9. Feature Based Grammar

9.1 Introduction

9.2 Agreement in CFGs

9.2.1 The Problem

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

(5)

We can make the role of morphological properties a bit more explicit as illustrated in (6) and (7). These representations indicate that the verb agrees with its subject in person and number.

the	dog	run-s
	dog.3.SG	run-3.SG

(6)

the	dog-s	run
	dog-3.PL	run.3.PL

(7)

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 (8).

(8)
```
S → NP VP
NP → Det N
VP → V

Det → 'this'
N → 'dog'
V → 'runs'
```

(8) 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):

(9)
```
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 Exercises

1. Augment (9) so that it will generate strings like *I am happy* and *she is happy* but not **you is happy* or **they am happy*.

2. Augment (9) so that it will correctly describe the following Spanish noun phrases:

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

(10a)

un-os	cuadro-s	hermos-os
INDEF-PL.MASC	picture-PL	beautiful-PL.MASC
'beautiful pictures'		

(10b)

un-a	cortina	hermos-a
INDEF-SG.FEM	curtain	beautiful-SG.FEM
'a beautiful curtain'		

(10c)

un-as	cortina-s	hermos-as
INDEF-PL.FEM	curtain-PL	beautiful-SG.FEM
'beautiful curtains'		

(10d)

9.2.3 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:

(11) N[NUM = pl]

In (11), 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:

(12)
```
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 (9). Things become more interesting when we allow *variables* over feature values, and use these to state constraints. This is illustrated in (13).

(13a) S → NP[NUM = ?n] VP[NUM = ?n]

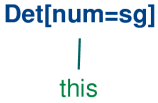
(13b) NP[NUM = ?n] → Det[NUM = ?n] N[NUM = ?n]

(13c) VP[NUM = ?n] → V[NUM = ?n]

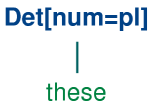
We are using '*?n*' as a variable over values of NUM; it can be instantiated either to sg or pl. Its scope is limited to individual rules. That is, within (13a), for example, *?n* must be instantiated to the same constant value; we can read the rule as saying that whatever value NP takes for the feature 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):

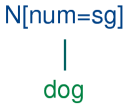
(14a)



(14b)



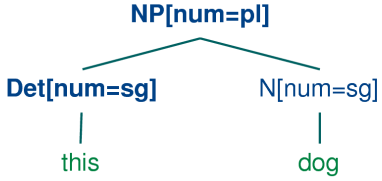
(15a)



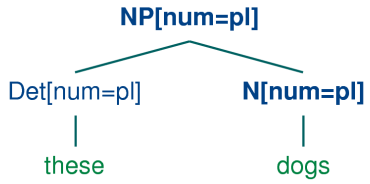
(15b)

Now (13b) says that whatever the NUM values of N and Det are, they have to be the same. Consequently, (13b) will permit (14a) and (15a) to be combined into an NP as shown in (16a) and it will also allow (14b) and (15b) to be combined, as in (16b). By contrast, (17a) and (17b) are prohibited because the roots of their constituent local trees differ in their values for the NUM feature.

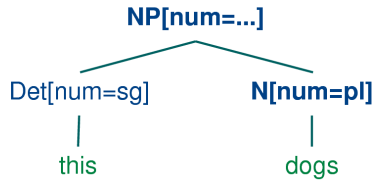
(16a)



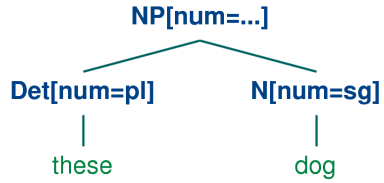
(16b)



(17a)

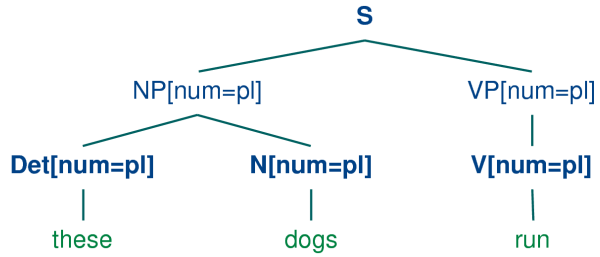


(17b)



Rule (13c) can be thought of as saying that NUM value of the head verb has to be the same as the NUM value of the VP mother. Combined with (13a), 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.

(18)



Grammar (19) illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones.

(19)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Grammar Rules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Start -> S

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[num=?n] -> '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'
```

First, 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.

Second, 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 point to note about (19) is that we have used `%` as an escape symbol in order to add comments to 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 (19) 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.contrib.grammarfile import GrammarFile
>>> from pprint import pprint
>>> from nltk_lite import tokenize
>>> g = GrammarFile.read_file('feat0.cfg')
>>>
```

We can inspect the rules and the lexicon.

```
>>> print g.earley_grammar()
Grammar with 7 productions (start state = Start[])
    Start -> S
```

```
    S -> NP[num=?n] VP[num=?n]
    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[num=?n, tense=?t] -> IV[num=?n, tense=?t]
    VP[num=?n, tense=?t] -> TV[num=?n, tense=?t] NP
>>> pprint(g.earley_lexicon())
{'Jody': [PropN[num=sg]],
 'Kim': [PropN[num=sg]],
 'all': [Det[num=pl]],
 'car': [N[num=sg]],
 'cars': [N[num=pl]],
 'child': [N[num=sg]],
 'children': [N[num=pl]],
 'disappear': [IV[num=pl, tense=pres]],
 'disappeared': [IV[num=?n, tense=past]],
 'disappears': [IV[num=sg, tense=pres]],
 'dog': [N[num=sg]],
 'dogs': [N[num=pl]],
 'every': [Det[num=sg]],
 'girl': [N[num=sg]],
 'girls': [N[num=pl]],
 'like': [TV[num=pl, tense=pres]],
 'liked': [TV[num=?n, tense=past]],
 'likes': [TV[num=sg, tense=pres]],
 'saw': [TV[num=?n, tense=past]],
 'see': [TV[num=pl, tense=pres]],
 'sees': [TV[num=sg, tense=pres]],
 'some': [Det[num=?n]],
 'the': [Det[num=?n]],
 'these': [Det[num=pl]],
 'this': [Det[num=sg]],
 'walk': [IV[num=pl, tense=pres]],
 'walked': [IV[num=?n, tense=past]],
 'walks': [IV[num=sg, tense=pres]]}
 >>>
```

Now we can tokenize a sentence and use the **`parse_n()`** function to invoke the Earley chart parser.

```
>>> from nltk_lite import tokenize
>>> sent = 'Kim likes children'
>>> tokens = list(tokenize.whitespace(sent))
>>> tokens
['Kim', 'likes', 'children']
>>> cp = g.earley_parser()
>>> trees = cp.parse_n(tokens)
|.K.l.c.|
Predictor |> . . .| Start -> * S
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   |[-] . .| PropN[num=sg] -> '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   |. [-] .| TV[num=sg, tense=pres] -> '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   |. . [-]| N[num=pl] -> '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 |[=====]| Start -> S *
Completer |[=====]| [INIT] -> Start *
>>>
```

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'))
      (VP[num=sg, tense=pres]:
        (TV[num=sg, tense=pres]: 'likes')
        (NP[num=pl]: (N[num=pl]: 'children'))))))
>>>
```

9.2.4 Exercises

1. Redo the previous two exercises, but using (19) as your starting point.

9.2.5 Terminology

So far, we have only seen feature values like sg and *pl*:fval. 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. Thus, our lexicon for verbs might include entries such as the following:

(20)
```
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:

(21)
```
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 (22).

(22)
```
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 specification, as shown in (23).

(23)
```
[POS = N, NUM = sg]
```

In fact, we regard (23) as our 'official' representation of a feature-based linguistic category, and (22) 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 (24).

(24)
```
[pos: N
 agr: [per: 3
       num: pl
       gend: fem]]
```

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 (24) is equivalent to (24).

(25)
```
[agr: [num: pl
       per: 3
       gend: fem]
 pos: N]
```

9.2.6 Feature Structures in NLTK and Unification

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

```
>>> from nltk_lite.parse.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', gend='fem')
>>> print fs1['gend']
fem
>>>
```

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

```
>>> fs1[case] = 'acc'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'case' is not defined
>>>
```

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

```
>>> fs2 = FeatureStructure(pos='N', agr=fs1)
>>> print fs2
[       [ gend = 'fem' ] ]
[ agr = [ num  = 'pl'  ] ]
[       [ per  = 3     ] ]
[                        ]
[ pos = 'N'              ]
>>> print fs2['agr']
[ gend = 'fem' ]
[ num  = 'pl'  ]
[ per  = 3     ]
>>> print fs2['agr']['per']
3
>>>
```

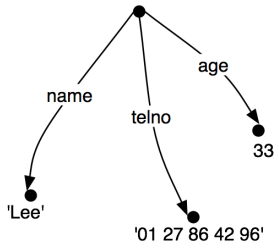
An alternative method of specifying feature structures in NLTK is to use the `parse()` method of `FeatureStructure`. This gives us the facility to use square bracket notation for embedding one feature structure within another.

```
>>> FeatureStructure.parse("[pos='N', agr=[per=3, num='pl', gend='fem']]")
[agr=[gend='fem', num='pl', per=3], 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:

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

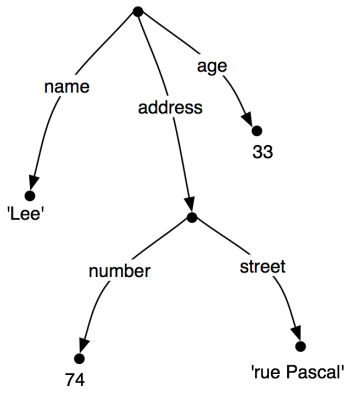
It is sometimes helpful to picture feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (26) is equivalent to the feature structure `person01` just shown.



(26)

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.

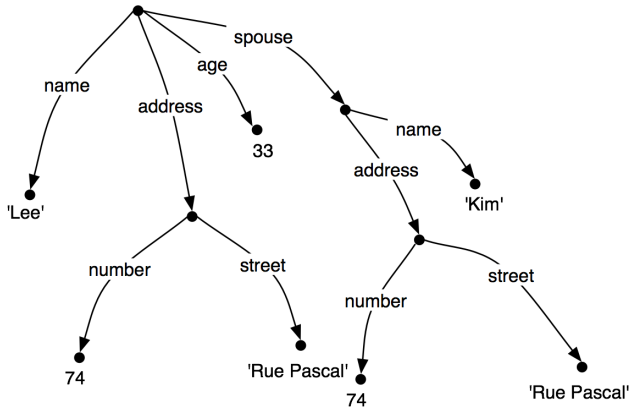
Just as before, feature values can be complex:



(27)

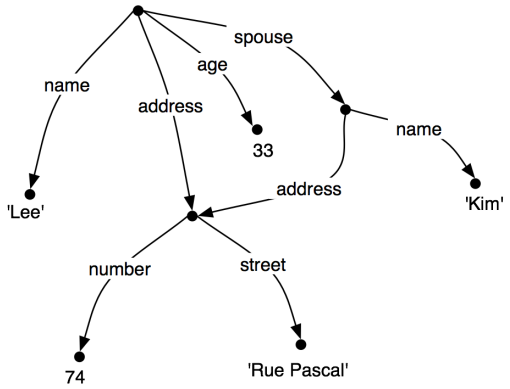
When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths in NLTK as tuples. Thus, `('address', 'street')` is a feature path whose value in (27) is the string 'rue 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 (28).



(28)

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



(29)

In other words, the value of the path `('address')` in (29) is identical to the value of the path `('spouse', 'address')`. DAGs such as (29) 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 Pasc
>>> print fs
[ 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.

```
>>> fs = FeatureStructure.parse("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
>>> print fs
[ A = 'a'              ]
[                      ]
[ B = (1) [ C = 'c' ] ]
[                      ]
[ D -> (1)             ]
[ E -> (1)             ]
>>> fs1 = FeatureStructure.parse("[A=(1)[], B=(2)[], C->(1), D->(2)]")
>>> print fs
[ A = (1) [] ]
[            ]
[ B = (2) [] ]
[            ]
[ C -> (1)   ]
[ D -> (2)   ]
>>>
```

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, (30a) is more general (less specific) than (30b), which in turn is more general than (30c).

(30a)
```
[number: 74]
```

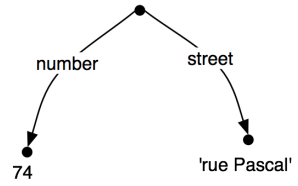
(30b)
```
[number: 74
 street: 'rue Pascal']
```

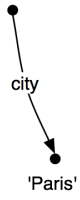
(30c)
```
[number: 74
 street: 'rue Pascal'
 city: 'Paris']
```

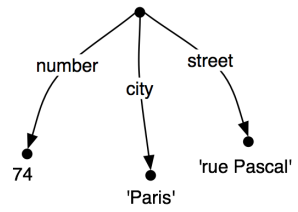
This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write $FS_0 \sqsubseteq FS_1$), then FS_1 must have all the paths and path equivalences of FS_0, and may have additional paths and equivalences as well. Thus, (28) subsumes (29), 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, (31) neither subsumes nor is subsumed by (30a).

(31)
```
[telno = '01 27 86 42 96']
```

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 (32b) with (32a) to yield (32c).

(32a)


(32b)


(32c)


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: $FS_0 \sqcap FS_1$. Unification is symmetric, so

(33) $FS_0 \sqcap FS_1 = FS_1 \sqcap FS_0$.

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:

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

For example, the result of unifying (30b) with (30c) is (30c).

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 (28).

```
>>> fs0=FeatureStructure.parse("[name='Lee', 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` includes the whole path from the root of the feature structure down to CITY.)

```
>>> fs1=FeatureStructure.parse("[spouse = [address = [city = 'Paris']]]")
>>> print fs0.unify(fs1)
[ 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, (29).

```
>>> fs2=FeatureStructure.parse("[name='Lee', address=(1)[number=74, street='rue Pas
>>> print fs2.unify(fs1)
[               [ 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 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)                          ]
>>>
```

9.2.7 Exercises

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

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

9.3 Extending a Feature-Based Grammar

9.3.1 Subcatorization

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

(35) 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 (19), shown in (36).

(36)
```
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

V[subcat=0, tense=pres,  num=sg] -> 'disappears' | 'walks'
V[subcat=1, tense=pres, num=sg] -> 'sees' | 'likes'

V[subcat=0, tense=pres,  num=pl] -> 'disappear' | 'walk'
V[subcat=1, tense=pres, num=pl] -> 'see' | 'like'

V[subcat=0, tense=past, num=?n] -> 'disappeared' | 'walked'
V[subcat=1, tense=past, num=?n] -> 'saw' | 'liked'
```

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 and 57 as our two values in (36). On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP.

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*:lx) might be represented as (37):

(37) V[SUBCAT = <NP, NP, PP>]

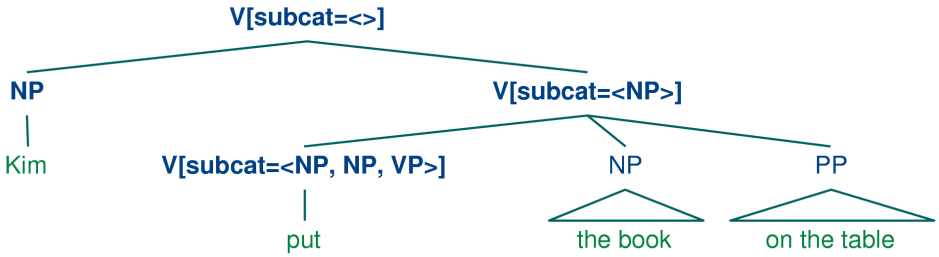
This says that the verb can combine with three arguments. The leftmost element in the list is the subject NP, while everything else — an NP followed by a PP in this case — comprises the

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.

(38) 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 (39) shows how these category assigments combine in a parse of *Kim put the book on the table*.

(39)



9.3.2 Unbounded Dependency Constructions

Consider the following contrasts:

(40a) We liked the music.

(40b) *We liked.

(41a) We put the card into the slot.

(41b) *We put into the slot.

(41c) *We put the card.

(41d) *We put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. Examples (40) and (41) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (42) and (43) illustrate.

(42a) She knows which music we like.

(42b) This music, we really like.

(43a) Which card did you put into the slot?

(43b) Which slot did you put the card into?

9.4 Adding Compositional Semantics

9.4.1 Overview

One of the goals of linguistic theory is to provide a systematic correspondence between form and meaning. One widely adopted approach to representing meaning — or at least, some aspects of meaning — involves translating expressions of natural language in to first order logic. From a computational point of view, a strong argument in favour of first order logic is that it strikes a reasonable balance between expressiveness and logical tractability. On the one hand, it is flexible enough to represent many aspects of the logical structure of natural language. On the other hand, automated theorem proving for first order logic has received much attention, and although inference in first order logic is not decidable, in practice many reasoning problems are efficiently solvable using modern theorem provers.

Standard textbooks on first order logic often contain exercises in which the reader is required to translate between English and logic, as illustrated in (44) and (45).[1]

(44a) If all whales are mammals, then Moby Dick is not a fish.

(44b) $\forall x(\text{whale}(x) \rightarrow \text{mammal}(x)) \rightarrow \neg\text{fish}(\text{MD})$

(45a) There is a painting that all critics admire.

(45b) $\exists y(\text{painting}(y) \wedge \forall x(\text{critic}(x) \rightarrow \text{admire}(x, y)))$

Although there are numerous subtle and thorny issues about how this translation should be carried out in particular cases, we will put these to one side. The main focus of our discussion will be on a different problem: how can we systematically construct a semantic representation for a sentence which proceeds in step with the process of parsing that sentence?

Unfortunately, it is not within the scope of this chapter to introduce the syntax and semantics of first order logic, so if you don't already have some familiarity with it, we suggest you consult an appropriate source.

9.4.2 The λ calculus

syntax of λ calculus; functions; β conversion

9.4.3 Compositionality

Sample grammar
- coordination
- quantification and scope

9.4.4 Feature-based Semantics

9.5 Further Reading

Gerald Gazdar, Ewan Klein, Geoffrey Pullum and Ivan Sag (1985) *Generalized Phrase Structure Grammar*, Basil Blackwell.

[1] These examples come, respectively, from D. Kalish and R. Montague (1964) *Logic: Techniques of Formal Reasoning*, Harcourt, Brace and World, p94, and W. v. Quine (1952) *Methods of Logic*, Routledge and Kegan Paul, p121.

Ivan A. Sag and Thomas Wasow (1999) *Syntactic Theory: A Formal Introduction*, CSLI Publications.

Patrick Blackburn and Johan Bos *Representation and Inference for Natural Language: A First Course in Computational Semantics*, CSLI Publications

9.6 Exercises

1.