9   Building Feature Based Grammars

Natural languages have an extensive range of grammatical constructions which are hard to handle with the simple methods described in Chapter 8. In order to gain more flexibility, we change our treatment of grammatical categories like S, NP and V. In place of atomic labels, we decompose them into structures like dictionaries, where features can take on a range of values.

The goal of this chapter is to answer the following questions:

1. How can we extend the framework of context free grammars with features so as to gain more fine-grained control over grammatical categories and productions?
2. What are the main formal properties of feature structures and how do we use them computationally?
3. What kinds of linguistic patterns and grammatical constructions can we now capture with feature based grammars?

Along the way, we will cover more topics in English syntax, including phenomena such as agreement, subcategorization, and unbounded dependency constructions.

9.1   Grammatical Features

In chap-data-intensive, we described how to build classifiers that rely on detecting features of text. Such features may be quite simple, such as extracting the last letter of a word, or more complex, such as a part-of-speech tag which has itself been predicted by the classifier. In this chapter, we will investigate the role of features in building rule-based grammars. In contrast to feature extractors, which record features that have been automatically detected, we are now going to *declare* the features of words and phrases. We start off with a very simple example, using dictionaries to store features and their values.

```
>>> kim = {'CAT': 'NP', 'ORTH': 'Kim', 'REF': 'k'}
>>> chase = {'CAT': 'V', 'ORTH': 'chased', 'REL': 'chase'}
```

The objects kim and chase both have a couple of shared features, CAT (grammatical category) and ORTH (orthography, i.e., spelling). In addition, each has a more semantically-oriented feature: kim['REF'] is intended to give the referent of kim, while chase['REL'] gives the relation expressed by chase. In the context of rule-based grammars, such pairings of features and values are known as **feature structures**, and we will shortly see alternative notations for them.

Feature structures contain various kinds of information about grammatical entities. The information need not be exhaustive, and we might want to add further properties. For example, in the case of a verb, it is often useful to know what "semantic role" is played by the arguments of the verb. In the case of *chase*, the subject plays the role of "agent", while the object has the role of "patient". Let's add this information, using 'sbj' and 'obj' as placeholders which will get filled once the verb combines with its grammatical arguments:

```
>>> chase['AGT'] = 'sbj'
>>> chase['PAT'] = 'obj'
```

If we now process a sentence *Kim chased Lee*, we want to "bind" the verb's agent role to the subject and the patient role to the object. We do this by linking to the REF feature of the relevant NP. In the following example, we make the simple-minded assumption that the NPs immediately to the left and right of the verb are the subject and object respectively. We also add a feature structure for *Lee* to complete the example.

```
>>> sent = "Kim chased Lee"
>>> tokens = sent.split()
>>> lee = {'CAT': 'NP', 'ORTH': 'Lee', 'REF': 'l'}
>>> def lex2fs(word):
...     for fs in [kim, lee, chase]:
...         if fs['ORTH'] == word:
...             return fs
>>> subj, verb, obj = lex2fs(tokens[0]), lex2fs(tokens[1]), lex2fs(tokens[2])
>>> verb['AGT'] = subj['REF']
>>> verb['PAT'] = obj['REF']
>>> for k in ['ORTH', 'REL', 'AGT', 'PAT']:
...     print "%-5s => %s" % (k, verb[k])
ORTH  => chased
REL   => chase
AGT   => k
PAT   => l
```

The same approach could be adopted for a different verb, say *surprise*, though in this case, the subject would play the role of "source" (SRC) and the object, the role of "experiencer" (EXP):

```
>>> surprise = {'CAT': 'V', 'ORTH': 'surprised', 'REL': 'surprise',
...             'SRC': 'sbj', 'EXP': 'obj'}
```

Feature structures are pretty powerful, but the way in which we have manipulated them is extremely *ad hoc*. Our next task in this chapter is to

show how the framework of context free grammar and parsing can be expanded to accommodate feature structures, so that we can build analyses like this in a more generic and principled way. 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 their use in a simple grammar.

Since 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 illustrate the support for feature structures offered by NLTK. In the final part of the chapter, we demonstrate that the additional expressiveness of features opens up a wide spectrum of possibilities for describing sophisticated aspects of linguistic structure.

Syntactic Agreement

The following examples show pairs of word sequences, the first of which is grammatical and the second not. (We use an asterisk at the start of a word sequence to signal that is ungrammatical.)

(1)
    a.   this dog

    b.   *these dog

(2)
    a.   these dogs

    b.   *this dogs

In English, nouns are usually marked as being singular or plural. The form of the demonstrative also varies: *this* (singular) and *these* (plural). Examples (1b) and (2b) show that there are constraints on the use of demonstratives and nouns within a noun phrase: either both are singular or both are plural. A similar constraint holds between subjects and predicates:

(3)
    a.   the dog runs

    b.   *the dog run

(4)
    a.   the dogs run

    b.   *the dogs runs

Here we can see that morphological properties of the verb co-vary with syntactic properties of the subject noun phrase. This co-variance is called **agreement**. 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, as shown in Table 9.1.

Table 9.1:

Agreement Paradigm for English Regular Verbs

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

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.)

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

(6)   the   dog-s   run

```
       dog.3.PL      run-3.PL
```

Let's see what happens when we encode these agreement constraints in a context-free grammar. We will begin with the simple CFG in (7).

```
(7)  S    ->    NP VP
     NP   ->    Det N
     VP   ->    V

     Det  ->    'this'
     N    ->    'dog'
     V    ->    'runs'
```

Grammar (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 sequences like **these dogs run* and **these dog runs*. The most straightforward approach is to add new non-terminals and productions to the grammar:

```
(8)  S -> NP_SG VP_SG
     S -> 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'
```

In place of a single production expanding S, we now have two productions, one covering the sentences involving singular subject NPs and VPs, the other covering sentences with plural subject NPs and VPs. In fact, every production in (7) has two counterparts in (8). With a small grammar, this is not really such a problem, although it is aesthetically unappealing. However, with a larger grammar that covers a reasonable subset of English constructions, the prospect of doubling the grammar size is very unattractive. Let's suppose now that we used the same approach to deal with first, second and third person agreement, for both singular and plural. This would lead to the original grammar being multiplied by a factor of 6, which we definitely want to avoid. Can we do better than this? In the next section we will show that capturing number and person agreement need not come at the cost of "blowing up" the number of productions.

Using Attributes and Constraints

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

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

In (9), we have introduced some new notation which says that the category N has a (grammatical) **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:

```
(11) S -> NP[NUM=?n] VP[NUM=?n]
     NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
     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, within a given production. We can read the first production 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 productions will admit the following local trees (trees of depth one):

(12)
      a.

Det[NUM=sg]
|
this

b. Det[NUM=pl]
|
these

(13)

a. **N[NUM=sg]**
|
dog

b. **N[NUM=pl]**
|
dogs

Now `S -> NP[NUM=?n] VP[NUM=?n]` says that whatever the NUM values of N and Det are, they have to be the same. Consequently, `NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]` 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 subtrees differ in their values for the NUM feature; this incompatibility of values is indicated informally with a *FAIL* value at the top node.

(14)

a. NP[NUM=sg]
/ \
Det[NUM=sg]   **N[NUM=sg]**
| |
this   dog

b. **NP[NUM=pl]**
/ \
Det[NUM=pl]   **N[NUM=pl]**
| |
these   dogs

(15)

a. **NP[NUM=** *FAIL* **]**
/ \
Det[NUM=sg]   **N[NUM=pl]**
| |
this   dogs

b. **NP[NUM=** *FAIL* **]**
/ \
Det[NUM=pl]   **N[NUM=sg]**
| |
these   dog

Production `VP[NUM=?n] -> V[NUM=?n]` says that the NUM value of the head verb has to be the same as the NUM value of the VP parent. Combined with the production for expanding S, 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)
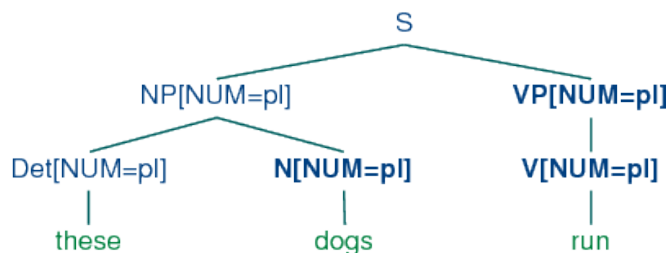
Grammar (10) illustrated lexical productions for determiners like *this* and *these* which require a singular or plural head noun respectively. However, other determiners in English are not choosy about the grammatical number of the noun they combine with. One way of describing this would be to add two lexical entries to the grammar, one each for the singular and plural versions of determiner such as *the*

```
Det[NUM=sg] -> 'the' | 'some' | 'several'
Det[NUM=pl] -> 'the' | 'some' | 'several'
```

However, a more elegant solution is to leave the NUM value **underspecified** and letting it agree in number with whatever noun it combines with. Assigning a variable value to NUM is one way of achieving this result:

```
Det[NUM=?n] -> 'the' | 'some' | 'several'
```

But in fact we can be even more economical, and just omit any specification for NUM in such productions. We only need to explicitly enter a variable value when this constrains another value elsewhere in the same production.

The grammar in Example 9.1 illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones.

```
>>> nltk.data.show_cfg('grammars/book_grammars/feat0.fcfg')
% start S
# ###################
# Grammar Productions
# ###################
# S expansion productions
S -> NP[NUM=?n] VP[NUM=?n]
# NP expansion productions
NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]
# VP expansion productions
VP[TENSE=?t, NUM=?n] -> IV[TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> TV[TENSE=?t, NUM=?n] NP
# ###################
# Lexical 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'
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] -> 'disappeared' | 'walked'
TV[TENSE=past] -> 'saw' | 'liked'
```

Example 9.1 (code_feat0cfg.py): **Figure 9.1**: Example Feature based Grammar

Notice that a syntactic category can have more than one feature; for example, V[TENSE=pres, NUM=pl]. In general, we can add as many features as we like.

A final detail about Example 9.1 is the statement %start S. This "directive" 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 productions in a file where they can be edited, tested and revised. We have saved Example 9.1 as a file named 'feat0.fcfg' in the NLTK data distribution. You can make your own copy of this for further experimentation using nltk.data.load().

Feature based grammars are parsed in NLTK using an Earley chart parser (see Section 9.5 for more information about this). After tokenizing the input, we import the load_parser function [1] which takes a grammar filename as input and returns a chart parser cp [2]. Calling the parser's nbest_parse() method will return a list trees of parse trees; trees will be empty if the grammar fails to parse the input and will contain one or more parse trees, depending on whether the input is syntactically ambiguous or not.

```
>>> tokens = 'Kim likes children'.split()
>>> from nltk import load_parser [1]
>>> cp = load_parser('grammars/book_grammars/feat0.fcfg', trace=2)  [2]
>>> trees = cp.nbest_parse(tokens)
|.Kim .like.chil.|
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:
|[---->    .    .| [0:1] S[] -> NP[NUM=?n] * VP[NUM=?n] {?n: 'sg'}
Feature Bottom Up Predict Combine Rule:
|.    [----]    .| [1:2] TV[NUM='sg', TENSE='pres'] -> 'likes' *
Feature Bottom Up Predict Combine Rule:
|.    [---->    .| [1:2] VP[NUM=?n, TENSE=?t] -> TV[NUM=?n, TENSE=?t] * NP[] {?n: 'sg', ?t: 'pres'}
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:
|.    .    [---->| [2:3] S[] -> NP[NUM=?n] * VP[NUM=?n] {?n: 'pl'}
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'] *
```
Example 9.2 (code_featurecharttrace.py): **Figure 9.2**: Trace of Feature based Chart Parser

The details of the parsing procedure are not that important for present purposes. However, there is an implementation issue which bears on our earlier discussion of grammar size. One possible approach to parsing productions containing feature constraints is to compile out all admissible values of the features in question so that we end up with a large, fully specified CFG along the lines of (8). By contrast, the parser process illustrated above works directly with the underspecified productions given by the grammar. Feature values "flow upwards" from lexical entries, and variable values are then associated with those values, via bindings (i.e., dictionaries) such as {?n: 'sg', ?t: 'pres'}. As the parser assembles information about the nodes of the tree it is building, these variable bindings are used to instantiate values in these nodes; thus the underspecified VP[NUM=?n, TENSE=?t] -> TV[NUM=?n, TENSE=?t] NP[] becomes instantiated as VP[NUM='sg', TENSE='pres'] -> TV[NUM='sg', TENSE='pres'] NP[] by looking up the values of ?n and ?t in the bindings.

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

```
>>> for tree in trees: print tree
(S[]
  (NP[NUM='sg'] (PropN[NUM='sg'] Kim))
  (VP[NUM='sg', TENSE='pres']
    (TV[NUM='sg', TENSE='pres'] likes)
    (NP[NUM='pl'] (N[NUM='pl'] children))))
```

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 that just specify whether a property is true or false. For example, we might want to distinguish **auxiliary** verbs such as *can*, *may*, *will* and *do* with the boolean feature AUX. For example, the production V[TENSE=pres, aux=+] -> 'can' means that *can* receives the value pres for TENSE and + or true for AUX. There is a widely adopted convention which abbreviates the representation of boolean features f; instead of aux=+ or aux=-, we use +aux and -aux respectively. These are just abbreviations, however, and the parser interprets them as though + and - are like any other atomic value. (17) shows some representative productions:

(17) V[TENSE=pres, +aux] -> 'can'
      V[TENSE=pres, +aux] -> 'may'

      V[TENSE=pres, -aux] -> 'walks'
      V[TENSE=pres, -aux] -> 'likes'

We have spoken of attaching "feature annotations" to syntactic categories. A more radical approach represents the whole category — that is, the non-terminal symbol plus the annotation — as a bundle of features. For example, N[NUM=sg] contains part of speech information which can be represented as POS=N. An alternative notation for this category therefore is [POS=N, NUM=sg].

In addition to atomic-valued features, features may take values that are themselves feature structures. For example, we can group together agreement features (e.g., person, number and gender) as a distinguished part of a category, grouped together as the value of AGR. In this case, we say that AGR has a **complex** value. (18) depicts the structure, in a format known as an **attribute value matrix** (AVM).

(18) [POS = N   ]
      [   ]
      [AGR = [PER = 3   ]]
      [[NUM = pl]]
      [[GND = fem]]

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

Figure 9.3: Rendering a Feature Structure as an Attribute Value Matrix

In passing, we should point out that there are alternative approaches for displaying AVMs; Figure 9.3 shows an example. Athough feature structures rendered in the style of (18) are less visually pleasing, we will stick with this format, since it corresponds to the output we will be getting from NLTK.

On the topic of representation, we also note that feature structures, like dictionaries, assign no particular significance to the *order* of features. So (18) is equivalent to:

```
(19)   [AGR = [NUM = pl  ]]
       [      [PER = 3   ]]
       [      [GND = fem ]]
       [                  ]
       [POS = N           ]
```

Once we have the possibility of using features like AGR, we can refactor a grammar like Example 9.1 so that agreement features are bundled together. A tiny grammar illustrating this idea is shown in (20).

```
(20)   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   Processing 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.

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

```
>>> fs1 = nltk.FeatStruct(TENSE='past', NUM='sg')
>>> print fs1
[ NUM   = 'sg'   ]
[ TENSE = 'past' ]
```

A feature structure is actually just a kind of dictionary, and so we access its values by indexing in the usual way. We can use our familiar syntax to *assign* values to features:

```
>>> fs1 = nltk.FeatStruct(PER=3, NUM='pl', GND='fem')
>>> print fs1['GND']
fem
>>> fs1['CASE'] = 'acc'
```

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

```
>>> fs2 = nltk.FeatStruct(POS='N', AGR=fs1)
>>> print fs2
[       [ CASE = 'acc' ] ]
[ AGR = [ GND  = 'fem' ] ]
[       [ NUM  = 'pl'  ] ]
[       [ PER  = 3     ] ]
[                        ]
[ POS = 'N'             ]
>>> print fs2['AGR']
[ CASE = 'acc' ]
[ GND  = 'fem' ]
[ NUM  = 'pl'  ]
[ PER  = 3     ]
>>> print fs2['AGR']['PER']
3
```

An alternative method of specifying feature structures is to use a bracketed string consisting of feature-value pairs in the format `feature=value`, where values may themselves be feature structures:

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

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:

```
>>> print nltk.FeatStruct(name='Lee', telno='01 27 86 42 96', age=33)
[ age   = 33                ]
[ name  = 'Lee'             ]
[ telno = '01 27 86 42 96' ]
```

In the next couple of pages, we are going to use examples like this to explore standard operations over feature structures. This will briefly divert us from processing natural language, but we need to lay the groundwork before we can get back to talking about grammars. Hang on tight!

It is often helpful to view feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (21) is equivalent to the above AVM.

(21)
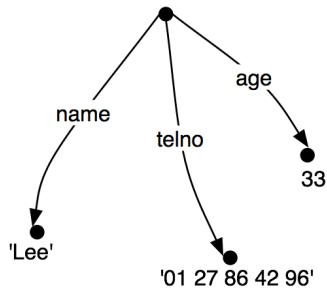
The feature names appear as labels on the directed arcs, and feature values appear as labels on the nodes that are pointed to by the arcs.

Just as before, feature values can be complex:

(22)
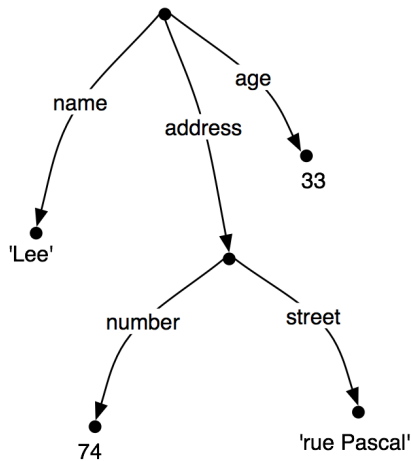
When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths as tuples. Thus, `('address', 'street')` is a feature path whose value in (22) is the sequence "rue Pascal".

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 (23).

(23)
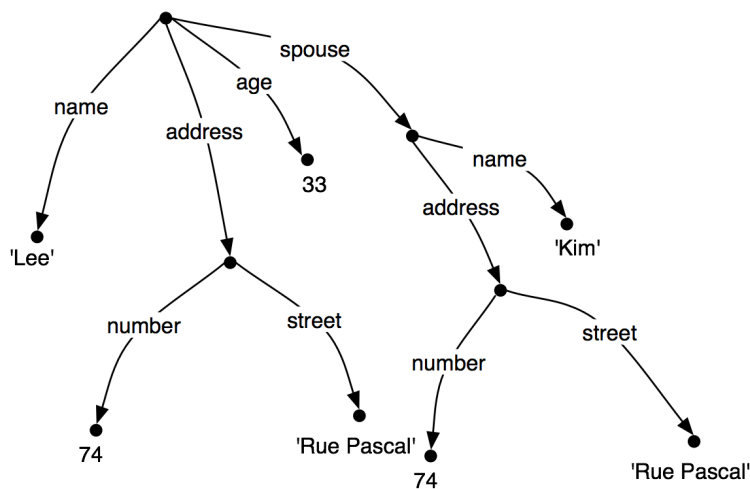
However, rather than repeating the address information in the feature structure, we can "share" the same sub-graph between different arcs:

(24)

In other words, the value of the path (`'ADDRESS'`) in (24) is identical to the value of the path (`'SPOUSE'`, `'ADDRESS'`). DAGs such as (24) are said to involve **structure sharing** or **reentrancy**. When two paths have the same value, they are said to be **equivalent**.
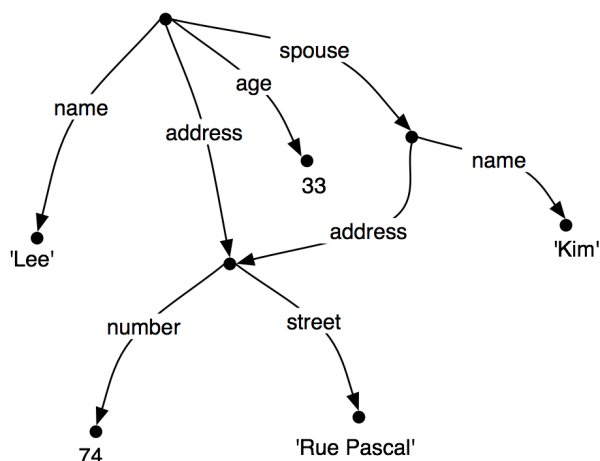
In order to indicate reentrancy in our matrix-style representations, we will prefix the first occurrence of a shared feature structure with an integer in parentheses, such as (1). Any later reference to that structure will use the notation ->(1), as shown below.

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

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.

```
>>> print nltk.FeatStruct("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
[ A = 'a'           ]
[                   ]
[ B = (1) [ C = 'c' ] ]
[                   ]
[ D -> (1)          ]
[ E -> (1)          ]
```

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, (25a) is more general (less specific) than (25b), which in turn is more general than (25c).

(25)

a. [NUMBER = 74]

b. ```
[NUMBER = 74   ]
[street = 'rue Pascal']
```

c. ```
[NUMBER = 74           ]
[STREET = 'rue Pascal']
[CITY = 'Paris'        ]
```

This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write $FS_0 \sqsubseteq FS_1$), then FS_1 must have all the paths and path equivalences of FS_0, and may have additional paths and equivalences as well. Thus, (23) subsumes (24), 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, (26) neither subsumes nor is subsumed by (25a).

(26) `[TELNO = 01 27 86 42 96]`

So we have seen that some feature structures are more specific than others. How do we go about specializing 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 (27b) with (27a) to yield (27c).

(27)

a.

b.

c.
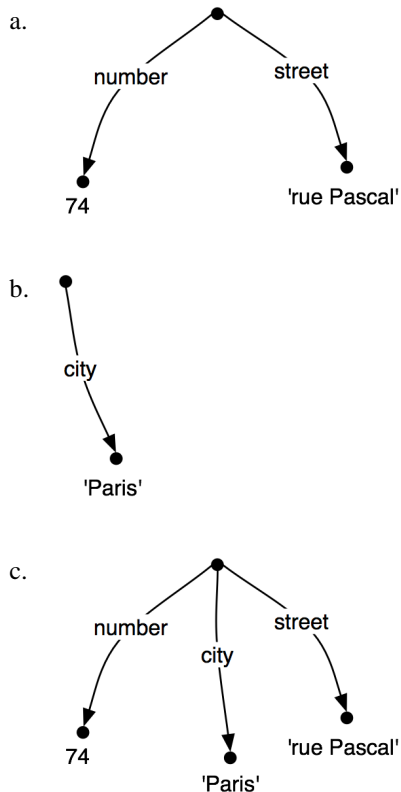
Merging information from two feature structures is called **unification** and is supported by the `unify()` method.

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

Unification is formally defined as a binary operation: $FS_0 \sqcup FS_1$. Unification is symmetric, so $FS_0 \sqcup FS_1 = FS_1 \sqcup FS_0$. The same is true in Python:

```
>>> 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:

(28)   If $FS_0 \sqsubseteq FS_1$, then $FS_0 \sqcap FS_1 = FS_1$

For example, the result of unifying (25b) with (25c) is (25c).

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. This is implemented by setting the result of unification to be `None`.

```
>>> fs0 = nltk.FeatStruct(A='a')
>>> fs1 = nltk.FeatStruct(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 (23) in Python:

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

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

```
>>> fs1 = nltk.FeatStruct("[SPOUSE = [ADDRESS = [CITY = Paris]]]")
>>> print fs1.unify(fs0)
[ ADDRESS = [ NUMBER = 74            ]                    ]
[           [ STREET = 'rue Pascal' ]                    ]
[                                                        ]
[ NAME     = 'Lee'                                       ]
[                                                        ]
[           [            [ CITY    = 'Paris'     ] ] ]
[           [ ADDRESS = [ NUMBER = 74            ] ] ]
[ SPOUSE   = [           [ STREET = 'rue Pascal' ] ] ]
[           [                                    ] ]
[           [ NAME     = 'Kim'                   ] ]
```

By contrast, the result is very different if `fs1` is unified with the structure-sharing version `fs2` (also shown earlier as the graph (24)):

```
>>> fs2 = nltk.FeatStruct("""[NAME=Lee, ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
...                           SPOUSE=[NAME=Kim, ADDRESS->(1)]]""")
>>> print fs1.unify(fs2)
[             [ CITY    = 'Paris'     ] ]
[ ADDRESS = (1) [ NUMBER = 74         ] ]
[             [ STREET = 'rue Pascal' ] ]
[                                       ]
[ NAME     = 'Lee'                      ]
[                                       ]
[ SPOUSE   = [ ADDRESS -> (1)   ]       ]
[           [ NAME     = 'Kim' ]        ]
```

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 specializing the value of some path π, then that unification simultaneously specializes the value of *any path that is equivalent to π*.

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

```
>>> fs1 = nltk.FeatStruct("[ADDRESS1=[NUMBER=74, STREET='rue Pascal']]")
>>> fs2 = nltk.FeatStruct("[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   Extending a Feature based Grammar

In this section, we return to feature based grammar and explore a variety of linguistic issues, and demonstrate the benefits of incorporating features into the grammar.

Subcategorization

In [Chapter 8](#), we augmented our category labels to represent different kinds of verb, and used the labels IV and TV for intransitive and transitive verbs respectively. This allowed us to write productions like the following:

(29)   ```
       VP -> IV
       VP -> TV NP
       ```

Although we know that IV and TV are two kinds of V, they are just atomic nonterminal symbols from a CFG, as distinct from each other as any other pair of symbols. This notation doesn't let us say anything about verbs in general, e.g. we cannot say "All lexical items of category V can be marked for tense", since *walk*, say, is an item of category IV, not V. So, can we replace category labels such as TV and IV by V along with a feature that tells us whether the verb combines with a following NP object or whether it can occur without any complement?

A simple approach, originally developed for a grammar framework called Generalized Phrase Structure Grammar (GPSG), tries to solve this problem by allowing lexical categories to bear a SUBCAT which tells us what subcategorization class the item belongs to. While GPSG used integer values for SUBCAT, the example below adopts more mnemonic values, namely intrans, trans and clause:

(30)   ```
       VP[TENSE=?t, NUM=?n] -> V[SUBCAT=intrans, TENSE=?t, NUM=?n]
       VP[TENSE=?t, NUM=?n] -> V[SUBCAT=trans, TENSE=?t, NUM=?n] NP
       VP[TENSE=?t, NUM=?n] -> V[SUBCAT=clause, TENSE=?t, NUM=?n] SBar

       V[SUBCAT=intrans, TENSE=pres, NUM=sg] -> 'disappears' | 'walks'
       V[SUBCAT=trans, TENSE=pres, NUM=sg] -> 'sees' | 'likes'
       V[SUBCAT=clause, TENSE=pres, NUM=sg] -> 'says' | 'claims'

       V[SUBCAT=intrans, TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
       V[SUBCAT=trans, TENSE=pres, NUM=pl] -> 'see' | 'like'
       V[SUBCAT=clause, TENSE=pres, NUM=pl] -> 'say' | 'claim'

       V[SUBCAT=intrans, TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
       V[SUBCAT=trans, TENSE=past, NUM=?n] -> 'saw' | 'liked'
       V[SUBCAT=clause, TENSE=past, NUM=?n] -> 'said' | 'claimed'
       ```

When we see a lexical category like V[SUBCAT=trans], we can interpret the SUBCAT specification as a pointer to a production in which V[SUBCAT=trans] is introduced as the head child in a VP production. By convention, there is a correspondence between the values of SUBCAT and the productions that introduce lexical heads. On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP. As required, *walk* and *like* both belong to the category V. Nevertheless, *walk* will only occur in VPs expanded by a production with the feature SUBCAT=intrans on the right hand side, as opposed to *like*, which requires a SUBCAT=trans.
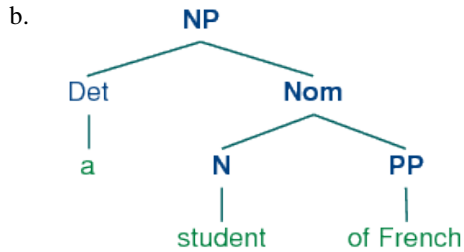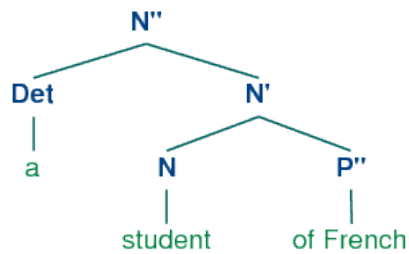
In our third class of verbs above, we have specified a category SBar. This is a label for subordinate clauses such as the complement of *claim* in the example *You claim that you like children*. We require two further productions to analyze such sentences:

(31)   ```
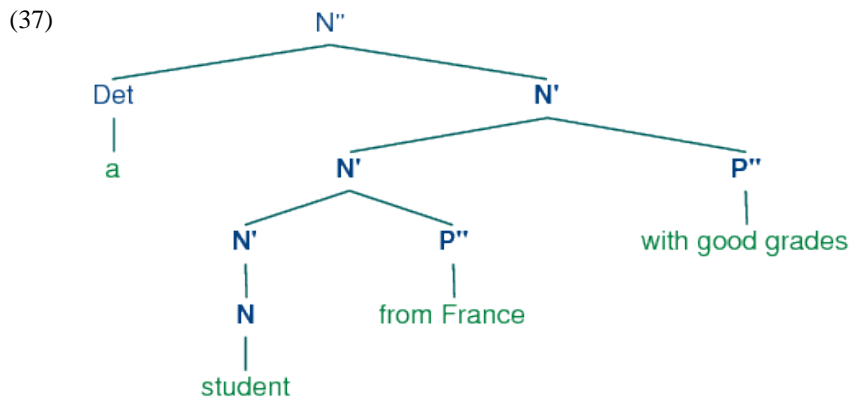   SBar -> Comp S
       Comp -> 'that'
       ```

The resulting structure is the following.

(32)

S
NP  VP
you  V[-AUX, SUBCAT=clause]  SBar
claim  Comp  S
that  NP  VP
you  V[-AUX, SUBCAT=trans]  NP
like  children

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 productions, 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* that takes NP and PP complements (*put the book on the table*) might be represented as (33):

(33)   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.

(34)   V[SUBCAT=<NP>]

Finally, a sentence is a kind of verbal category that has *no* requirements for further arguments, and hence has a SUBCAT whose value is the empty list. The tree (35) shows how these category assignments combine in a parse of *Kim put the book on the table*.

(35)
V[SUBCAT=<>]
NP  V[SUBCAT=<NP>]
Kim  V[SUBCAT=<NP, NP, PP>]  NP  PP
put  the book  on the table

## 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, Ns are heads of NPs, As (i.e., adjectives) are heads of APs, and Ps (i.e., prepositions) 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 parent / head-child relation where it holds. At present, V and VP are just atomic symbols, and we need to find a way to relate them using features (as we did earlier to relate IV and TV).

X-bar Syntax addresses this issue by abstracting out the notion of **phrasal level**. It is usual to recognize 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. (36a) illustrates a representative structure while (36b) is the more conventional counterpart.

(36)
    a.

The head of the structure (36a) 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 a lexical head X are always placed as siblings of the head, whereas *adjuncts* are placed as siblings of the intermediate category, X'. Thus, the configuration of the two P'' adjuncts in (37) contrasts with that of the complement P'' in (36a).

(37)

The productions in (38) illustrate how bar levels can be encoded using feature structures. The nested structure in (37) is achieved by two applications of the recursive rule expanding N[BAR=1].

(38)
```
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]
```

## Auxiliary Verbs and Inversion

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

(39)

    a.   Do you like children?

    b.   Can Jody walk?

(40)

    a.   Rarely do you see Kim.

b. Never have I seen this dog.

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

(41)
　　a. *Like you children?

　　b. *Walks Jody?

(42)
　　a. *Rarely see you Kim.

　　b. *Never saw I this dog.

Verbs that can be positioned initially 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 production:

(43)　`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.) (44) illustrates the structure of an inverted clause.

(44)

```
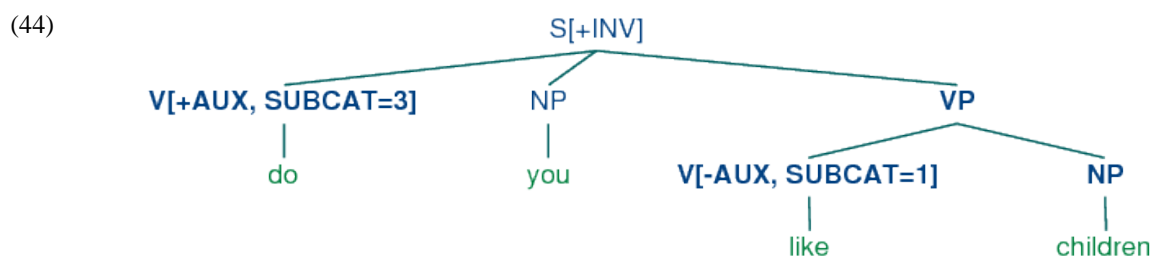                          S[+INV]
           _____|_____
          |                |                         |
  V[+AUX, SUBCAT=3]        NP                        VP
          |                |              _____|_____
          do              you           |                       |
                                 V[-AUX, SUBCAT=1]              NP
                                        |                       |
                                       like                  children
```

Unbounded Dependency Constructions

Consider the following contrasts:

(45)
　　a. You like Jody.

　　b. *You like.

(46)
　　a. You put the card into the slot.

　　b. *You put into the slot.

　　c. *You put the card.

　　d. *You put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. (45) and (46) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (47) and (48)

illustrate.

(47)

    a.   Kim knows who you like.

    b.   This music, you really like.

(48)

    a.   Which card do you put into the slot?

    b.   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 [(47a)](), the preposed topic *this music* in [(47b)](), or the *wh* phrases *which card/slot* in [(48)](). It is common to say that sentences like [(47)]() – [(48)]() contain **gaps** where the obligatory complements have been omitted, and these gaps are sometimes made explicit using an underscore:

(49)

    a.   Which card do you put __ into the slot?

    b.   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.

(50)

    a.   *Kim knows who you like Jody.

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

(51)

    a.   *Which card do you put this into the slot?

    b.   *Which slot do you put the card into this one?

The mutual co-occurence between filler and gap 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 sequences 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 [(52)]().

(52)

    a.   Who do you like __?

    b.   Who do you claim that you like __?

    c.   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; here we illustrate the approach due to Generalized Phrase Structure Grammar that involves **slash categories**. A slash category has the form Y/XP; we interpret this as a phrase of

category Y that is missing a sub-constituent of category XP. For example, S/NP is an S that is missing an NP. The use of slash categories is illustrated in (53).

(53)

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, immediately dominated by NP/NP.

Do we need to think of slash categories as a completely new kind of object? Fortunately, we can accommodate them within our existing feature based framework, by treating slash as a feature, and the category to its right as a value; that is, S/NP is reducible S[SLASH=NP]. In practice, this is also how the parser interprets slassh categories.

The grammar show in Example 9.4 illustrates the main principles of slash categories, and also includes productions for inverted clauses. To simplify presentation, we have omitted any specification of tense on the verbs.

```
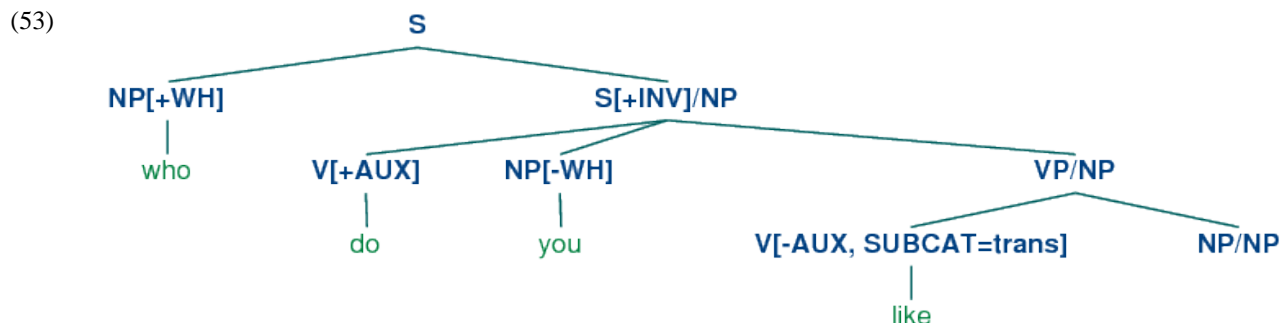>>> nltk.data.show_cfg('grammars/book_grammars/feat1.fcfg')
% start S
# ###################
# Grammar Productions
# ###################
S[-INV] -> NP VP
S[-INV]/?x -> NP VP/?x
S[-INV] -> NP S/NP
S[-INV] -> Adv[+NEG] S[+INV]
S[+INV] -> V[+AUX] NP VP
S[+INV]/?x -> V[+AUX] NP VP/?x
SBar -> Comp S[-INV]
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
# ###################
# Lexical Productions
# ###################
V[SUBCAT=intrans, -AUX] -> 'walk' | 'sing'
V[SUBCAT=trans, -AUX] -> 'see' | 'like'
V[SUBCAT=clause, -AUX] -> 'say' | 'claim'
V[+AUX] -> 'do' | 'can'
NP[-WH] -> 'you' | 'cats'
NP[+WH] -> 'who'
Adv[+NEG] -> 'rarely' | 'never'
NP/NP ->
Comp -> 'that'
```

Example 9.4 (code_slashcfg.py): **Figure 9.4**: Grammar with productions for inverted clauses and long-distance dependencies, making use of slash categories

The grammar in Example 9.4 contains one "gap-introduction" production, namely 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 productions that expand S, VP and NP. For example, VP/?x -> V SBar/?x is the slashed version of VP -> V SBar and says that a slash value can be specified on the VP parent of a constituent if the same value is also specified on the SBar child. Finally, NP/NP -> allows the slash information on NP to be discharged as the empty string. Using Example 9.4, we can parse the sequence *who do you claim that you like*

```
>>> tokens = 'who do you claim that you like'.split()
>>> from nltk import load_parser
>>> cp = load_parser('grammars/book_grammars/feat1.fcfg')
>>> for tree in cp.nbest_parse(tokens):
...         print tree
(S[-INV]
  (NP[+WH] who)
  (S[+INV]/NP[]
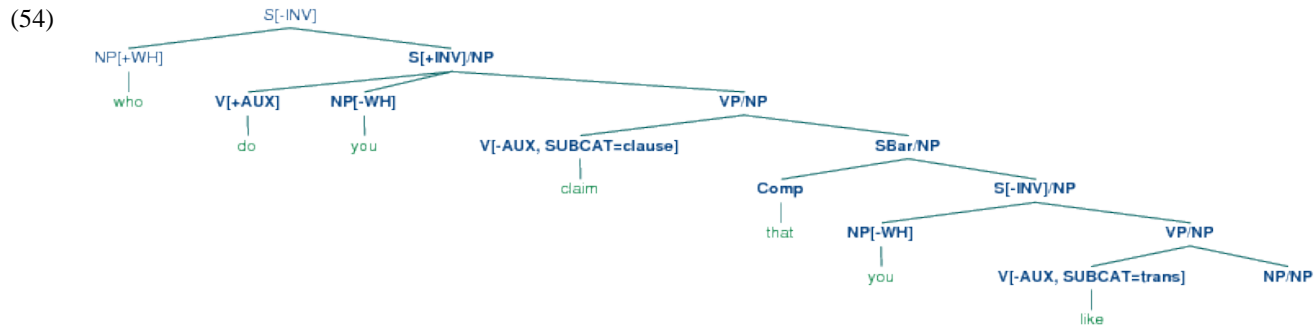```

```
      (V[+AUX] do)
      (NP[-WH] you)
      (VP[]/NP[]
        (V[-AUX, SUBCAT='clause'] claim)
        (SBar[]/NP[]
          (Comp[] that)
          (S[-INV]/NP[]
            (NP[-WH] you)
            (VP[]/NP[] (V[-AUX, SUBCAT='trans'] like) (NP[]/NP[] )))))))
```

A more readable version of this tree is shown in (54).

(54)

The grammar in Example 9.4 will also allow us to parse sentences without gaps:

```
>>> tokens = 'you claim that you like cats'.split()
>>> for tree in cp.nbest_parse(tokens):
...       print tree
(S[-INV]
  (NP[-WH] you)
  (VP[]
    (V[-AUX, SUBCAT='clause'] claim)
    (SBar[]
      (Comp[] that)
      (S[-INV]
        (NP[-WH] you)
        (VP[] (V[-AUX, SUBCAT='trans'] like) (NP[-WH] cats))))))
```

In addition, it admits inverted sentences which do not involve *wh* constructions:

```
>>> tokens = 'rarely do you sing'.split()
>>> for tree in cp.nbest_parse(tokens):
...       print tree
(S[-INV]
  (Adv[+NEG] rarely)
  (S[+INV]
    (V[+AUX] do)
    (NP[-WH] you)
    (VP[] (V[-AUX, SUBCAT='intrans'] sing))))
```

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.

Table 9.2:

Morphological Paradigm for the German definite Article

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

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:

(55)

   a.   Die   Katze   sieht   den   Hund

	the.NOM.FEM.SG	cat.3.FEM.SG	see.3.SG	the.ACC.MASC.SG	dog.3.MASC.SG
	'the cat sees the dog'				
b.	*Die the.NOM.FEM.SG	Katze cat.3.FEM.SG	sieht see.3.SG	dem the.DAT.MASC.SG	Hund dog.3.MASC.SG
c.	Die the.NOM.FEM.SG 'the cat helps the dog'	Katze cat.3.FEM.SG	hilft help.3.SG	dem the.DAT.MASC.SG	Hund dog.3.MASC.SG
d.	*Die the.NOM.FEM.SG	Katze cat.3.FEM.SG	hilft help.3.SG	den the.ACC.MASC.SG	Hund dog.3.MASC.SG

The grammar in Example 9.5 illustrates the interaction of agreement (comprising person, number and gender) with case.

```
>>> nltk.data.show_cfg('grammars/book_grammars/german.fcfg')
% start S
 # Grammar Productions
 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 Productions
 # 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'
 Det[CASE=dat, AGR=[PER=3,NUM=pl]] -> 'den'
 Det[CASE=acc, AGR=[PER=3,NUM=pl]] -> 'die'
 # Nouns
 N[AGR=[GND=masc,PER=3,NUM=sg]] -> 'Hund'
 N[CASE=nom, AGR=[GND=masc,PER=3,NUM=pl]] -> 'Hunde'
 N[CASE=dat, AGR=[GND=masc,PER=3,NUM=pl]] -> 'Hunden'
 N[CASE=acc, AGR=[GND=masc,PER=3,NUM=pl]] -> 'Hunde'
 N[AGR=[GND=fem,PER=3,NUM=sg]] -> 'Katze'
 N[AGR=[GND=fem,PER=3,NUM=pl]] -> 'Katzen'
 # Pronouns
 PRO[CASE=nom, AGR=[PER=1,NUM=sg]] -> 'ich'
 PRO[CASE=acc, AGR=[PER=1,NUM=sg]] -> 'mich'
 PRO[CASE=dat, AGR=[PER=1,NUM=sg]] -> 'mir'
 PRO[CASE=nom, AGR=[PER=2,NUM=sg]] -> 'du'
 PRO[CASE=nom, AGR=[PER=3,NUM=sg]] -> 'er' | 'sie' | 'es'
 PRO[CASE=nom, AGR=[PER=1,NUM=pl]] -> 'wir'
 PRO[CASE=acc, AGR=[PER=1,NUM=pl]] -> 'uns'
 PRO[CASE=dat, AGR=[PER=1,NUM=pl]] -> 'uns'
 PRO[CASE=nom, AGR=[PER=2,NUM=pl]] -> 'ihr'
 PRO[CASE=nom, AGR=[PER=3,NUM=pl]] -> 'sie'
 # Verbs
 IV[AGR=[NUM=sg,PER=1]] -> 'komme'
 IV[AGR=[NUM=sg,PER=2]] -> 'kommst'
 IV[AGR=[NUM=sg,PER=3]] -> 'kommt'
 IV[AGR=[NUM=pl, PER=1]] -> 'kommen'
 IV[AGR=[NUM=pl, PER=2]] -> 'kommt'
 IV[AGR=[NUM=pl, PER=3]] -> 'kommen'
 TV[OBJCASE=acc, AGR=[NUM=sg,PER=1]] -> 'sehe' | 'mag'
 TV[OBJCASE=acc, AGR=[NUM=sg,PER=2]] -> 'siehst' | 'magst'
 TV[OBJCASE=acc, AGR=[NUM=sg,PER=3]] -> 'sieht' | 'mag'
 TV[OBJCASE=dat, AGR=[NUM=sg,PER=1]] -> 'folge' | 'helfe'
 TV[OBJCASE=dat, AGR=[NUM=sg,PER=2]] -> 'folgst' | 'hilfst'
 TV[OBJCASE=dat, AGR=[NUM=sg,PER=3]] -> 'folgt' | 'hilft'
 TV[OBJCASE=acc, AGR=[NUM=pl,PER=1]] -> 'sehen' | 'moegen'
 TV[OBJCASE=acc, AGR=[NUM=pl,PER=2]] -> 'sieht' | 'moegt'
 TV[OBJCASE=acc, AGR=[NUM=pl,PER=3]] -> 'sehen' | 'moegen'
 TV[OBJCASE=dat, AGR=[NUM=pl,PER=1]] -> 'folgen' | 'helfen'
 TV[OBJCASE=dat, AGR=[NUM=pl,PER=2]] -> 'folgt' | 'helft'
 TV[OBJCASE=dat, AGR=[NUM=pl,PER=3]] -> 'folgen' | 'helfen'
```

Example 9.5 (code_germancfg.py): **Figure 9.5**: Example Feature based Grammar

As you can see, the feature *objcase* is used to specify the case that a verb governs on its object. The next example illustrates the parse tree for a sentence containing a verb which governs dative case.

```
>>> tokens = 'ich folge den Katzen'.split()
```

```
>>> cp = load_parser('grammars/book_grammars/german.fcfg')
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S[]
  (NP[AGR=[NUM='sg', PER=1], CASE='nom']
    (PRO[AGR=[NUM='sg', PER=1], CASE='nom'] ich))
  (VP[AGR=[NUM='sg', PER=1]]
    (TV[AGR=[NUM='sg', PER=1], OBJCASE='dat'] folge)
    (NP[AGR=[GND='fem', NUM='pl', PER=3], CASE='dat']
      (Det[AGR=[NUM='pl', PER=3], CASE='dat'] den)
      (N[AGR=[GND='fem', NUM='pl', PER=3]] Katzen))))
```

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 of the `load_parser()` method can be crucial. Consider the following parse failure:

```
>>> tokens = 'ich folge den Katze'.split()
>>> cp = load_parser('grammars/book_grammars/german.fcfg', trace=2)
>>> for tree in cp.nbest_parse(tokens):
...     print tree
|.ich.fol.den.Kat.|
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:
|.   [--->   .   .| [1:2] VP[AGR=?a] -> TV[AGR=?a, OBJCASE=?c] * NP[CASE=?c] {?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' *
|.   .   [---]   .| [2:3] Det[AGR=[NUM='pl', PER=3], CASE='dat'] -> 'den' *
Feature Bottom Up Predict Combine Rule:
|.   .   [--->   .| [2:3] NP[AGR=?a, CASE=?c] -> Det[AGR=?a, CASE=?c] * N[AGR=?a, CASE=?c] {?a: [NUM='pl',
PER=3], ?c: 'dat'}
Feature Bottom Up Predict Combine Rule:
|.   .   [--->   .| [2:3] NP[AGR=?a, CASE=?c] -> Det[AGR=?a, CASE=?c] * N[AGR=?a, CASE=?c] {?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' *
```

The last two `Scanner` 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 [Example 9.5](#) that `Katze` has category `N[AGR=[GND=fem, NUM=sg, PER=3]]`. Thus there is no binding for the variable `?a` in production `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]`.

9.4 Summary

- The traditional categories of context-free grammar are atomic symbols. An important motivation for feature structures is to capture fine-grained distinctions that would otherwise require a massive multiplication of atomic categories.
- By using variables over feature values, we can express constraints in grammar productions that 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 children.
- Feature values are either atomic or complex. A particular sub-case 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 indexes (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 that contains the combined information of both FS_0 and FS_1.
- If unification specializes a path π in FS, then it also specializes 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.

9.5 Further Reading

Please consult [NLTK-URL] for further materials on this chapter, including HOWTOs feature structures, feature grammars, Earley parsing and grammar test suites.

X-bar Syntax: (Jacobs & Rosenbaum, 1970), (Jackendoff, 1977) (The primes we use replace Chomsky's typographically more demanding horizontal bars.)

For an excellent introduction to the phenomenon of agreement, see (Corbett, 2006).

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 theoretical syntax was advocated by Generalized Phrase Structure Grammar (GPSG; (Gazdar, Klein, & and, 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** that 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)). A comprehensive bibliography of HPSG literature can be found at `http://www.cl.uni-bremen.de/HPSG-Bib/`.

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 specification 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=trans]. **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) focuses 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, 2002) gives a very extensive and detailed analysis of German syntax in HPSG.

Chapter 15 of (Jurafsky & Martin, 2008) discusses feature structures, the unification algorithm, and the integration of unification into parsing algorithms.

9.6 Exercises

1. ☼ What constraints are required to correctly parse word sequences 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 (20) as the starting point.

2. ☼ Develop a variant of grammar in Example 9.1 that uses a feature *count* to make the distinctions shown below:

   (56)
       a.
          The boy sings.

       b.
          *Boy sings.

(57)

    a.
        The boys sing.

    b.
        Boys sing.

(58)

    a.
        The boys sing.

    b.
        Boys sing.

(59)

    a.
        The water is precious.

    b.
        Water is precious.

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

4. ☼ Modify the grammar illustrated in (30) to incorporate a *bar* feature for dealing with phrasal projections.

5. ☼ Modify the German grammar in Example 9.5 to incorporate the treatment of subcategorization presented in Section 9.3.

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

(60)
un	cuadro	hermos-o
INDEF.SG.MASC	picture	beautiful-SG.MASC

'a beautiful picture'

(61)
un-os	cuadro-s	hermos-os
INDEF-PL.MASC	picture-PL	beautiful-PL.MASC

'beautiful pictures'

(62)
un-a	cortina	hermos-a
INDEF-SG.FEM	curtain	beautiful-SG.FEM

'a beautiful curtain'

(63)
un-as	cortina-s	hermos-as
INDEF-PL.FEM	curtain	beautiful-PL.FEM

'beautiful curtains'

7.   Develop a wrapper for the `earley_parser` so that a trace is only printed if the input sequence fails to parse.

8.   Consider the feature structures shown in Example 9.6.

```
fs1 = nltk.FeatStruct("[A = ?x, B= [C = ?x]]")
fs2 = nltk.FeatStruct("[B = [D = d]]")
fs3 = nltk.FeatStruct("[B = [C = d]]")
fs4 = nltk.FeatStruct("[A = (1)[B = b], C->(1)]")
fs5 = nltk.FeatStruct("[A = (1)[D = ?x], C = [E -> (1), F = ?x] ]")
fs6 = nltk.FeatStruct("[A = [D = d]]")
fs7 = nltk.FeatStruct("[A = [D = d], C = [F = [D = d]]]")
```

```
fs8 = nltk.FeatStruct("[A = (1)[D = ?x, G = ?x], C = [B = ?x, E -> (1)] ]")
fs9 = nltk.FeatStruct("[A = [B = b], C = [E = [G = e]]]")
fs10 = nltk.FeatStruct("[A = (1)[B = b], C -> (1)]")
```

[Example 9.6 (code_featstructures.py)](#): **Figure 9.6**: Exploring Feature Structures

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

1. `fs1` and `fs2`
2. `fs1` and `fs3`
3. `fs4` and `fs5`
4. `fs5` and `fs6`
5. `fs5` and `fs7`
6. `fs8` and `fs9`
7. `fs8` and `fs10`

Check your answers using Python.

9. List two feature structures that subsume [A=?x, B=?x].

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

11. Extend the German grammar in [Example 9.5](#) so that it can handle so-called verb-second structures like the following:

(64)
    Heute sieht der Hund die Katze.

12. Seemingly synonymous verbs have slightly different syntactic properties [(Levin, 1993)](#). Consider the patterns of grammaticality for the verbs *loaded*, *filled*, and *dumped* below. Can you write grammar productions to handle such data?

(65)
    a.
        The farmer *loaded* the cart with sand

    b.
        The farmer *loaded* sand into the cart

    c.
        The farmer *filled* the cart with sand

    d.
        *The farmer *filled* sand into the cart

    e.
        *The farmer *dumped* the cart with sand

    f.
        The farmer *dumped* sand into the cart

13. Morphological paradigms are rarely completely regular, in the sense of every cell in the matrix having a different realization. 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.

14. So-called **head features** are shared between the parent node and head child. For example, TENSE is a head feature that is shared between a VP and its head V child. See [(Gazdar, Klein, & and, 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 productions. Develop an approach that automatically accounts for this regular behavior of head features.

15. Extend NLTK's treatment of feature structures to allow unification into list-valued features, and use this to implement an HPSG-style analysis of subcategorization, whereby the SUBCAT of a head category is the concatenation its complements' categories with the SUBCAT value of its immediate parent.

16.   Extend NLTK's treatment of feature structures to allow productions with underspecified categories, such as `S[-INV] --> ?x S/?x`.

17.   Extend NLTK's treatment of feature structures to allow typed feature structures.

18.   Pick some grammatical constructions described in [(Huddleston & Pullum, 2002)](#), and develop a feature based grammar to account for them.

About this document...