn is more general than (37c).

(37a)
$$\begin{bmatrix} \text{NUMBER} & 74 \end{bmatrix}$$

(37b)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix}$$

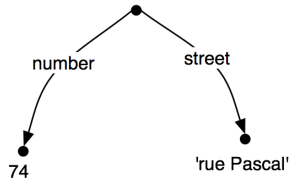
(37c)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \\ \text{CITY} & \text{'Paris'} \end{bmatrix}$$

This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write FS_0 FS_1), then FS_1 must have all the paths and path equivalences of FS_0, and may have additional paths and equivalences as well. Thus, (33) subsumes (34), since the latter has additional path equivalences.. It should be obvious that subsumption only provides a partial ordering on feature structures, since some feature structures are incommensurable. For example, (38) neither subsumes nor is subsumed by (37a).

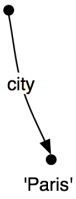
(38) $\left[\text{TELNO} \quad 01\ 27\ 86\ 42\ 96\right]$

So we have seen that some feature structures are more specific than others. How do we go about specialising a given feature structure? For example, we might decide that addresses should consist of not just a street number and a street name, but also a city. That is, we might want to *merge* graph (39b) with (39a) to yield (39c).

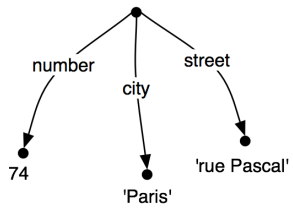
(39a)



(39b)



(39c)



Merging information from two feature structures is called **unification** and in NLTK is supported by the `unify()` method defined in the `FeatureStructure` class.

```
>>> fs1 = FeatureStructure(NUMBER=74, STREET='rue Pascal')
>>> fs2 = FeatureStructure(CITY='Paris')
>>> print fs1.unify(fs2)
[ CITY   = 'Paris'      ]
[ NUMBER = 74           ]
[ STREET = 'rue Pascal' ]
```

Unification is formally defined as a binary operation: FS_0 FS_1. Unification is symmetric, so

(40) FS_0 $FS_1 = FS_1$ FS_0.

The same is true in NLTK:

```
>>> print fs2.unify(fs1)
[ CITY   = 'Paris'      ]
[ NUMBER = 74           ]
[ STREET = 'rue Pascal' ]
```

If we unify two feature structures which stand in the subsumption relationship, then the result of unification is the most specific of the two:

(41) If $FS_0 \sqsubseteq FS_1$, then $FS_0 \sqcup FS_1 = FS_1$

For example, the result of unifying (37b) with (37c) is (37c).

Unification between FS_0 and FS_1 will fail if the two feature structures share a path π, but the value of π in FS_0 is a distinct atom from the value of π in FS_1. In NLTK, this is implemented by setting the result of unification to be `None`.

```
>>> fs0 = FeatureStructure(A='a')
>>> fs1 = FeatureStructure(A='b')
>>> fs2 = fs0.unify(fs1)
>>> print fs2
None
```

Now, if we look at how unification interacts with structure-sharing, things become really interesting. First, let's define the NLTK version of (33).

```
>>> fs0=FeatureStructure.parse("""[NAME=Lee,
...                                ADDRESS=[NUMBER=74,
...                                         STREET='rue Pascal'],
...                                SPOUSE= [NAME=Kim,
...                                         ADDRESS=[NUMBER=74,
...                                                  STREET='rue Pascal']]]""")
```

(42)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

What happens when we augment Kim's address with a specification for CITY? (Notice that `fs1` includes the whole path from the root of the feature structure down to CITY.)

```
>>> fs1=FeatureStructure.parse("[SPOUSE = [ADDRESS = [CITY = Paris]]]")
```

(43) shows the result of unifying `fs0` with `fs1`:

(43)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{CITY} & \text{'Paris'} \\ \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

By contrast, the result is very different if `fs1` is unified with the structure-sharing version `fs2` (also shown as (34)):

```
>>> fs2=FeatureStructure.parse("""[NAME=Lee, ADDRESS=(1)[NUMBER=74, STREET='rue Pas
...                                 SPOUSE=[NAME=Kim, ADDRESS->(1)]]""")
```

(44)
$$\begin{bmatrix} & \begin{bmatrix} \text{CITY} & \textit{'Paris'} \\ \text{NUMBER} & \textit{74} \\ \text{STREET} & \textit{'rue Pascal'} \end{bmatrix} \\ \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \textit{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \textit{'Kim'} \end{bmatrix} \end{bmatrix}$$

Rather than just updating what was in effect Kim's "copy" of Lee's address, we have now updated *both* their addresses at the same time. More generally, if a unification involves specialising the value of some path π, then that unification simultaneously specialises the value of *any path that is equivalent to* π.

As we have already seen, structure sharing can also be stated in NLTK using variables such as `?x`.

```
>>> fs1=FeatureStructure.parse("[ADDRESS1=[NUMBER=74, STREET='rue Pascal']]")
>>> fs2=FeatureStructure.parse("[ADDRESS1=?x, ADDRESS2=?x]")
>>> print fs2
[ ADDRESS1 = ?x ]
[ ADDRESS2 = ?x ]
>>> print fs2.unify(fs1)
[ ADDRESS1 = (1) [ NUMBER = 74           ] ]
[                [ STREET = 'rue Pascal' ] ]
[                                          ]
[ ADDRESS2 -> (1)                          ]
```

9.3.4 Exercises

1. ☼ Write a function *subsumes()* which holds of two feature structures `fs1` and `fs2` just in case `fs1` subsumes `fs2`.

2. ◑ Consider the feature structures shown in Listing 9.2.

Listing 2

```
fs1 = FeatureStructure.parse("[A = (1)b, B= [C ->(1)]]")
fs2 = FeatureStructure.parse("[B = [D = d]]")
fs3 = FeatureStructure.parse("[B = [C = d]]")
fs4 = FeatureStructure.parse("[A = (1)[B = b], C->(1)]")
fs5 = FeatureStructure.parse("[A = [D = (1)e], C = [E -> (1)] ]")
fs6 = FeatureStructure.parse("[A = [D = (1)e], C = [B -> (1)] ]")
fs7 = FeatureStructure.parse("[A = [D = (1)e, F = (2)[]], C = [B -> (1), E -> (2)]
fs8 = FeatureStructure.parse("[A = [B = b], C = [E = [G = e]]]")
fs9 = FeatureStructure.parse("[A = (1)[B = b], C -> (1)]")
```

Work out on paper what the result is of the following unifications. (Hint: you might find it useful to draw the graph structures.)

- `fs1` and `fs2`
- `fs1` and `fs3`
- `fs4` and `fs5`
- `fs5` and `fs6`
- `fs7` and `fs8`
- `fs7` and `fs9`

Check your answers on the computer.

3. ◑ List two feature structures which subsume [A=?x, B=?x].

4. ◑ Ignoring structure sharing, give an informal algorithm for unifying two feature structures.

9.4 Extending a Feature-Based Grammar

9.4.1 Subcategorization

In Chapter 7, we proposed to augment our category labels in order to represent different subcategories of verb. More specifically, we introduced labels such as IV and TV for intransitive and transitive verbs respectively. This allowed us to write rules like the following:

(45)
```
VP → IV
VP → TV NP
```

Although it is tempting to think of IV and TV as two kinds of V, this is unjustified: from a formal point of view, IV has no closer relationship with TV than it does, say, with NP. As it stands, IV and TV are unanalyzable nonterminal symbols from a CFG. One unwelcome consequence is that we do not seem able to say anything about the class of verbs in general. For example, we cannot say something like "All lexical items of category V can be marked for tense", since *bark*, say, is an item of category IV, not V.

Using features gives us some useful room for manoeuvre but there is no obvious consensus on how to model subcategorization information. One approach which has the merit of simplicity is due to Generalized Phrase Structure Grammar (GPSG). GPSG stipulates that lexical categories may bear a SUBCAT whose values are integers. This is illustrated in a modified portion of (17), shown in (46).

(46)
```
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=0, TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=1, TENSE=?t, NUM=?n] NP
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=2, TENSE=?t, NUM=?n] Sbar

V[SUBCAT=0, TENSE=pres, NUM=sg] -> 'disappears' | 'walks'
V[SUBCAT=1, TENSE=pres, NUM=sg] -> 'sees' | 'likes'
V[SUBCAT=2, TENSE=pres, NUM=sg] -> 'says' | 'claims'

V[SUBCAT=0, TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
V[SUBCAT=1, TENSE=pres, NUM=pl] -> 'see' | 'like'
```

```
V[SUBCAT=2, TENSE=pres, NUM=pl] -> 'say' | 'claim'

V[SUBCAT=0, TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
V[SUBCAT=1, TENSE=past, NUM=?n] -> 'saw' | 'liked'
V[SUBCAT=2, TENSE=past, NUM=?n] -> 'said' | 'claimed'
```

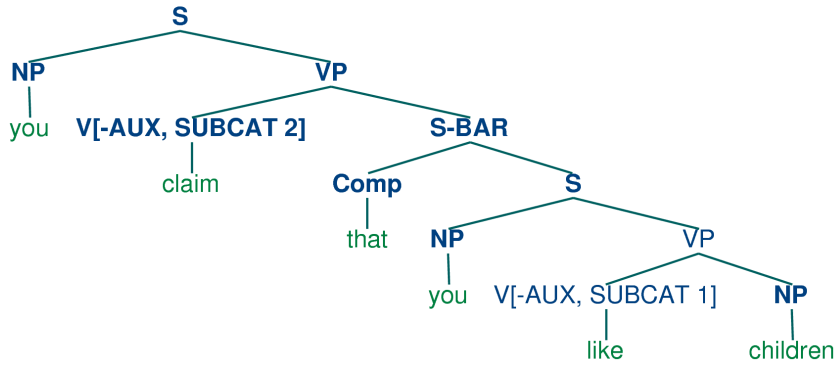
When we see a lexical category like V[SUBCAT *1*], we can interpret the SUBCAT specification as a pointer to the rule in which V[SUBCAT *1*] is introduced as the head daughter in a VP expansion rule. By convention, there is a one-to-one correspondence between SUBCAT values and rules which introduce lexical heads. It's worth noting that the choice of integer which acts as a value for SUBCAT is completely arbitrary — we could equally well have chosen 3999, 113 and 57 as our two values in (46). On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP.

In our third class of verbs above, we have specified a category S-BAR. This is a label for subordinate clauses such as the complement of *claim* in the example *You claim that you like children*. We require two further rules to analyse such sentences:

(47)
```
S-BAR -> Comp S
Comp -> 'that'
```

The resulting structure is the following.

(48)



An alternative treatment of subcategorization, due originally to a framework known as categorial grammar, is represented in feature-based frameworks such as PATR and Head-driven Phrase Structure Grammar. Rather than using SUBCAT values as a way of indexing rules, the SUBCAT value directly encodes the valency of a head (the list of arguments that it can combine with). For example, a verb like *put* which takes NP and PP complements (*put the book on the table*) might be represented as (49):

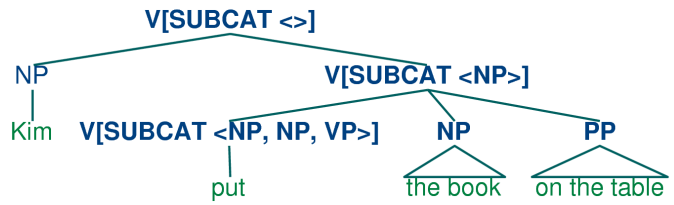
(49) V[SUBCAT NP, NP, PP]

This says that the verb can combine with three arguments. The leftmost element in the list is the subject NP, while everything else — an NP followed by a PP in this case — comprises the subcategorized-for complements. When a verb like *put* is combined with appropriate complements, the requirements which are specified in the SUBCAT are discharged, and only a subject NP is needed. This category, which corresponds to what is traditionally thought of as VP, might be represented as follows.

(50) V[SUBCAT NP]

Finally, a sentence is a kind of verbal category which has *no* requirements for further arguments, and hence has a SUBCAT whose value is the empty list. The tree (51) shows how these category assigments combine in a parse of *Kim put the book on the table*.

(51)

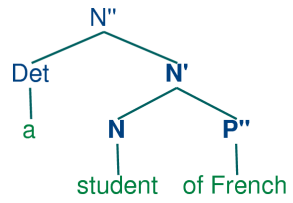


9.4.2 Heads Revisited

We noted in the previous section that by factoring subcategorization information out of the main category label, we could express more generalizations about properties of verbs. Another property of this kind is the following: expressions of category V are heads of phrases of category VP. Similarly (and more informally) Ns are heads of NPs, As (i.e., adjectives) are heads of APs, and Ps (i.e., adjectives) are heads of PPs. Not all phrases have heads — for example, it is standard to say that coordinate phrases (e.g., *the book and the bell*) lack heads — nevertheless, we would like our grammar formalism to express the mother / head-daughter relation where it holds. Now, although it looks as though there is something in common between, say, V and VP, this is more of a handy convention than a real claim, since V and VP formally have no more in common than V and DET.

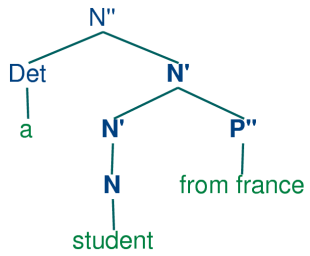
X-bar syntax (cf. [Jacobs & Rosenbaum, 1970], [Jackendoff, 1977]) addresses this issue by abstracting out the notion of **phrasal level**. It is usual to recognise three such levels. If N represents the lexical level, then N' represents the next level up, corresponding to the more traditional category NOM, while N" represents the phrasal level, corresponding to the category NP. (The primes here replace the typographically more demanding horizontal bars of [Jacobs & Rosenbaum, 1970]). (52) illustrates a representative structure.

(52)



The head of the structure (52) is N while N' and N" are called **(phrasal) projections** of N. N" is the **maximal projection**, and N is sometimes called the **zero projection**. One of the central claims of X-bar syntax is that all constituents share a structural similarity. Using X as a variable over N, V, A and P, we say that directly subcategorized *complements* of the head are always placed as sisters of the lexical head, whereas *adjuncts* are placed as sisters of the intermediate category, X'. Thus, the configuration of the P" adjunct in (53) contrasts with that of the complement P" in (52).

(53)



The productions in (54) illustrate how bar levels can be encoded using feature structures.

(54)
```
S → N[BAR 2] V[BAR 2]
N[BAR 2] → DET N[BAR 1]
N[BAR 1] → N[BAR 1] P[BAR 2]
N[BAR 1] → N[BAR 0] P[BAR 2]
```

9.4.3 Auxiliary verbs and Inversion

Inverted clauses — where the order of subject and verb is switched — occur in English interrogatives and also after 'negative' adverbs:

(55a) Do you like children?

(55b) Can Jody walk?

(56a) Rarely do you see Kim.

(56b) Never have I seen this dog.

However, we cannot place just any verb in pre-subject position:

(57a) *Like you children?

(57b) *Walks Jody?

(58a) *Rarely see you Kim.

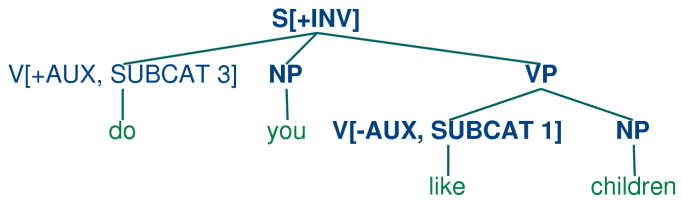
(58b) *Never saw I this dog.

Verbs which can be positioned inititally in inverted clauses belong to the class known as **auxiliaries**, and as well as *do*, *can* and *have* include *be*, *will* and *shall*. One way of capturing such structures is with the following rule:

(59)
```
S[+inv] -> V[+AUX] NP VP
```

That is, a clause marked as [+INV] consists of an auxiliary verb followed by a VP. (In a more detailed grammar, we would need to place some constraints on the form of the VP, depending on the choice of auxiliary.) (60) illustrates the structure of an inverted clause.

(60)



9.4.4 Unbounded Dependency Constructions

Consider the following contrasts:

(61a) You like Jody.

(61b) *You like.

(62a) You put the card into the slot.

(62b) *You put into the slot.

(62c) *You put the card.

(62d) *You put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. Examples (61) and (62) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (63) and (64) illustrate.

(63a) Kim knows who you like.

(63b) This music, you really like.

(64a) Which card do you put into the slot?

(64b) Which slot do you put the card into?

That is, an obligatory complement can be omitted if there is an appropriate **filler** in the sentence, such as the question word *who* in (63a), the preposed topic *this music* in (63b), or the *wh* phrases *which card/slot* in (64). It is common to say that sentences like (63) – (64) contain **gaps** where the obligatory complements have been omitted, and these gaps are sometimes made explicit using an underscore:

(65a) Which card do you put __ into the slot?

(65b) Which slot do you put the card into __?

So, a gap can occur if it is **licensed** by a filler. Conversely, fillers can only occur if there is an appropriate gap elsewhere in the sentence, as shown by the following examples.

(66a) *Kim knows who you like Jody.

(66b) *This music, you really like hip-hop.

(67a) *Which card do you put this into the slot?

(67b) *Which slot do you put the card into this one?

The mutual co-occurence between filler and gap leads to (63) – (64) is sometimes termed a "dependency". One issue of considerable importance in theoretical linguistics has been the nature of the material that can intervene between a filler and the gap that it licenses; in particular, can we simply list a finite set of strings that separate the two? The answer is No: there is no upper bound on the distance between filler and gap. This fact can be easily illustrated with constructions involving sentential complements, as shown in (68).

(68a) Who do you like __?

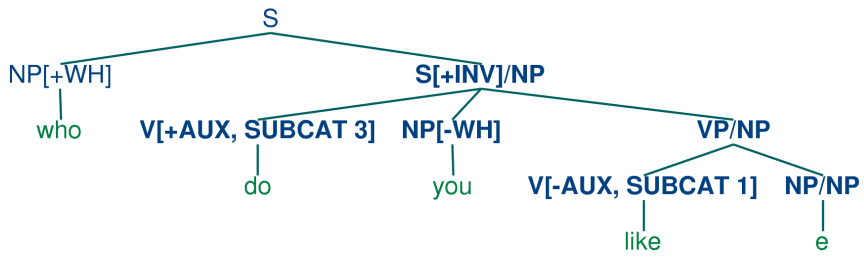
(68b) Who do you claim that you like __?

(68c) Who do you claim that Jody says that you like __?

Since we can have indefinitely deep recursion of sentential complements, the gap can be embedded indefinitely far inside the whole sentence. This constellation of properties leads to the notion of an **unbounded dependency construction**; that is, a filler-gap dependency where there is no upper bound on the distance between filler and gap.

A variety of mechanisms have been suggested for handling unbounded dependencies in formal grammars; we shall adopt an approach due to Generalized Phrase Structure Grammar that involves something called **slash categories**. A slash category is something of the form Y/XP; we interpret this as a phrase of category Y which is somewhere missing a sub-constituent of category XP. For example, S/NP is an S which is missing an NP. The use of slash categories is illustrated in (69).

(69)



The top part of the tree introduces the filler *who* (treated as an expression of category NP[+WH]) together with a corresponding gap-containing constituent S/NP. The gap information is then "percolated" down the tree via the VP/NP category, until it reaches the category NP/NP. At this point, the dependency is discharged by realizing the gap information as the empty string *e* immediately dominated by NP/NP.

Do we need to think of slash categories as a completely new kind of object in our grammars? Fortunately, no, we don't — in fact, we can accommodate them within our existing feature-based framework. We do this by treating slash as a feature, and the category to its right as a value. In other words, our "official" notation for S/NP will be S[SLASH = *NP*]. Once we have taken this step, it is straightforward to write a small grammar in NLTK for analyzing unbounded dependency constructions. (70) illustrates the main principles of slash categories, and also includes rules for inverted clauses. To simplify presentation, we have omitted any specification of tense on the verbs.

(70)

```
% start S
############################
# Grammar Rules
############################
S[-INV] -> NP  S/NP
S[-INV]/?x -> NP VP/?x
S[+INV]/?x -> V[+AUX] NP VP/?x
S-BAR/?x -> Comp S[-INV]/?x

NP/NP ->
```

```
VP/?x -> V[SUBCAT=1, -AUX] NP/?x
VP/?x -> V[SUBCAT=2, -AUX] S-BAR/?x
VP/?x -> V[SUBCAT=3, +AUX] VP/?x

############################
# Lexical Rules
############################
V[SUBCAT=1, -AUX] -> 'see' | 'like'
V[SUBCAT=2, -AUX] -> 'say' | 'claim'
V[SUBCAT=3, +AUX] -> 'do' | 'can'

NP[-WH] -> 'you' | 'children' | 'girls'
NP[+WH] -> 'who'

Comp -> 'that'
```

(70) contains one gap-introduction rule, namely

(71) `S[-INV] → NP S/NP`

In order to percolate the slash feature correctly, we need to add slashes with variable values to both sides of the arrow in rules which expand S, VP and NP. For example,

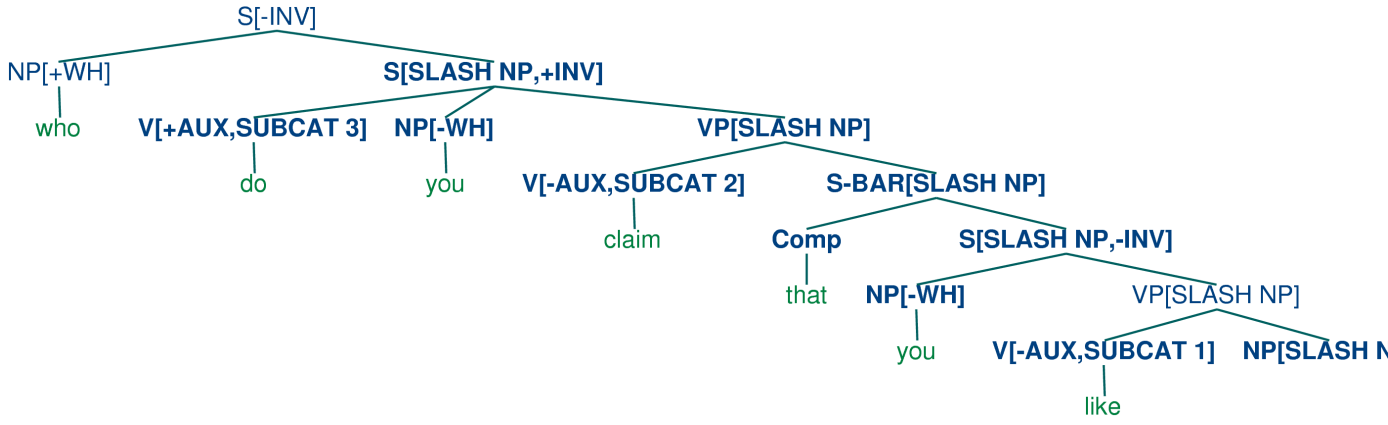
(72) `VP/?X → V S-BAR/?X`

says that a slash value can be specified on the VP mother of a constituent if the same value is also specified on the S-BAR daughter. Finally, (73) allows the slash information on NP to be discharged as the empty string.

(73) `NP/NP →`

Using (70), we can parse the string *who do you claim that you like* into the tree shown in (74).

(74)



9.4.5 Case and Gender in German

Compared with English, German has a relatively rich morphology for agreement. For example, the definite article in German varies with case, gender and number, as shown in Table 9.2.

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 9.2: Morphological Paradigm for the German definite Article

Subjects in German take the nominative case, and most verbs govern their objects in the accusative case. However, there are exceptions like *helfen* which govern the dative case.

(75a)	Die the.NOM.FEM.SG	katze cat.3.FEM.SG	sieht see.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG
	'the cat sees the dog'				
(75b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	sieht see.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
(76a)	Die the.NOM.FEM.SG	katze cat.3.FEM.SG	hilft help.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
	'the cat helps the dog'				
(76b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	hilft help.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG

The grammar (77) illustrates the interaction of agreement (comprising person, number and gender) with case.

(77)

```
% start S
################
# Grammar Rules
################
S -> NP[CASE=nom, AGR=?a] VP[AGR=?a]

NP[CASE=?c, AGR=?a] -> PRO[CASE=?c, AGR=?a]
NP[CASE=?c, AGR=?a] -> Det[CASE=?c, AGR=?a] N[CASE=?c, AGR=?a]

VP[AGR=?a] -> IV[AGR=?a]
VP[AGR=?a] -> TV[OBJCASE=?c, AGR=?a] NP[CASE=?c]

###############
# Lexical Rules
###############
Det[CASE=nom, AGR=[GND=masc,PER=3,NUM=sg]] -> 'der'
Det[CASE=acc, AGR=[GND=masc,PER=3,NUM=sg]] -> 'den'
Det[CASE=dat, AGR=[GND=masc,PER=3,NUM=sg]] -> 'dem'
Det[AGR=[PER=3,NUM=pl]] -> 'die'
```

```
Det[CASE=nom, AGR=[GND=fem,PER=3]] -> 'die'
Det[CASE=acc, AGR=[GND=fem,PER=3]] -> 'die'
Det[CASE=dat, AGR=[GND=fem,PER=3]] -> 'der'

N[AGR=[GND=masc,PER=3,NUM=sg]] -> 'hund'
N[AGR=[GND=masc,PER=3,NUM=pl]] -> 'hunde'
N[AGR=[PER=3,NUM=pl]] -> 'hunde'
N[AGR=[GND=fem,PER=3,NUM=sg]] -> 'katze'
N[AGR=[GND=fem,PER=3,NUM=pl]] -> 'katzen'

PRO[CASE=nom, AGR=[PER=1,NUM=sg]] -> 'ich'
PRO[CASE=acc, AGR=[PER=1,NUM=sg]] -> 'mich'

TV[OBJCASE=acc, AGR=[NUM=sg,PER=1]] -> 'sehe'
TV[OBJCASE=acc, AGR=[NUM=sg,PER=3]] -> 'sieht' | 'mag'
TV[OBJCASE=acc, AGR=[NUM=pl]] -> 'siehen' | 'moegen'
TV[OBJCASE=dat, AGR=[NUM=sg,PER=1]] -> 'folge' | 'helfe'
TV[OBJCASE=dat, AGR=[NUM=sg,PER=3]] -> 'folgt' | 'hilft'
TV[OBJCASE=dat, AGR=[NUM=pl]] -> 'folgen' | 'helfen'

IV[AGR=[NUM=sg,PER=3]] -> 'kommt'
IV[AGR=[NUM=sg,PER=1]] -> 'komme'
IV[AGR=[NUM=pl]] -> 'kommen'
```

As you will see, the feature OBJCASE is used to specify the case which the verb governs on its object.

9.4.6 Exercises

1. ☼ Modify the grammar illustrated in (46) to incorporate a BAR feature for dealing with phrasal projections.

2. ☼ Modify the German grammar in (77) to incorporate the treatment of subcategorization presented in 9.4.1.

3. ◑ Extend the German grammar in (77) so that it can handle so-called verb-second structures like the following:

   (78) Heute sieht der hund die katze.

4. ★ Morphological paradigms are rarely completely regular, in the sense of every cell in the matrix having a different realisation. For example, the present tense conjugation of the lexeme WALK only has two distinct forms: *walks* for the 3rd person singular, and *walk* for all other combinations of person and number. A successful analysis should not require redundantly specifying that 5 out of the 6 possible morphological combinations have the same realization. Propose and implement a method for dealing with this.

5. ★ So-called **head features** are shared between the mother and head daughter. For example, TENSE is a head feature that is shared between a VP and its head V daughter. See [Gazdar et al, 1985] for more details. Most of the features we have looked at are head features — exceptions are SUBCAT and SLASH. Since the sharing of head features is

predictable, it should not need to be stated explicitly in the grammar rules. Develop an approach which automatically accounts for this regular behaviour of head features.

9.5 Summary

- The traditional categories of context-free grammar are atomic symbols. An important motivation feature structures is to capture fine-grained distinctions which would otherwise require a masssive multiplication of atomic categories.
- By using variables over feature values, we can express constraints in grammar rules which allow the realization of different feature specifications to be inter-dependent.
- Typically we specify fixed values of features at the lexical level and constrain the values of features in phrases to unify with the corresponding values in their daughters.
- Feature values are either atomic or complex. A particular subcase of atomic value is the Boolean value, represented by convention as [+/- F].
- Two features can share a value (either atomic or complex). Structures with shared values are said to be re-entrant. Shared values are represented by numerical indices (or tags) in AVMs.
- A path in a feature structure is a tuple of features corresponding to the labels on a sequence of arcs from the root of the graph representation.
- Two paths are equivalent if they share a value.
- Feature structures are partially ordered by subsumption. FS_0 subsumes FS_1 when FS_0 is more general (less informative) than FS_1.
- The unification of two structures FS_0 and FS_1, if successful, is the feature structure FS_2 which contains the combined information of both FS_0 and FS_1.
- If unification specialises a path π in FS, then it also specialises every path π' equivalent to π.
- We can use feature structures to build succinct analyses of a wide variety of linguistic phenomena, including verb subcategorization, inversion constructions, unbounded dependency constructions and case government.

9.6 Further Reading

The earliest use of features in theoretical linguistics was designed to capture phonological properties of phonemes. For example, a sound like **/b/** might be decomposed into the structure [+LABIAL, +VOICE]. An important motivation was to capture generalizations across classes of segments; for example, that **/n/** gets realized as **/m/** preceding any +LABIAL consonant. Within Chomskyan grammar, it was standard to use atomic features for phenomena like agreement, and also to capture generalizations across syntactic categories, by analogy with phonology. A radical expansion of the use of features in theortical syntax was advocated by Generalized Phrase Structure Grammar (GPSG; [Gazdar et al, 1985]), particularly in the use of features with complex values.

Coming more from the perspective of computational linguistics, [Dahl & Saint-Dizier, 1985] proposed that functional aspects of language could be captured by unification of attribute-value structures,

and a similar approach was elaborated by [Grosz & Stickel, 1983] within the PATR-II formalism. Early work in Lexical-Functional grammar (LFG; [Bresnan, 1982]) introduced the notion of an **f-structure** which was primarily intended to represent the grammatical relations and predicate-argument structure associated with a constituent structure parse. [Shieber, 1986] provides an excellent introduction to this phase of research into feature-based grammars.

One conceptual difficulty with algebraic approaches to feature structures arose when researchers attempted to model negation. An alternative perspective, pioneered by [Kasper & Rounds, 1986] and [Johnson, 1988], argues that grammars involve *descriptions* of feature structures rather than the structures themselves. These descriptions are combined using logical operations such as conjunction, and negation is just the usual logical operation over feature descriptions. This description-oriented perspective was integral to LFG from the outset (cf. [Huang & Chen, 1989], and was also adopted by later versions of Head-Driven Phrase Structure Grammar (HPSG; [Sag & Wasow, 1999]).

Feature structures, as presented in this chapter, are unable to capture important constraints on linguistic information. For example, there is no way of saying that the only permissible values for NUM are *sg* and *pl*, while a specifcation such as [NUM *masc*] is anomalous. Similarly, we cannot say that the complex value of AGR *must* contain specifications for the features PER, NUM and GND, but *cannot* contain a specification such as [SUBCAT 3]. **Typed feature structures** were developed to remedy this deficiency. To begin with, we stipulate that feature values are always typed. In the case of atomic values, the values just are types. For example, we would say that the value of NUM is the type *num*. Moreover, *num* is the most general type of value for NUM. Since types are organized hierarchically, we can be more informative by specifying the value of NUM is a **subtype** of *num*, namely either *sg* or *pl*.

In the case of complex values, we say that feature structures are themselves typed. So for example the value of AGR will be a feature structure of type *agr*. We also stipulate that all and only PER, NUM and GND are **appropriate** features for a structure of type *agr*. A good early review of work on typed feature structures is [Emele & Zajac, 1990]. A more comprehensive examination of the formal foundations can be found in [Carpenter, 1992], while [Copestake, 2002] focusses on implementing an HPSG-oriented approach to typed feature structures.

There is a copious literature on the analysis of German within feature-based grammar frameworks. [Nerbonne, Netter, & Pollard, 1994] is a good starting point for the HPSG literature on this topic, while [M\"{u}ller, 1999] gives a very extensive and detailed analysis of German syntax in HPSG.