Part II

PARSING

Chapter 6

Structured Programming in Python

6.1 Introduction

In Part I you had an intensive introduction to Python (Chapter 2) followed by chapters on words, tags, and chunks (Chapters 3-5). These chapters contain many examples and exercises that should have helped you consolidate your Python skills and apply them to simple NLP tasks. So far our programs — and the data we have been processing — have been relatively unstructured. In Part II we will focus on *structure*: i.e. structured programming with structured data.

In this chapter we will review key programming concepts and explain many of the minor points that could easily trip you up. More fundamentally, we will introduce important concepts in structured programming that help you write readable, well-organized programs that you and others will be able to re-use. Each section is independent, so you can easily select what you most need to learn and concentrate on that. As before, this chapter contains many examples and exercises (and as before, some exercises introduce new material). Readers new to programming should work through them carefully and consult other introductions to programming if necessary; experienced programmers can quickly skim this chapter.

6.2 Back to the Basics

Let's begin by revisiting some of the fundamental operations and data structures required for natural language processing in Python. It is important to appreciate several finer points in order to write Python programs which are not only correct but also *idiomatic* — by this, we mean using the features of the Python language in a natural and concise way. To illustrate, here is a technique for iterating over the members of a list by initializing an index `i` and then incrementing the index each time we pass through the loop:

```
>>> sent = ['I', 'am', 'the', 'Walrus']
>>> i = 0
>>> while i < len(sent):
...     sent[i].lower()
...     i += 1
>>> sent
['i', 'am', 'the', 'walrus']
```

Although this does the job, it is not idiomatic Python. By contrast, Python's `for` statement allows us to achieve the same effect much more succinctly:

```
>>> sent = ['I', 'am', 'the', 'Walrus']
>>> for s in sent:
...     s.lower()
>>> sent
['i', 'am', 'the', 'walrus']
```

We'll start with the most innocuous operation of all: *assignment*. Then we will look at sequence types in detail.

6.2.1 Assignment

Python's assignment statement operates on *values*. But what is a value? Consider the following code fragment:

```
>>> word1 = 'Monty'
>>> word2 = word1              ①
>>> word1 = 'Python'           ②
>>> word2
'Monty'
```

This code shows that when we write `word2 = word1` in line ①, the value of `word1` (the string `'Monty'`) is assigned to `word2`. That is, `word2` is a **copy** of `word1`, so when we overwrite `word1` with a new string `'Python'` in line ②, the value of `word2` is not affected.

However, assignment statements do not always involve making copies in this way. An important subtlety of Python is that the "value" of a structured object (such as a list) is actually a *reference* to the object. In the following example, line ① assigns the reference of `list1` to the new variable `list2`. When we modify something inside `list1` on line ②, we can see that the contents of `list2` have also been changed.

```
>>> list1 = ['Monty', 'Python']
>>> list2 = list1              ①
>>> list1[1] = 'Bodkin'        ②
>>> list2
['Monty', 'Bodkin']
```

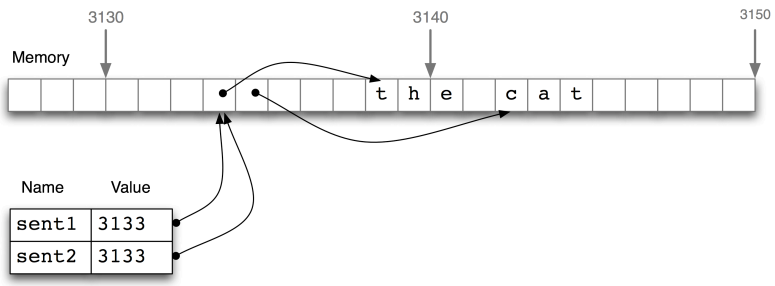


Figure 6.1: List Assignment and Computer Memory

Thus line ① does not copy the contents of the variable, only its "object reference". To understand what is going on here, we need to know how lists are stored in the computer's memory. In Figure 6.1, we see that a list `sent1` is a reference to an object stored at location 3133 (which is itself a series of pointers to other locations holding strings). When we assign `sent2 = sent1`, it is just the object reference 3133 that gets copied.

6.2.2 Sequences: Strings, Lists and Tuples

We have seen three kinds of sequence object: strings, lists, and tuples. As sequences, they have some common properties: they can be indexed and they have a length:

```
>>> string = 'I turned off the spectroroute'
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> pair = (6, 'turned')
>>> string[2], words[3], pair[1]
('t', 'the', 'turned')
>>> len(string), len(words), len(pair)
(29, 5, 2)
```

We can iterate over the items in a sequence `s` in a variety of useful ways, as shown in Table 6.1.

Python Expression	Comment
`for item in s`	iterate over the items of `s`
`for item in sorted(s)`	iterate over the items of `s` in order
`for item in set(s)`	iterate over unique elements of `s`
`for item in reversed(s)`	iterate over elements of `s` in reverse
`for item in set(s).difference(t)`	iterate over elements of `s` not in `t`

Table 6.1: Various ways to iterate over sequences

The sequence functions illustrated in Table 6.1 can be combined in various ways; for example, to get unique elements of `s` sorted in reverse, use `reversed(sorted(set(s)))`. Sometimes random order is required:

```
>>> import random
>>> random.shuffle(words)
```

We can convert between these sequence types. For example, `tuple(s)` converts any kind of sequence into a tuple, and `list(s)` converts any kind of sequence into a list. We can convert a list of strings to a single string using the `join()` function, e.g. `':'.join(words)`.

Notice in the above code sample that we computed multiple values on a single line, separated by commas. These comma-separated expressions are actually just tuples — Python allows us to omit the parentheses around tuples if there is no ambiguity. When we print a tuple, the parentheses are always displayed. By using tuples in this way, we are implicitly aggregating items together.

In the next example, we use tuples to re-arrange the contents of our list. (We can omit the parentheses because the comma has higher precedence than assignment.)

```
>>> words[2], words[3], words[4] = words[3], words[4], words[2]
>>> words
['I', 'turned', 'the', 'spectroroute', 'off']
```

This is an idiomatic and readable way to move items inside a list. It is equivalent to the following traditional way of doing such tasks that does not use tuples (notice that this method needs a temporary variable `tmp`).

```
>>> tmp = words[2]
>>> words[2] = words[3]
>>> words[3] = words[4]
>>> words[4] = tmp
```

As we have seen, Python has sequence functions such as `sorted()` and `reversed()` which rearrange the items of a sequence. There are also functions that modify the *structure* of a sequence and which can be handy for language processing. Thus, `zip()` takes the items of two sequences and 'zips' them together into a single list of pairs. Given a sequence `s`, `enumerate(s)` returns an iterator that produces a pair of an index and the item at that index.

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['nnp', 'vbd', 'in', 'dt', 'nn']
>>> zip(words, tags)
[('I', 'nnp'), ('turned', 'vbd'), ('off', 'in'),
('the', 'dt'), ('spectroroute', 'nn')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]
```

6.2.3 Combining different sequence types

Let's combine our knowledge of these three sequence types, together with list comprehensions, to perform the task of sorting the words in a string by their length.

```
>>> words = 'I turned off the spectroroute'.split()      ①
>>> wordlens = [(len(word), word) for word in words]     ②
>>> wordlens
[(1, 'I'), (6, 'turned'), (3, 'off'), (3, 'the'), (12, 'spectroroute')
>>> wordlens.sort()                                      ③
>>> ' '.join([word for count, word in wordlens])         ④
'I off the turned spectroroute'
```

Each of the above lines of code contains a significant feature. Line ① demonstrates that a simple string is actually an object with methods defined on it, such as `split()`. Line ② shows the construction of a list of tuples, where each tuple consists of a number (the word length) and the word, e.g. `(3, 'the')`. Line ③ sorts the list, modifying the list in-place. Finally, line ④ discards the length information then joins the words back into a single string.

We began by talking about the commonalities in these sequence types, but the above code illustrates important differences in their roles. First, strings appear at the beginning and the end: this is typical in the context where our program is reading in some text and producing output for us to read. Lists and tuples are used in the middle, but for different purposes. A list is typically a sequence of objects all having the *same type*, of *arbitrary length*. We often use lists to hold sequences of words. In contrast, a tuple is typically a collection of objects of *different types*, of *fixed length*. We often use a tuple to hold a **record**, a collection of different **fields** relating to some entity. This distinction between the use of lists and tuples takes some getting used to, so here is another example:

```
>>> lexicon = [
...     ('the', 'DT', ['Di:', 'D@']),
...     ('off', 'IN', ['Qf', 'O:f'])
... ]
```

Here, a lexicon is represented as a list because it is a collection of objects of a single type — lexical entries — of no predetermined length. An individual entry is represented as a tuple because it is a collection of objects with different interpretations, such as the orthographic form, the part of speech, and the pronunciations represented in the SAMPA computer readable phonetic alphabet. Note that these pronunciations are stored using a list. (Why?)

The distinction between lists and tuples has been described in terms of usage. However, there is a more fundamental difference: in Python, lists are **mutable**, while tuples are **immutable**. In other words, lists can be modified, while tuples cannot. Here are some of the operations on lists which do in-place modification of the list. None of these operations is permitted on a tuple, a fact you should confirm for yourself.

```
>>> lexicon.sort()
>>> lexicon[1] = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> del lexicon[0]
```

6.2.4 Stacks and Queues

Lists are a particularly versatile data type. We can use lists to implement higher-level data types such as stacks and queues. A **stack** is a container that has a **first-in-first-out** policy for adding and removing items (see Figure 6.2).

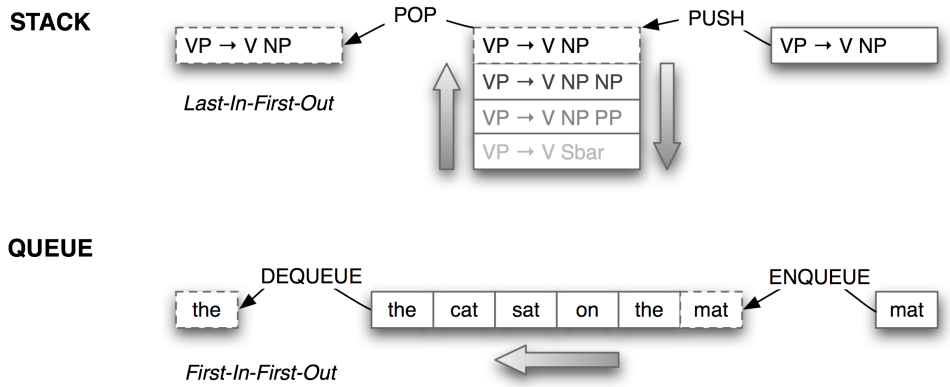


Figure 6.2: Stacks and Queues

Stacks are used to keep track of the current context in computer processing of natural languages (and programming languages too). We will seldom have to deal with stacks explicitly, as the implementation of NLTK parsers, treebank corpus readers, (and even Python functions), all use stacks behind the scenes. However, it is important to understand what stacks are and how they work.

In Python, we can treat a list as a stack by limiting ourselves to the three operations defined on stacks: `append(item)` (to push `item` onto the stack), `pop()` to pop the item off the top of the stack, and `[-1]` to access the item on the top of the stack. Listing 6.1 processes a sentence with phrase markers, and checks that the parentheses are balanced. The loop pushes material onto the stack when it gets an open parenthesis, and pops the stack when it gets a close parenthesis. We see that two are left on the stack at the end; i.e. the parentheses are not balanced.

Although Listing 6.1 is a useful illustration of stacks, it is overkill because we could have done a direct count: `phrase.count('(') == phrase.count(')')`. However, we can use stacks for more sophisticated processing of strings containing nested structure, as shown in Listing 6.2. Here we build a (potentially deeply-nested) list of lists. Whenever a token other than a parenthesis is encountered, we add it to a list at the appropriate level of nesting. The stack cleverly keeps track of this level of nesting, exploiting the fact that the item at the top of the stack is actually shared with a more deeply nested item. (Hint: add diagnostic print statements to the function to help you see what it is doing.)

Listing 12 Check parentheses are balanced

```
def check_parens(tokens):
    stack = []
    for token in tokens:
        if token == '(':      # push
            stack.append(token)
        elif token == ')':    # pop
            stack.pop()
    return stack

>>> phrase = "( the cat ) ( sat ( on ( the mat )"
>>> print check_parens(phrase.split())
['(', '(']
```

Listing 13 Convert a nested phrase into a nested list using a stack

```
def convert_parens(tokens):
    stack = [[]]
    for token in input:
        if token == '(':      # push
            sublist = []
            stack[-1].append(sublist)
            stack.append(sublist)
        elif token == ')':    # pop
            stack.pop()
        else:                 # update top of stack
            stack[-1].append(token)
    return stack[0]

>>> phrase = "( the cat ) ( sat ( on ( the mat ) ) )"
>>> print convert_parens(phrase.split())
[['the', 'cat'], ['sat', ['on', ['the', 'mat']]]]
```

Lists can be used to represent another important data structure. A **queue** is a container that has a **last-in-first-out** policy for adding and removing items (see Figure 6.2). Queues are used for scheduling activities or resources. As with stacks, we will seldom have to deal with queues explicitly, as the implementation of NLTK n-gram taggers (Section 4.6) and chart parsers (Section 8.2) use queues behind the scenes. However, we will take a brief look at how queues are implemented using lists.

```
>>> queue = ['the', 'cat', 'sat']
>>> queue.append('on')
>>> queue.append('the')
>>> queue.append('mat')
>>> queue.pop(0)
'the'
>>> queue.pop(0)
'cat'
>>> queue
['sat', 'on', 'the', 'mat']
```

6.2.5 More List Comprehensions

You may recall that in Chapter 3, we introduced list comprehensions, with examples like the following:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> [word.lower() for word in sent]
['the', 'dog', 'gave', 'john', 'the', 'newspaper']
```

List comprehensions are a convenient and readable way to express list operations in Python, and they have a wide range of uses in natural language processing. In this section we will see some more examples. The first of these takes successive overlapping slices of size `n` (a **sliding window**) from a list (pay particular attention to the range of the variable `i`).

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> n = 3
>>> [sent[i:i+n] for i in range(len(sent)-n+1)]
[['The', 'dog', 'gave'],
 ['dog', 'gave', 'John'],
 ['gave', 'John', 'the'],
 ['John', 'the', 'newspaper']]
```

You can also use list comprehensions for a kind of multiplication. Here we generate all combinations of two determiners, two adjectives, and two nouns. The list comprehension is split across three lines for readability.

```
>>> [(dt,jj,nn) for dt in ('two', 'three')
...             for jj in ('old', 'blind')
...             for nn in ('men', 'mice')]
[('two', 'old', 'men'), ('two', 'old', 'mice'), ('two', 'blind', 'men'),
 ('two', 'blind', 'mice'), ('three', 'old', 'men'), ('three', 'old', 'mice'),
 ('three', 'blind', 'men'), ('three', 'blind', 'mice')]
```

The above example contains three independent `for` loops. These loops have no variables in common, and we could have put them in any order. We can also have **nested loops** with shared variables. The next example iterates over all sentences in a section of the Brown Corpus, and for each sentence, iterates over each word.

```
>>> from nltk_lite.corpora import brown
>>> [word for sent in brown.raw('a')
...      for word in sent
...      if len(word) == 17]
['September-October', 'Sheraton-Biltmore', 'anti-organization',
 'anti-organization', 'Washington-Oregon', 'York-Pennsylvania',
 'misunderstandings', 'Sheraton-Biltmore', 'neo-stagnationist',
 'cross-examination', 'bronzy-green-gold', 'Oh-the-pain-of-it',
 'Secretary-General', 'Secretary-General', 'textile-importing',
 'textile-exporting', 'textile-producing', 'textile-producing']
```

As you will see, the list comprehension in this example contains a final `if` clause which allows us to filter out any words that fail to meet the specified condition.

Another way to use loop variables is to ignore them! This is the standard method for building multidimensional structures. For example, to build an array with *m* rows and *n* columns, where each cell is a set, we would use a nested list comprehension, as shown in line ① below. Observe that the loop variables `i` and `j` are not used anywhere in the expressions preceding the `for` clauses.

```
>>> from pprint import pprint
>>> m, n = 3, 7
>>> array = [[set() for i in range(n)] for j in range(m)]      ①
>>> array[2][5].add('foo')
>>> pprint(array)
[[set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set(['foo']), set([])]]
```

Sometimes we use a list comprehension as part of a larger aggregation task. In the following example we calculate the average length of words in part of the Brown Corpus. Notice that we don't bother storing the list comprehension in a temporary variable, but use it directly as an argument to the `average()` function.

```
>>> from numpy import average
>>> average([len(word) for sent in brown.raw('a') for word in sent])
4.40154543827
```

Now that we have reviewed the sequence types, we have one more fundamental data type to revisit.

6.2.6 Dictionaries

As you have already seen, the dictionary data type can be used in a variety of language processing tasks (e.g. Section 2.6). However, we have only scratched the surface. Dictionaries have many more applications than you might have imagined.

> **Note**
>
> The dictionary data type is often known by the name **associative array**. A normal array maps from integers (the keys) to arbitrary data types (the values), while an associative array places no such constraint on keys. Keys can be strings, tuples, or other more deeply nested structure. Python places the constraint that keys must be *immutable*.

Let's begin by comparing dictionaries with tuples. Tuples allow *access by position*; to access the part-of-speech of the following lexical entry we just have to know it is found at index position `1`. However, dictionaries allow *access by name*:

```
>>> lexical_entry = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> lexical_entry[1]
'VBD'
>>> entry_dict = {'lexeme':'turned', 'pos':'VBD', 'pron':['t3:nd', 't3`nd']}
>>> entry_dict['pos']
'VBD'
```

In this case, dictionaries are little more than a convenience. We can even simulate access by name using well-chosen constants, e.g.:

```
>>> LEXEME = 0; POS = 1; PRON = 2
>>> entry_tuple[POS]
'VBD'
```

This method works when there is a closed set of keys and the keys are known in advance. Dictionaries come into their own when we are mapping from an open set of keys, which happens when the keys are drawn from an unrestricted vocabulary or where they are generated by some procedure. Listing 6.3 illustrates the first of these. The function `mystery()` begins by initializing a dictionary called `groups`, then populates it with words. We leave it as an exercise for the reader to work out what this function computes. For now, it's enough to note that the keys of this dictionary are an open set, and it would not be feasible to use a integer keys, as would be required if we used lists or tuples for the representation.

Listing 14 Mystery program

```
from string import join
def mystery(input):
    groups = {}
    for word in input:
        key = join(sorted(list(word)), '')
        if key not in groups:                    ①
            groups[key] = set()                  ②
        groups[key].add(word)                    ③
    return sorted(join(sorted(v)) for v in groups.values() if len(v) > 1)

>>> from nltk_lite.corpora import words
>>> text = words.raw()
>>> print mystery(text)
```

Listing 6.3 illustrates two important idioms, which we already touched on in Chapter 2. First, dictionary keys are unique; in order to store multiple items in a single entry we define the value to be a list or a set, and simply update the value each time we want to store another item (line ③). Second, if a key does not yet exist in a dictionary (line ①) we must explicitly add it and give it an initial value (line ②).

The second important use of dictionaries is for mappings that involve **compound keys**. Suppose we want to categorize a series of linguistic observations according to two or more properties. We can combine the properties using a tuple and build up a dictionary in the usual way, as exemplified in Listing 6.4.

Listing 15 Illustration of compound keys

```
from nltk_lite.corpora import ppattach
attachment = {}
V, N = 0, 1
for entry in ppattach.dictionary('training'):
    key = verb, prep
    if key not in attachment:
        attachment[key] = [0,0]
    if entry['attachment'] == 'V':
        attachment[key][V] += 1
    else:
        attachment[key][N] += 1
```

6.2.7 Exercises

1. ☼ Find out more about sequence objects using Python's help facility. In the interpreter, type `help(str)`, `help(list)`, and `help(tuple)`. This will give you a full list of the functions supported by each type. Some functions have special names flanked with underscore; as the help documentation shows, each such function corresponds to something more familiar. For example `x.__getitem__(y)` is just a long-winded way of saying `x[y]`.

2. ☼ Identify three operations that can be performed on both tuples and lists. Identify three list operations that cannot be performed on tuples. Name a context where using a list instead of a tuple generates a Python error.

3. ☼ Find out how to create a tuple consisting of a single item. There are at least two ways to do this.

4. ☼ Create a list `words = ['is', 'NLP', 'fun', '?']`. Use a series of assignment statements (e.g. `words[1] = words[2]`) and a temporary variable `tmp` to transform this list into the list `['NLP', 'is', 'fun', '!']`. Now do the same transformation using tuple assignment.

5. ☼ Does the method for creating a sliding window of n-grams behave correctly for the two limiting cases: $n = 1$, and $n =$ `len(sent)`?

6. ◑ Create a list of words and store it in a variable `sent1`. Now assign `sent2 = sent1`. Modify one of the items in `sent1` and verify that `sent2` has changed.

   a) Now try the same exercise but instead assign `sent2 = sent1[:]`. Modify `sent1` again and see what happens to `sent2`. Explain.

   b) Now define `text1` to be a list of lists of strings (e.g. to represent a text consisting of multiple sentences. Now assign `text2 = text1[:]`, assign a new value to one of the words, e.g. `text1[1][1] = 'Monty'`. Check what this did to `text2`. Explain.

   c) Load Python's `deepcopy()` function (i.e. `from copy import deepcopy`), consult its documentation, and test that it makes a fresh copy of any object.

7. ◑ Write code which starts with a string of words and results in a new string consisting of the same words, but where the first word swaps places with the second, and so on. For example, `'the cat sat on the mat'` will be converted into `'cat the on sat mat the'`.

8. ◑ Initialize an *n*-by-*m* list of lists of empty strings using list multiplication, e.g. `word_table = [['']*n]*m`. What happens when you set one of its values, e.g. `word_table[1][2] = "hello"`? Explain why this happens. Now write an expression using `range()` to construct a list of lists, and show that it does not have this problem.

9. ◑ Write code to initialize a two-dimensional array of sets called `word_vowels` and process a list of words, adding each word to `word_vowels[l][v]` where `l` is the length of the word and `v` is the number of vowels it contains.

10. ◑ Write code that builds a dictionary of dictionaries of sets.

11. ◑ Use `sorted()` and `set()` to get a sorted list of tags used in the Brown corpus, removing duplicates.

12. ★ Extend the example in Listing 6.4 in the following ways:

    a) Define two sets `verbs` and `preps`, and add each verb and preposition as they are encountered. (Note that you can add an item to a set without bothering to check whether it is already present.)

    b) Create nested loops to display the results, iterating over verbs and prepositions in sorted order. Generate one line of output per verb, listing prepositions and attachment ratios as follows: `raised: about 0:3, at 1:0, by 9:0, for 3:6, from 5:0, in 5:5...`

    c) We used a tuple to represent a compound key consisting of two strings. However, we could have simply concatenated the strings, e.g. `key = verb + ":" + prep`, resulting in a simple string key. Why is it better to use tuples for compound keys?

6.3 Presenting Results

Often we write a program to report a single datum, such as a particular element in a corpus that meets some complicated criterion, or a single summary statistic such as a word-count or the performance of a tagger. More often, we write a program to produce a structured result, such as a tabulation of numbers or linguistic forms, or a reformatting of the original data. When the results to be presented are linguistic, textual output is usually the most natural choice. However, when the results are numerical, it may be preferrable to produce graphical output. In this section you will learn about a variety of ways to present program output.

6.3.1 Strings and Formats

We have seen that there are two ways to display the contents of an object:

```
>>> word = 'cat'
>>> sentence = """hello
... world"""
>>> print word
cat
>>> print sentence
hello
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

The `print` command yields Python's attempt to produce the most human-readable form of an object. The second method — naming the variable at a prompt — shows us a string that can be used to recreate this object. It is important to keep in mind that both of these are just strings, displayed for the benefit of you, the user. They do not give us any clue as to the actual internal representation of the object.

There are many other useful ways to display an object as a string of characters. This may be for the benefit of a human reader, or because we want to **export** our data to a particular file format for use in an external program.

Formatted output typically contains a combination of variables and pre-specified strings, e.g. given a dictionary `wordcount` consisting of words and their frequencies we could do:

```
>>> wordcount = {'cat':3, 'dog':4, 'snake':1}
>>> for word in wordcount:
...     print word, '->', wordcount[word], ';',
dog -> 4 ; cat -> 3 ; snake -> 1
```

Apart from the problem of unwanted whitespace, print statements that contain alternating variables and constants can be difficult to read and maintain. A better solution is to use print formatting strings:

```
>>> for word in wordcount:
...     print '%s->%d;' % (word, wordcount[word]),
dog->4; cat->3; snake->1
```

6.3.2 Lining things up

So far our formatting strings have contained specifications of fixed width, such as `%6s`, a string that is padded to width 6 and right-justified. We can include a minus sign to make it left-justified. In case we don't know in advance how wide a displayed value should be, the width value can be replaced with a star in the formatting string, then specified using a variable:

```
>>> '%6s' % 'dog'
'   dog'
>>> '%-6s' % 'dog'
'dog   '
>>> width = 6
>>> '%-*s' % (width, 'dog')
'dog   '
```

Other control characters are used for decimal integers and floating point numbers. Since the percent character `%` has a special interpretation in formatting strings, we have to precede it with another `%` to get it in the output:

```
>>> "accuracy for %d words: %2.4f%%" % (9375, 100.0 * 3205/9375)
'accuracy for 9375 words: 34.1867%'
```

An important use of formatting strings is for tabulating data. The program in Listing 6.5 iterates over five genres of the Brown Corpus. For each token having the `md` tag we increment a count. To do this we have used `ConditionalFreqDist()`, where the condition is the current genre and the event is the modal, i.e. this constructs a frequency distribution of the modal verbs in each genre. Line ① identifies a small set of modals of interest, and calls the function `tabulate()` which processes the data structure to output the required counts. Note that we have been careful to separate the language processing from the tabulation of results.

There are some interesting patterns in the table produced by Listing 6.5. For instance, compare the rows for government literature and adventure literature; the former is dominated by the use of `can , may, must, will` while the latter is characterised by the use of `could` and `might`. With some further work it might be possible to guess the genre of a new text automatically, according to its distribution of modals.

Our next example, in Listing 6.6, generates a concordance display. We use the left/right justification of strings and the variable width to get vertical alignment of a variable-width window.

[TODO: explain ValueError exception]

6.3.3 Writing results to a file

We have seen how to read text from files (Section 3.2.1). It is often useful to write output to files as well. The following code opens a file `output.txt` for writing, and saves the program output to the file.

```
>>> from nltk_lite.corpora import genesis
>>> file = open('output.txt', 'w')
>>> words = set(genesis.raw())
>>> words.sort()
>>> for word in words:
...     file.write(word + "\n")
```

When we write non-text data to a file we must convert it to a string first. We can do this conversion using formatting strings, as we saw above. We can also do it using Python's backquote notation, which converts any object into a string. Let's write the total number of words to our file, before closing it.

```
>>> len(words)
4408
>>> `len(words)`
'4408'
>>> file.write(`len(words)` + "\n")
>>> file.close()
```

6.3.4 Graphical presentation

So far we have focussed on textual presentation and the use of formatted print statements to get output lined up in columns. It is often very useful to display numerical data in graphical form, since this often

Listing 16 Frequency of Modals in Different Sections of the Brown Corpus

```
from nltk_lite.probability import ConditionalFreqDist
def count_words_by_tag(t, genres):
    cfdist = ConditionalFreqDist()
    for genre in genres:                          # each genre
        for sent in brown.tagged(genre):          # each sentence
            for (word,tag) in sent:               # each tagged token
                if tag == t:                      # found a word tagged t
                    cfdist[genre].inc(word.lower())
    return cfdist

def tabulate(cfdist, words):
    print '%-18s' % 'Genre', ' '.join([('%6s' % w) for w in words])
    for genre in cfdist.conditions():                # for each genre
        print '%-18s' % brown.item_name[genre],      # print row heading
        for w in words:                              # for each word
            print '%6d' % cfdist[genre].count(w),    # print table cell
        print                                        # end the row

>>> genres = ['a', 'd', 'e', 'h', 'n']
>>> cfdist = count_words_by_tag('md', genres)
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']    ①
>>> tabulate(cfdist, modals)
Genre                 can  could    may  might   must   will
press: reportage       94     86     66     36     50    387
skill and hobbies     273     59    130     22     83    259
religion               84     59     79     12     54     64
miscellaneous: gov    115     37    152     13     99    237
fiction: adventure     48    154      6     58     27     48
```

Listing 17 Simple Concordance Display

```
def concordance(word, context):
    "Generate a concordance for the word with the specified context window"
    for sent in brown.raw('a'):
        try:
            pos = sent.index(word)
            left = ' '.join(sent[:pos])
            right = ' '.join(sent[pos+1:])
            print '%*s %s %-*s' %\
                (context, left[-context:], word, context, right[:context])
        except ValueError:
            pass

>>> concordance('line', 32)
ce , is today closer to the NATO line .
n more activity across the state line in Massachusetts than in Rhode I
 , gained five yards through the line and then uncorked a 56-yard touc
                `` Our interior line and out linebackers played excep
k then moved Cooke across with a line drive to left .
chal doubled down the rightfield line and Cooke singled off Phil Shart
              -- Billy Gardner's line double , which just eluded the d
           -- Nick Skorich , the line coach for the football champion
                    Maris is in line for a big raise .
uld be impossible to work on the line until then because of the large
         Murray makes a complete line of ginning equipment except for
    The company sells a complete line of gin machinery all over the co
tter Co. of Sherman makes a full line of gin machinery and equipment .
fred E. Perlman said Tuesday his line would face the threat of bankrup
 sale of property disposed of in line with a plan of liquidation .
 little effort spice up any chow line .
es , filed through the cafeteria line .
l be particularly sensitive to a line between first and second class c
A skilled worker on the assembly line , for example , earns $37 a week
```

makes it easier to detect patterns. For example, in Listing 6.5 we saw a table of numbers showing the frequency of particular modal verbs in the Brown Corpus, classified by genre. In Listing 6.7 we present the same information in graphical format. The output is shown in Figure 6.3 (a color figure in the online version).

Note

Listing 6.7 uses the PyLab package which supports sophisticated plotting functions with a MATLAB-style interface. For more information about this package please see `http://matplotlib.sourceforge.net/`.

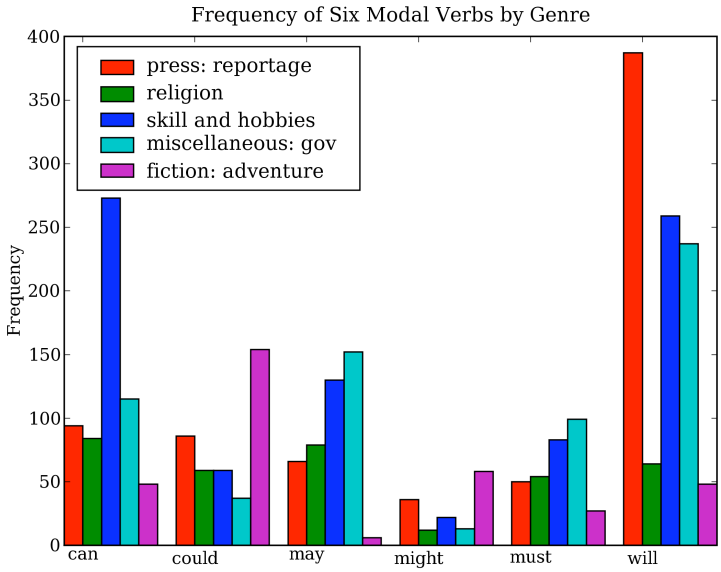


Figure 6.3: Bar Chart Showing Frequency of Modals in Different Sections of Brown Corpus

From the bar chart it is immediately obvious that *may* and *must* have almost identical relative frequencies. The same goes for *could* and *might*.

6.3.5 Exercises

1. ☼ Write code that removes whitespace at the beginning and end of a string, and normalizes whitespace between words to be a single space character.

   1) do this task using `split()` and `join()`
   2) do this task using regular expression substitutions

2. ☼ What happens when the formatting strings `%6s` and `%-6s` are used to display strings that are longer than six characters?

3. ☼ We can use a dictionary to specify the values to be substituted into a formatting string. Read Python's library documentation for formatting strings (http://docs.python.org/lib/typesseq-strings.html), and use this method to display today's date in two different formats.

Listing 18 Frequency of Modals in Different Sections of the Brown Corpus

```
from nltk_lite.corpora import brown
colors = 'rgbcmyk' # red, green, blue, cyan, magenta, yellow, black
def bar_chart(categories, words, counts):
    "Plot a bar chart showing counts for each word by category"
    import pylab
    ind = pylab.arange(len(words))
    width = 1.0 / (len(categories) + 1)
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind+c*width, counts[categories[c]], width, color=colors[c % len(c
        bar_groups.append(bars)
    pylab.xticks(ind+width, words)
    pylab.legend([b[0] for b in bar_groups], [brown.item_name[c][:18] for c in categories],
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()

>>> genres = ['a', 'd', 'e', 'h', 'n']
>>> cfdist = count_words_by_tag('md', genres)
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> counts = {}
>>> for genre in genres:
...     counts[genre] = [cfdist[genre].count(word) for word in modals]
>>> bar_chart(genres, modals, counts)
```

4. ◑ Listing 4.4 in Chapter 4 plotted a curve showing change in the performance of a lookup tagger as the model size was increased. Plot the performance curve for a unigram tagger, as the amount of training data is varied.

6.4 Functions

Once you have been programming for a while, you will find that you need to perform a task that you have done in the past. In fact, over time, the number of completely novel things you have to do in creating a program decreases significantly. Half of the work may involve simple tasks that you have done before. Thus it is important for your code to be *re-usable*. One effective way to do this is to abstract commonly used sequences of steps into a **function**, as we briefly saw in Chapter 2.

For example, suppose we find that we often want to read text from an HTML file. This involves several steps: opening the file, reading it in, normalizing whitespace, and stripping HTML markup. We can collect these steps into a function, and give it a name such as `get_text()`:

Listing 19 Read text from a file

```
import re
def get_text(file):
    """Read text from a file, normalizing whitespace
    and stripping HTML markup."""
    text = open(file).read()
    text = re.sub('\s+', ' ', text)
    text = re.sub(r'<.*?>', ' ', text)
    return text
```

Now, any time we want to get cleaned-up text from an HTML file, we can just call `get_text()` with the name of the file as its only argument. It will return a string, and we can assign this to a variable, e.g.: `contents = get_text("test.html")`. Each time we want to use this series of steps we only have to call the function.

Notice that a function consists of the keyword `def` (short for "define"), followed by the function name, followed by a sequence of parameters enclosed in parentheses, then a colon. The following lines contain an indented block of code, the **function body**.

Using functions has the benefit of saving space in our program. More importantly, our choice of name for the function helps make the program *readable*. In the case of the above example, whenever our program needs to read cleaned-up text from a file we don't have to clutter the program with four lines of code, we simply need to call `get_text()`. This naming helps to provide some "semantic interpretation" — it helps a reader of our program to see what the program "means".

Notice that the above function definition contains a string. The first string inside a function definition is called a **docstring**. Not only does it document the purpose of the function to someone reading the code, it is accessible to a programmer who has loaded the code from a file:

```
>>> help(get_text)
get_text(file)
    Read text from a file, normalizing whitespace and stripping HTML markup
```

We have seen that functions help to make our work reusable and readable. They also help make it *reliable*. When we re-use code that has already been developed and tested, we can be more confident

that it handles a variety of cases correctly. We also remove the risk that we forget some important step, or introduce a bug. The program which calls our function also has increased reliability. The author of that program is dealing with a shorter program, and its components behave transparently.

- [More: overview of section]

6.4.1 Function arguments

- multiple arguments
- named arguments
- default values

Python is a **dynamically typed** language. It does not force us to declare the type of a variable when we write a program. This feature is often useful, as it permits us to define functions that are flexible about the type of their arguments. For example, a tagger might expect a sequence of words, but it wouldn't care whether this sequence is expressed as a list, a tuple, or an iterator.

However, often we want to write programs for later use by others, and want to program in a defensive style, providing useful warnings when functions have not been invoked correctly. Observe that the `tag()` function in Listing 6.9 behaves sensibly for string arguments, but that it does not complain when it is passed a dictionary.

Listing 20 A tagger which tags anything

```
def tag(word):
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'

>>> tag('the')
'dt'
>>> tag('dog')
'nn'
>>> tag({'lexeme':'turned', 'pos':'VBD', 'pron':['t3:nd', 't3`nd']})
'nn'
```

It would be helpful if the author of this function took some extra steps to ensure that the `word` parameter of the `tag()` function is a string. A naive approach would be to check the type of the argument and return a diagnostic value, such as Python's special empty value, `None`, as shown in Listing 6.10.

However, this approach is dangerous because the calling program may not detect the error, and the diagnostic return value may be propagated to later parts of the program with unpredictable consequences. A better solution is shown in Listing 6.11.

This produces an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. (We will see an even better approach, known as "duck typing" in Chapter 10.)

Another aspect of defensive programming concerns the return statement of a function. In order to be confident that all execution paths through a function lead to a return statement, it is best to have

Listing 21 A tagger which only tags strings

```
def tag(word):
    if not type(word) is str:
        return None
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'
```

Listing 22 A tagger which generates an error message when not passed a string

```
def tag(word):
    if not type(word) is str:
        raise ValueError, "argument to tag() must be a string"
    if word in ['a', 'the', 'all']:
        return 'dt'
    else:
        return 'nn'
```

a single return statement at the end of the function definition. This approach has a further benefit: it makes it more likely that the function will only return a single type. Thus, the following version of our `tag()` function is safer:

```
>>> def tag(word):
...     result = 'nn'                          # default value, a string
...     if word in ['a', 'the', 'all']:        # in certain cases...
...         result = 'dt'                      #   overwrite the value
...     return result                          # all paths end here
```

A return statement can be used to pass multiple values back to the calling program, by packing them into a tuple. Here we define a function that returns a tuple consisting of the average word length of a sentence, and the inventory of letters used in the sentence. It would have been clearer to write two separate functions.

Of course, functions do not need to have a return statement at all. Some functions do their work as a side effect, printing a result, modifying a file, or updating the contents of a parameter to the function. Consider the following three sort functions; the last approach is dangerous because a programmer could use it without realizing that it had modified its input.

```
>>> def my_sort1(l):       # good: modifies its argument, no return value
...     l.sort()
>>> def my_sort2(l):       # good: doesn't touch its argument, returns value
...     return sorted(l)
>>> def my_sort3(l):       # bad: modifies its argument and also returns it
...     l.sort()
...     return l
```

6.4.2 An Important Subtlety

Back in Section 6.2.1 you saw that in Python, assignment works on values, but that the value of a structured object is a reference to that object. The same is true for functions. Python interprets function

parameters as values (this is known as **call-by-value**). Consider Listing 6.12. Function `set_up()` has two parameters, both of which are modified inside the function. We begin by assigning an empty string to `w` and an empty dictionary to `p`. After calling the function, `w` is unchanged, while `p` is changed:

Listing 23

```
def set_up(word, properties):
    word = 'cat'
    properties['pos'] = 'noun'

>>> w = ''
>>> p = {}
>>> set_up(w, p)
>>> w
''
>>> p
{'pos': 'noun'}
```

To understand why `w` was not changed, it is necessary to understand call-by-value. When we called `set_up(w, p)`, the value of `w` (an empty string) was assigned to a new variable `word`. Inside the function, the value of `word` was modified. However, that had no affect on the external value of `w`. This parameter passing is identical to the following sequence of assignments:

```
>>> w = ''
>>> word = w
>>> word = 'cat'
>>> w
''
```

In the case of the structured object, matters are quite different. When we called `set_up(w, p)`, the value of `p` (an empty dictionary) was assigned to a new local variable `properties`. Since the value of `p` is an object reference, both variables now reference the same memory location. Modifying something inside `properties` will also change `p`, just as if we had done the following sequence of assignments:

```
>>> p = {}
>>> properties = p
>>> properties['pos'] = 'noun'
>>> p
{'pos': 'noun'}
```

Thus, to understand Python's call-by-value parameter passing, it is enough to understand Python's assignment operation. We will address some closely related issues in our later discussion of variable scope (Section 10.1.1).

6.4.3 Functional Decomposition

Well-structured programs usually make extensive use of functions. When a block of program code grows longer than 10-20 lines, it is a great help to readability if the code is broken up into one or more functions, each one having a clear purpose. This is analogous to the way a good essay is divided into paragraphs, each expressing one main idea.

Functions provide an important kind of *abstraction*. They allow us to group multiple actions into a single, complex action, and associate a name with it. (Compare this with the way we combine the actions of *go* and *bring back* into a single more complex action *fetch*.) When we use functions, the main program can be written at a higher level of abstraction, making its structure transparent, e.g.

```
>>> data = load_corpus()
>>> results = analyze(data)
>>> present(results)
```

Appropriate use of functions makes programs more readable and maintainable. Additionally, it becomes possible to reimplement a function — replacing the function's body with more efficient code — without having to be concerned with the rest of the program.

Consider the `freq_words` function in Listing 6.13. It updates the contents of a frequency distribution that is passed in as a parameter, and it also prints a list of the *n* most frequent words.

Listing 24

```
def freq_words(url, freqdist, n):
    from nltk_lite.corpora import web
    for word in web.raw(url):
        freqdist.inc(word.lower())
    print freqdist.sorted_samples()[:n]

>>> constitution = "http://www.archives.gov/national-archives-experience/charters/c
>>> from nltk_lite.probability import FreqDist
>>> fd = FreqDist()
>>> freq_words(constitution, fd, 20)
['the', ',', 'of', 'and', 'shall', '.', 'be', 'to', 'in', 'states', 'or',
';', 'united', 'a', 'state', 'by', 'for', 'any', 'president', 'which']
```

This function has a number of problems. The function has two side-effects: it modifies the contents of its second parameter, and it prints a selection of the results it has computed. The function would be easier to understand and to reuse elsewhere if we initialize the `FreqDist()` object inside the function (in the same place it is populated), and if we moved the selection and display of results to the calling program. In Listing 6.14 we **refactor** this function, and simplify its interface by providing a single `url` parameter.

6.4.4 Documentation (notes)

- some guidelines for literate programming (e.g. variable and function naming)
- documenting functions (user-level and developer-level documentation)

6.4.5 Functions as Arguments

So far the arguments we have passed into functions have been simple objects like strings, or structured objects like lists. These arguments allow us to parameterise the behavior of a function. As a result, functions are very flexible and powerful abstractions, permitting us to repeatedly apply the *same operation* on *different data*. Python also lets us pass a function as an argument to another function. Now

Listing 25

```
def freq_words(url):
    from nltk_lite.corpora import web
    from nltk_lite.probability import FreqDist
    freqdist = FreqDist()
    for word in web.raw(url):
        freqdist.inc(word.lower())
    return freqdist

>>> fd = freq_words(constitution)
>>> print fd.sorted_samples()[:20]
['the', ',', 'of', 'and', 'shall', '.', 'be', 'to', 'in', 'states', 'or',
';', 'united', 'a', 'state', 'by', 'for', 'any', 'president', 'which']
```

we can abstract out the operation, and apply a *different operation* on the *same data*. As the following examples show, we can pass the built-in function `len()` or a user-defined function `last_letter()` as parameters to another function:

```
>>> def extract_property(prop):
...     words = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
...     return [prop(word) for word in words]
>>> extract_property(len)
[3, 3, 4, 4, 3, 9]
>>> def last_letter(word):
...     return word[-1]
>>> extract_property(last_letter)
['e', 'g', 'e', 'n', 'e', 'r']
```

Surprisingly, `len` and `last_letter` are objects that can be passed around like lists and dictionaries. Notice that parentheses are only used after a function name if we are invoking the function; when we are simply passing the function around as an object these are not used.

Python provides us with one more way to define functions as arguments to other functions, so-called **lambda expressions**. Supposing there was no need to use the above `last_letter()` function in multiple places, we can equivalently write the following:

```
>>> extract_property(lambda w: w[-1])
['e', 'g', 'e', 'n', 'e', 'r']
```

Our next example illustrates passing a function to the `sorted()` function. When we call the latter with a single argument (the list to be sorted), it uses the built-in lexicographic comparison function `cmp()`. However, we can supply our own sort function, e.g. to sort by decreasing length.

```
>>> words = 'I turned off the spectroroute'.split()
>>> sorted(words)
['I', 'off', 'spectroroute', 'the', 'turned']
>>> sorted(words, cmp)
['I', 'off', 'spectroroute', 'the', 'turned']
>>> sorted(words, lambda x, y: cmp(len(y), len(x)))
['spectroroute', 'turned', 'off', 'the', 'I']
```

In 6.2.5 we saw an example of filtering out some items in a list comprehension, using an `if` test. Similarly, we can restrict a list to just the lexical words, using `[word for word in sent if is_lexical(word)]`. This is a little cumbersome as it mentions the `word` variable three times. A more compact way to express the same thing is as follows.

```
>>> def is_lexical(word):
...     return word.lower() not in ('a', 'an', 'the', 'that', 'to')
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> filter(is_lexical, sent)
['dog', 'gave', 'John', 'newspaper']
```

The function `is_lexical(word)` returns `True` just in case `word`, when normalized to lowercase, is not in the given list. This function is itself used as an argument to `filter()`; in Python, functions are just another kind of object that can be passed around a program; we will return to this in Section 6.4.5. The `filter()` function applies its first argument (a function) to each item of its second (a sequence), only passing it through if the function returns true for that item. Thus `filter(f, seq)` is equivalent to `[item for item in seq if apply(f,item) == True]`.

Another helpful function, which like `filter()` applies a function to a sequence, is `map()`. Here is a simple way to find the average length of a sentence in a section of the Brown Corpus:

```
>>> average(map(len, brown.raw('a')))
21.7461072664
```

Instead of `len()`, we could have passed in any other function we liked:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> def is_vowel(letter):
...     return letter in "AEIOUaeiou"
>>> def vowelcount(word):
...     return len(filter(is_vowel, word))
>>> map(vowelcount, sent)
[1, 1, 2, 1, 1, 3]
```

Instead of using `filter()` to call a named function `is_vowel`, we can define a lambda expression as follows:

```
>>> map(lambda w: len(filter(lambda c: c in "AEIOUaeiou", w)), sent)
[1, 1, 2, 1, 1, 3]
```

6.4.6 Exercises

1. ☼ Review the answers that you gave for the exercises in 6.2, and rewrite the code as one or more functions.

2. ◑ In this section we saw examples of some special functions such as `filter()` and `map()`. Other functions in this family are `zip()` and `reduce()`. Find out what these do, and write some code to try them out. What uses might they have in language processing?

3. ◑ Write a function that takes a list of words (containing duplicates) and returns a list of words (with no duplicates) sorted by decreasing frequency. E.g. if the input list contained 10 instances of the word `table` and 9 instances of the word `chair`, then `table` would appear before `chair` in the output list.

4. ◑ As you saw, `zip()` combines two lists into a single list of pairs. What happens when the lists are of unequal lengths? Define a function `myzip()` which does something different with unequal lists.

5. ◑ Import the `itemgetter()` function from the `operator` module in Python's standard library (i.e. `from operator import itemgetter`). Create a list `words` containing several words. Now try calling: `sorted(words, key=itemgetter(1))`, and `sorted(words, key=itemgetter(-1))`. Explain what `itemgetter()` is doing.

6.5 Algorithm Design Strategies

A major part of algorithmic problem solving is selecting or adapting an appropriate algorithm for the problem at hand. Whole books are written on this topic (e.g. [Levitin, 2004]) and we only have space to introduce some key concepts and elaborate on the approaches that are most prevalent in natural language processing.

The best known strategy is known as **divide-and-conquer**. We attack a problem of size *n* by dividing it into two problems of size *n/2*, solve these problems, and combine their results into a solution of the original problem. Figure 6.4 illustrates this approach for sorting a list of words.

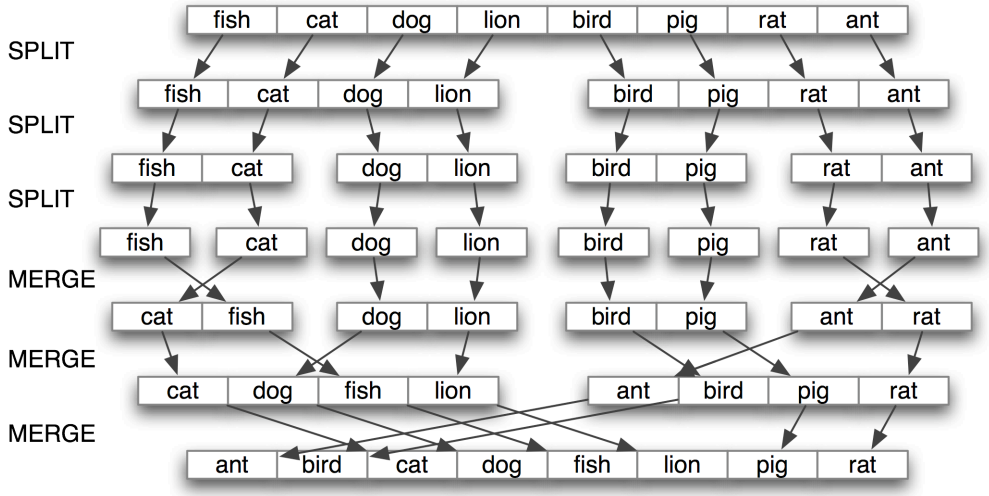


Figure 6.4: Sorting by Divide-and-Conquer (Mergesort)

Another strategy is **decrease-and-conquer**. In this approach, a small amount of work on a problem of size *n* permits us to reduce it to a problem of size *n/2*. Figure 6.5 illustrates this approach for the problem of finding the index of an item in a sorted list.

A third well-known strategy is **transform-and-conquer**. We attack a problem by transforming it into an instance of a problem we already know how to solve. For example, in order to detect duplicates entries in a list, we can **pre-sort** the list, then look for adjacent identical items, as shown in Listing 6.15. Our approach to n-gram chunking in Section 5.5 is another case of transform and conquer (why?).

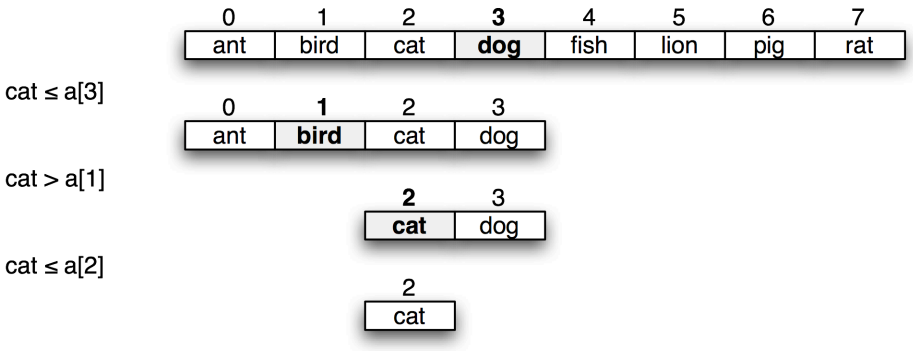


Figure 6.5: Searching by Decrease-and-Conquer (Binary Search)

Listing 26 Presorting a list for duplicate detection

```
def duplicates(words):
    prev = None
    dup = []
    for word in sorted(words):
        if word == prev:
            dup.append(word)
        else:
            prev = word
    return dup

>>> duplicates(['cat', 'dog', 'cat', 'pig', 'dog', 'cat', 'ant', 'cat'])
['cat', 'dog']
```

6.5.1 Recursion (notes)

We first saw recursion in Chapter 3, in a function that navigated the hypernym hierarchy of WordNet...

Iterative solution:

```
>>> def factorial(n):
...     result = 1
...     for i in range(n+1):
...         result *= i
...     return result
```

Recursive solution (base case, induction step)

```
>>> def factorial(n):
...     if n == 1:
...         return n
...     else:
...         return n * factorial(n-1)
```

[Simple example of recursion on strings.]

Generating all permutations of words, to check which ones are grammatical:

```
>>> def perms(seq):
...     if len(seq) <= 1:
...         yield seq
...     else:
...         for perm in perms(seq[1:]):
...             for i in range(len(perm)+1):
...                 yield perm[:i] + seq[0:1] + perm[i:]
>>> list(perms(['police', 'fish', 'cream']))
[['police', 'fish', 'cream'], ['fish', 'police', 'cream'],
 ['fish', 'cream', 'police'], ['police', 'cream', 'fish'],
 ['cream', 'police', 'fish'], ['cream', 'fish', 'police']]
```

6.5.2 Deeply Nested Objects (notes)

We can use recursive functions to build deeply-nested objects. Building a letter trie, Listing 6.16.

6.5.3 Dynamic Programming

Dynamic programming is a general technique for designing algorithms which is widely used in natural language processing. The term 'programming' is used in a different sense to what you might expect, to mean planning or scheduling. Dynamic programming is used when a problem contains overlapping sub-problems. Instead of computing solutions to these sub-problems repeatedly, we simply store them in a lookup table. In the remainder of this section we will introduce dynamic programming, but in a rather different context to syntactic parsing.

Pingala was an Indian author who lived around the 5th century B.C., and wrote a treatise on Sanscrit prosody called the *Chandas Shastra*. Virahanka extended this work around the 6th century A.D., studying the number of ways of combining short and long syllables to create a meter of length n. He found, for example, that there are five ways to construct a meter of length 4: V_4 = *{LL, SSL, SLS, LSS, SSSS}*. Observe that we can split V_4 into two subsets, those starting with L and those starting with S, as shown in (14).

Listing 27 Building a Letter Trie

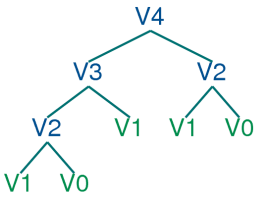
```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value

>>> trie = {}
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
>>> trie['c']['h']
{'a': {'t': {'value': 'cat'}}, 'i': {'e': {'n': {'value': 'dog'}}}}
>>> trie['c']['h']['a']['t']['value']
'cat'
>>> pprint(trie)
{'c': {'h': {'a': {'t': {'value': 'cat'}},
             'i': {'e': {'n': {'value': 'dog'}}}}}}
```

(14)

V_4 =
LL, LSS
i.e. L prefixed to each item of V_2 = {L, SS}
SSL, SLS, SSSS
i.e. S prefixed to each item of V_3 = {SL, LS, SSS}

With this observation, we can write a little recursive function called `virahanka1()` to compute these meters, shown in Listing 6.17. Notice that, in order to compute V_4 we first compute V_3 and V_2. But to compute V_3, we need to first compute V_2 and V_1. This **call structure** is depicted in (15).

(15)



As you can see, V_2 is computed twice. This might not seem like a significant problem, but it turns out to be rather wasteful as n gets large: to compute V_{20} using this recursive technique, we would compute V_2 4,181 times; and for V_{40} we would compute V_2 63,245,986 times! A much better alternative is to store the value of V_2 in a table and look it up whenever we need it. The same goes for other values, such as V_3 and so on. Function `virahanka2()` implements a dynamic programming approach to the problem. It works by filling up a table (called `lookup`) with solutions to *all* smaller instances of the problem, stopping as soon as we reach the value we're interested in. At this point we read off the value and return it. Crucially, each sub-problem is only ever solved once.

Notice that the approach taken in `virahanka2()` is to solve smaller problems on the way to solving larger problems. Accordingly, this is known as the **bottom-up** approach to dynamic programming.

Listing 28 Three Ways to Compute Sansrit Meter

```
def virahanka1(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l

def virahanka2(n):
    lookup = [[""], ["S"]]
    for i in range(n-1):
        s = ["S" + prosody for prosody in lookup[i+1]]
        l = ["L" + prosody for prosody in lookup[i]]
        lookup.append(s + l)
    return lookup[n]

def virahanka3(n, lookup={0:[""], 1:["S"]}):
    if n not in lookup:
        s = ["S" + prosody for prosody in virahanka3(n-1)]
        l = ["L" + prosody for prosody in virahanka3(n-2)]
        lookup[n] = s + l
    return lookup[n]

>>> virahanka1(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka2(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka3(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

Unfortunately it turns out to be quite wasteful for some applications, since it may compute solutions to sub-problems that are never required for solving the main problem. This wasted computation can be avoided using the **top-down** approach to dynamic programming, which is illustrated in the function `virahanka3()` in Listing 6.17. Unlike the bottom-up approach, this approach is recursive. It avoids the huge wastage of `virahanka1()` by checking whether it has previously stored the result. If not, it computes the result recursively and stores it in the table. The last step is to return the stored result.

This concludes our brief introduction to dynamic programming. We will encounter it again in Chapter 8.

> **Note**
>
> Dynamic programming is a kind of **memoization**. A memoized function stores results of previous calls to the function along with the supplied parameters. If the function is subsequently called with those parameters, it returns the stored result instead of recalculating it.

6.5.4 Timing (notes)

We can easily test the efficiency gains made by the use of dynamic programming, or any other putative performance enhancement, using the `timeit` module:

```
>>> from timeit import Timer
>>> Timer("PYTHON CODE", "INITIALIZATION CODE").timeit()
```

[MORE]

6.5.5 Exercises

1. ◑ Write a recursive function `lookup(trie, key)` that looks up a key in a trie, and returns the value it finds. Extend the function to return a word when it is uniquely determined by its prefix (e.g. `vanguard` is the only word which starts with `vang-`, so `lookup(trie, 'vang')` should return the same thing as `lookup(trie, 'vanguard')`).

2. ◑ Read about string edit distance and the Levenshtein Algorithm. Try the implementation provided in `nltk_lite.utilities.edit_dist()`. How is this using dynamic programming? Does it use the bottom-up or top-down approach?

3. ◑ The Catalan numbers arise in many applications of combinatorial mathematics, including the counting of parse trees (Chapter 8). The series can be defined as follows: $C_0 = 1$, and $C_{n+1} = \Sigma_{0..n} (C_i C_{n-i})$.

   a) Write a recursive function to compute nth Catalan number C_n

   b) Now write another function that does this computation using dynamic programming

   c) Use the `timeit` module to compare the performance of these functions as n increases.

4. ★ Write a recursive function that pretty prints a trie in alphabetically sorted order, as follows

   chat: 'cat' --ien: 'dog' -???: ???

5. ★ Write a recursive function that processes text, locating the uniqueness point in each word, and discarding the remainder of each word. How much compression does this give? How readable is the resulting text?

6.6 Conclusion

[TO DO]

6.7 Further Reading

[Harel, 2004]
[Levitin, 2004]
http://docs.python.org/lib/typesseq-strings.html

Chapter 7

Grammars and Parsing

7.1 Introduction

Early experiences with the kind of grammar taught in school are sometimes perplexing. Your written work might have been graded by a teacher who red-lined all the grammar errors they wouldn't put up with. Like the plural pronoun or the dangling preposition in the last sentence, or sentences like this one which lack a main verb. If you learnt English as a second language, you might have found it difficult to discover which of these errors need to be fixed (or *needs* to be fixed?). Correct punctuation is an obsession for many writers and editors. It is easy to find cases where changing punctuation changes meaning. In the following example, the interpretation of a relative clause as restrictive or non-restrictive depends on the presence of commas alone:

(16a) The presidential candidate, who was extremely popular, smiled broadly.

(16b) The presidential candidate who was extremely popular smiled broadly.

In (16a), we assume there is just one presidential candidate, and say two things about her: that she was popular and that she smiled. In (16b), on the other hand, we use the description *who was extremely popular* as a means of identifying for the hearer which of several candidates we are referring to.

It is clear that some of these rules are important. However, others seem to be vestiges of antiquated style. Consider the injunction that *however* — when used to mean *nevertheless* — must not appear at the start of a sentence. Pullum argues that Strunk and White [Strunk and White, 1999] were merely insisting that English usage should conform to "an utterly unimportant minor statistical detail of style concerning adverb placement in the literature they knew" [Pullum, 2005]. This is a case where, a *descriptive* observation about language use became a *prescriptive* requirement. In NLP we usually discard such prescriptions, and use grammar to formalize observations about language as it is used, particularly as it is used in corpora.

In this chapter we present the fundamentals of syntax, focusing on constituency and tree representations, before describing the formal notation of context free grammar. Next we present parsers as an automatic way to associate syntactic structures with sentences. Finally, we give a detailed presentation of simple top-down and bottom-up parsing algorithms available in NLTK. Before launching into the theory we present some more naive observations about grammar, for the benefit of readers who do not have a background in linguistics.

7.2 More Observations about Grammar

Another function of a grammar is to explain our observations about ambiguous sentences. Even when the individual words are unambiguous, we can put them together to create ambiguous sentences, as in (17).

(17a) Fighting animals could be dangerous.

(17b) Visiting relatives can be tiresome.

A grammar will be able to assign two structures to each sentence, accounting for the two possible interpretations.

Perhaps another kind of syntactic variation, word order, is easier to understand. We know that the two sentences *Kim likes Sandy* and *Sandy likes Kim* have different meanings, and that *likes Sandy Kim* is simply ungrammatical. Similarly, we know that the following two sentences are equivalent:

(18a) The farmer *loaded* the cart with sand

(18b) The farmer *loaded* sand into the cart

However, consider the semantically similar verbs *filled* and *dumped*. Now the word order cannot be altered (ungrammatical sentences are prefixed with an asterisk.)

(19a) The farmer *filled* the cart with sand

(19b) *The farmer *filled* sand into the cart

(19c) *The farmer *dumped* the cart with sand

(19d) The farmer *dumped* sand into the cart

A further notable fact is that we have no difficulty accessing the meaning of sentences we have never encountered before. It is not difficult to concoct an entirely novel sentence, one that has probably never been used before in the history of the language, and yet all speakers of the language will agree about its meaning. In fact, the set of possible sentences is infinite, given that there is no upper bound on length. Consider the following passage from a children's story, containing a rather impressive sentence:

> You can imagine Piglet's joy when at last the ship came in sight of him. In after-years he liked to think that he had been in Very Great Danger during the Terrible Flood, but the only danger he had really been in was the last half-hour of his imprisonment, when Owl, who had just flown up, sat on a branch of his tree to comfort him, and told him a very long story about an aunt who had once laid a seagull's egg by mistake, and the story went on and on, rather like this sentence, until Piglet who was listening out of his window without much hope, went to sleep quietly and naturally, slipping slowly out of the window towards the water until he was only hanging on by his toes, at which moment, luckily, a sudden loud squawk from Owl, which was really part of the story, being what his aunt said, woke the Piglet up and just gave him time to jerk himself back into safety and say, "How interesting, and did she?" when -- well, you can imagine his joy when at last he saw the good ship, Brain of Pooh (Captain, C. Robin; 1st Mate, P. Bear) coming over the sea to rescue him... (from A.A. Milne *In which Piglet is Entirely Surrounded by Water*)

Our ability to produce and understand entirely new sentences, of arbitrary length, demonstrates that the set of well-formed sentences in English is infinite. The same case can be made for any human language.

This chapter presents grammars and parsing, as the formal and computational methods for investigating and modelling the linguistic phenomena we have been touching on (or tripping over). As we shall see, patterns of well-formedness and ill-formedness in a sequence of words can be understood with respect to the underlying **phrase structure** of the sentences. We can develop formal models of these structures using grammars and parsers. As before, the motivation is natural language *understanding*. How much more of the meaning of a text can we access when we can reliably recognize the linguistic structures it contains? Having read in a text, can a program 'understand' it enough to be able to answer simple questions about "what happened" or "who did what to whom" Also as before, we will develop simple programs to process annotated corpora and perform useful tasks.

7.3 What's the Use of Syntax?

Earlier chapters focussed on words: how to identify them, how to analyse their morphology, and how to assign them to classes via part-of-speech tags. We have also seen how to identify recurring sequences of words (i.e. n-grams). Nevertheless, there seem to be linguistic regularities which cannot be described simply in terms of n-grams.

In this section we will see why it is useful to have some kind of syntactic representation of sentences. In particular, we will see that there are systematic aspects of meaning which are much easier to capture once we have established a level of syntactic structure.

7.3.1 Syntactic Ambiguity

We have seen that sentences can be ambiguous. If we overheard someone say *I went to the bank*, we wouldn't know whether it was a river bank or a financial institution. This ambiguity concerns the meaning of the word *bank*, and is a kind of **lexical ambiguity**.

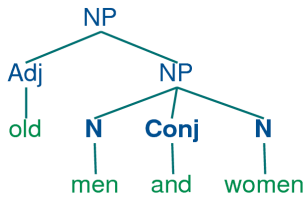
However, other kinds of ambiguity cannot be explained in terms of ambiguity of specific words. Consider a phrase involving an adjective with a conjunction: *old men and women*. Does *old* have wider scope than *and*, or is it the other way round? In fact, both interpretations are possible, and we can represent the different scopes using parentheses:

(20a) old (men and women)

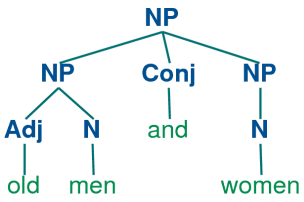
(20b) (old men) and women

One convenient way of representing this scope difference at a structural level is by means of a **tree diagram**, as shown in (21).

(21a)



(21b)



Note that linguistic trees grow upside down: the node labeled S is the **root** of the tree, while the **leaves** of the tree are labeled with the words.

In NLTK, you can easily produce trees like this yourself with the following commands:

```
>>> from nltk_lite.parse import bracket_parse
>>> tree = bracket_parse('(NP (Adj old) (NP (N men) (Conj and) (N women)))')
>>> tree.draw()                    # doctest: +SKIP
```

We can construct other examples of syntactic ambiguity involving the coordinating conjunctions *and* and *or*, e.g. *Kim left or Dana arrived and everyone cheered*. We can describe this ambiguity in terms of the relative semantic **scope** of *or* and *and*.

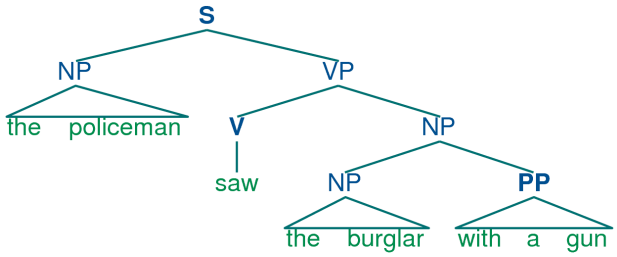
For our third illustration of ambiguity, we look at prepositional phrases. Consider a sentence like: *I saw the man with a telescope*. Who has the telescope? To clarify what is going on here, consider the following pair of sentences:

(22a) The policeman saw a burglar *with a gun*. (not some other burglar)

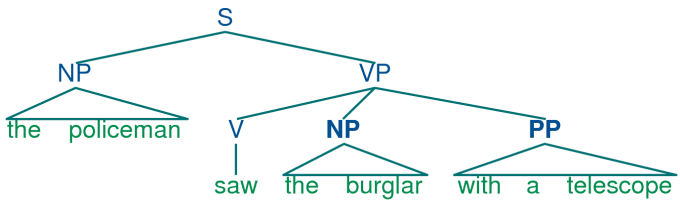
(22b) The policeman saw a burglar *with a telescope*. (not with his naked eye)

In both cases, there is a prepositional phrase introduced by *with*. In the first case this phrase modifies the noun *burglar*, and in the second case it modifies the verb *saw*. We could again think of this in terms of scope: does the prepositional phrase (PP) just have scope over the NP *a burglar*, or does it have scope over the whole verb phrase? As before, we can represent the difference in terms of tree structure:

(23a)



(23b)



In (23)a, the PP attaches to the NP, while in (23)b, the PP attaches to the VP.

We can generate these trees in Python as follows:

```
>>> s1 = '(S (NP the policeman) (VP (V saw) (NP (NP the burglar) (PP with a gun))))'
>>> s2 = '(S (NP the policeman) (VP (V saw) (NP the burglar) (PP with a telescope)))'
>>> tree1 = bracket_parse(s1)
>>> tree2 = bracket_parse(s2)
```

We can discard the structure to get the list of **leaves**, and we can confirm that both trees have the same leaves. We can also see that the trees have different **heights** (given by the number of nodes in the longest branch of the tree, starting at S and descending to the words):

```
>>> tree1.leaves()
['the', 'policeman', 'saw', 'the', 'burglar', 'with', 'a', 'gun']
>>> tree1.leaves() == tree2.leaves()
True
>>> tree1.height() == tree2.height()
False
```

In general, how can we determine whether a prepositional phrase modifies the preceding noun or verb? This problem is known as **prepositional phrase attachment ambiguity**. The **Prepositional Phrase Attachment Corpus** makes it possible for us to study this question systematically. The corpus is derived from the IBM-Lancaster Treebank of Computer Manuals and from the Penn Treebank, and distills out only the essential information about PP attachment. Consider the sentence from the WSJ in (24a). The corresponding line in the Prepositional Phrase Attachment Corpus is shown in (24b).

(24a) Four of the five surviving workers have asbestos-related diseases, including three with recently diagnosed cancer.

(24b)
```
16 including three with cancer N
```

That is, it includes an identifier for the original sentence, the head of the relevant verb phrase (i.e., *including*), the head of the verb's NP object (*three*), the preposition (*with*), and the head noun within the prepositional phrase (*cancer*). Finally, it contains an "attachment" feature (N or V) to indicate whether the prepositional phrase attaches to (modifies) the noun phrase or the verb phrase. Here are some further examples:

(25)
```
47830 allow visits between families N
47830 allow visits on peninsula V
42457 acquired interest in firm N
42457 acquired interest in 1986 V
```

The PP attachments in (25) can also be made explicit by using phrase groupings as in (26).

(26)
```
allow (NP visits (PP between families))
allow (NP visits) (PP on peninsula)
acquired (NP interest (PP in firm))
acquired (NP interest) (PP in 1986)
```

Observe in each case that the argument of the verb is either a single complex expression `(visits (between families))` or a pair of simpler expressions `(visits) (on peninsula)`.

We can access the Prepositional Phrase Attachment Corpus from NLTK as follows:

```
>>> from nltk_lite.corpora import ppattach, extract
>>> from pprint import pprint
>>> item = extract(9, ppattach.dictionary('training'))
```

```
>>> pprint(item)
{'attachment': 'N',
 'noun1': 'three',
 'noun2': 'cancer',
 'prep': 'with',
 'sent': '16',
 'verb': 'including'}
```

If we go back to our first examples of PP attachment ambiguity, it appears as though it is the PP itself (e.g., *with a gun* versus *with a telescope*) that determines the attachment. However, we can use this corpus to find examples where other factors come into play. For example, it appears that the verb is the key factor in (27).

(27)
```
8582 received offer from group V
19131 rejected offer from group N
```

7.3.2 Constituency

We claimed earlier that one of the motivations for building syntactic structure was to help make explicit how a sentence says "who did what to whom". Let's just focus for a while on the "who" part of this story: in other words, how can syntax tell us what the subject of a sentence is? At first, you might think this task is rather simple — so simple indeed that we don't need to bother with syntax. In a sentence such as *The fierce dog bit the man* we know that it is the dog that is doing the biting. So we could say that the noun phrase immediately preceding the verb is the subject of the sentence. And we might try to make this more explicit in terms of sequences part-of-speech tags. Let's try to come up with a simple definition of *noun phrase*; we might start off with something like this, based on our knowledge of noun phrase chunking (Chapter 5):

(28) DT JJ* NN

We're using regular expression notation here in the form of JJ* to indicate a sequence of zero or more JJs. So this is intended to say that a noun phrase can consist of a determiner, possibly followed by some adjectives, followed by a noun. Then we can go on to say that if we can find a sequence of tagged words like this that precedes a word tagged as a verb, then we've identified the subject. But now think about this sentence:

(29) The child with a fierce dog bit the man.

This time, it's the child that is doing the biting. But the tag sequence preceding the verb is:

(30) DT NN IN DT JJ NN

Our previous attempt at identifying the subject would have incorrectly come up with *the fierce dog* as the subject. So our next hypothesis would have to be a bit more complex. For example, we might say that the subject can be identified as any string matching the following pattern before the verb:

(31) DT JJ* NN (IN DT JJ* NN)*

In other words, we need to find a noun phrase followed by zero or more sequences consisting of a preposition followed by a noun phrase. Now there are two unpleasant aspects to this proposed solution. The first is aesthetic: we are forced into repeating the sequence of tags (DT JJ* NN) that constituted our initial notion of noun phrase, and our initial notion was in any case a drastic simplification. More worrying, this approach still doesn't work! For consider the following example:

(32) The seagull that attacked the child with the fierce dog bit the man.

This time the seagull is the culprit, but it won't be detected as subject by our attempt to match sequences of tags. So it seems that we need a richer account of how words are *grouped* together into patterns, and a way of referring to these groupings at different points in the sentence structure. This idea of grouping is often called syntactic **constituency**.

As we have just seen, a well-formed sentence of a language is more than an arbitrary sequence of words from the language. Certain kinds of words usually go together. For instance, determiners like *the* are typically followed by adjectives or nouns, but not by verbs. Groups of words form intermediate structures called phrases or **constituents**. These constituents can be identified using standard syntactic tests, such as substitution, movement and coordination. For example, if a sequence of words can be replaced with a pronoun, then that sequence is likely to be a constituent. According to this test, we can infer that the italicised string in the following example is a constituent, since it can be replaced by *they*:

(33a) *Ordinary daily multivitamin and mineral supplements* could help adults with diabetes fight off some minor infections.

(33b) *They* could help adults with diabetes fight off some minor infections.

In order to identify whether a phrase is the subject of a sentence, we can use the construction called **Subject-Auxiliary Inversion** in English. This construction allows us to form so-called Yes-No Questions. That is, corresponding to the statement in (34a), we have the question in (34b):

(34a) All the cakes have been eaten.

(34b) Have *all the cakes* been eaten?

Roughly speaking, if a sentence already contains an auxiliary verb, such as *has* in (34a), then we can turn it into a Yes-No Question by moving the auxiliary verb 'over' the subject noun phrase to the front of the sentence. If there is no auxiliary in the statement, then we insert the appropriate form of *do* as the fronted auxiliary and replace the tensed main verb by its base form:

(35a) The fierce dog bit the man.

(35b) Did *the fierce dog* bite the man?

As we would hope, this test also confirms our earlier claim about the subject constituent of (32):

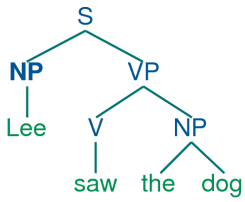
(36) Did *the seagull that attacked the child with the fierce dog* bite the man?

To sum up then, we have seen that the notion of constituent brings a number of benefits. By having a constituent labeled NOUN PHRASE, we can provide a unified statement of the classes of word that constitute that phrase, and reuse this statement in describing noun phrases wherever they occur in the sentence. Second, we can use the notion of a noun phrase in defining the subject of sentence, which in turn is a crucial ingredient in determining the "who does what to whom" aspect of meaning.

7.3.3 More on Trees

A tree is a set of connected nodes, each of which is labeled with a category. It common to use a 'family' metaphor to talk about the relationships of nodes in a tree: for example, S is the **parent** of VP; conversely VP is a **daughter** (or **child**) of S. Also, since NP and VP are both daughters of S, they are also **sisters**. Here is an example of a tree:

(37)



Although it is helpful to represent trees in a graphical format, for computational purposes we usually need a more text-oriented representation. One standard method (used in the Penn Treebank) is to use a combination of bracket and labels to indicate the structure, as shown here:

```
(S
   (NP  'Lee')
   (VP
      (V 'saw')
      (NP
         (Det 'the')
         (N  'dog'))))
```

The conventions for displaying trees in NLTK are similar:

```
(S: (NP: 'Lee') (VP: (V: 'saw') (NP: 'the' 'dog')))
```

In such trees, the node value is a string containing the tree's constituent type (e.g., NP or VP), while the children encode the hierarchical contents of the tree.

Although we will focus on syntactic trees, trees can be used to encode *any* homogeneous hierarchical structure that spans a sequence of linguistic forms (e.g. morphological structure, discourse structure). In the general case, leaves and node values do not have to be strings.

In NLTK, trees are created with the `Tree` constructor, which takes a node value and a list of zero or more children. Here's a couple of simple trees:

```
>>> from nltk_lite.parse import Tree
>>> tree1 = Tree('NP', ['John'])
>>> print tree1
(NP: 'John')
>>> tree2 = Tree('NP', ['the', 'man'])
>>> print tree2
(NP: 'the' 'man')
```

We can incorporate these into successively larger trees as follows:

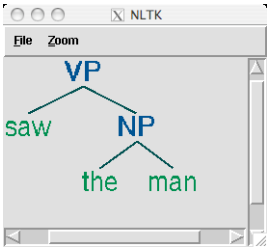
```
>>> tree3 = Tree('VP', ['saw', tree2])
>>> tree4 = Tree('S', [tree1, tree3])
>>> print tree4
(S: (NP: 'John') (VP: 'saw' (NP: 'the' 'man')))
```

Here are some of the methods available for tree objects:

```
>>> tree4[1]
(VP: 'saw' (NP: 'the' 'man'))
>>> tree4[1].node
'VP'
>>> tree4.leaves()
['John', 'saw', 'the', 'man']
>>> tree[1,1,0]
'saw'
```

The printed representation for complex trees can be difficult to read. In these cases, the `draw` method can be very useful. It opens a new window, containing a graphical representation of the tree. The tree display window allows you to zoom in and out; to collapse and expand subtrees; and to print the graphical representation to a postscript file (for inclusion in a document).

```
>>> tree3.draw()
```



7.3.4 Treebanks (notes)

The `nltk_lite.corpora` module defines the `treebank` corpus reader, which contains a 10% sample of the Penn Treebank corpus.

```
>>> from nltk_lite.corpora import treebank, extract
>>> print extract(0, treebank.parsed())
(S:
  (NP-SBJ:
    (NP: (NNP: 'Pierre') (NNP: 'Vinken'))
    (,: ',')
    (ADJP: (NP: (CD: '61') (NNS: 'years')) (JJ: 'old'))
    (,: ','))
  (VP:
    (MD: 'will')
    (VP:
      (VB: 'join')
      (NP: (DT: 'the') (NN: 'board'))
      (PP-CLR:
        (IN: 'as')
        (NP: (DT: 'a') (JJ: 'nonexecutive') (NN: 'director')))
      (NP-TMP: (NNP: 'Nov.') (CD: '29'))))
  (.: '.'))
```

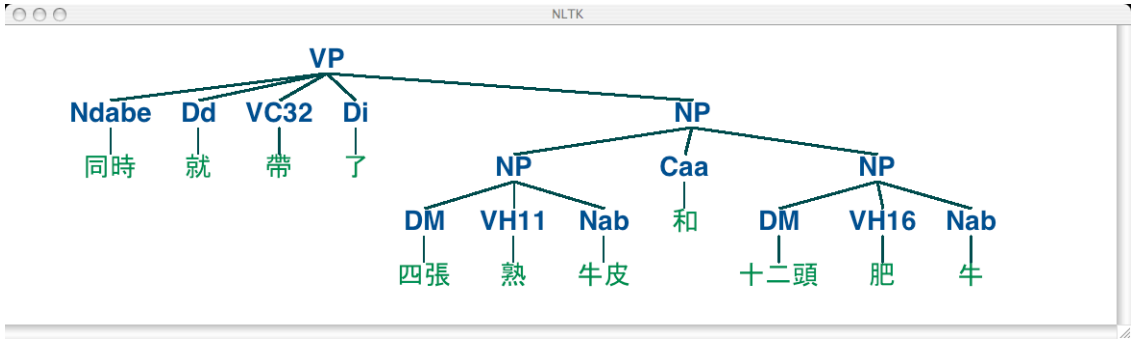
NLTK also includes a sample from the *Sinica Treebank Corpus*, consisting of 10,000 parsed sentences drawn from the *Academia Sinica Balanced Corpus of Modern Chinese*. Here is a code fragment to read and display one of the trees in this corpus.

Listing 29

```
def indent_tree(t, level=0, first=False, width=8):
    if not first:
        print ' '*(width+1)*level,
    try:
        print "%-*s" % (width, t.node),
        indent_tree(t[0], level+1, first=True)
        for child in t[1:]:
            indent_tree(child, level+1, first=False)
    except AttributeError:
        print t

>>> t = extract(0, treebank.parsed())
>>> indent_tree(t)
 S        NP-SBJ   NP       NNP      Pierre
                            NNP      Vinken
                   ,        ,
                   ADJP     NP       CD       61
                                     NNS      years
                            JJ       old
                   ,        ,
          VP       MD       will
                   VP       VB       join
                            NP       DT       the
                                     NN       board
                            PP-CLR   IN       as
                                     NP       DT       a
                                              JJ       nonexecutive
                                              NN       director
                            NP-TMP   NNP      Nov.
                                     CD       29
          .        .
```

```
>>> from nltk_lite.corpora import sinica_treebank, extract
>>> extract(3450, sinica_treebank.parsed()).draw()
```

(38)



Note that we can read tagged text from a Treebank corpus, using the `tagged()` method:

```
>>> print extract(0, treebank.tagged())
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'),
('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'),
('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'),
('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

7.3.5 Exercises

1. ☼ Can you come up with grammatical sentences which have probably never been uttered before? (Take turns with a partner.) What does this tell you about human language?

2. ☼ Recall Strunk and White's prohibition against sentence-initial *however* used to mean "although". Do a web search for *however* used at the start of the sentence. How widely used is this construction?

3. ☼ Consider the sentence *Kim arrived or Dana left and everyone cheered.* Write down the parenthesized forms to show the relative scope of *and* and *or*. Generate tree structures corresponding to both of these interpretations.

4. ☼ The `Tree` class implements a variety of other useful methods. See the `Tree` help documentation for more details, i.e. import the Tree class and then type `help(Tree)`.

5. ☼ **Building trees:**

   a) Write code to produce two trees, one for each reading of the phrase *old men and women*

   b) Encode any of the trees presented in this chapter as a labeled bracketing and use the `nltk_lite.parse` module's `bracket_parse()` method to check that it is well-formed. Now use the `draw()` to display the tree.

   c) As in (a) above, draw a tree for *The woman saw a man last Thursday*.

6. ☼ Write a recursive function to traverse a tree and return the depth of the tree, such that a tree with a single node would have depth zero. (Hint: the depth of a subtree is the maximum depth of its children, plus one.)

7. ☼ Analyze the A.A. Milne sentence about Piglet, by underlining all of the sentences it contains then replacing these with S (e.g. the first sentence becomes S *when*:lx‘ S). Draw a tree structure for this “compressed” sentence. What are the main syntactic constructions used for building such a long sentence?

8. ◑ To compare multiple trees in a single window, we can use the `draw_trees()` method. Define some trees and try it out:

```
>>> from nltk_lite.draw.tree import draw_trees
>>> draw_trees(tree1, tree2, tree3)
```

9. ◑ Using tree positions, list the subjects of the first 100 sentences in the Penn treebank; to make the results easier to view, limit the extracted subjects to subtrees whose height is 2.

10. ◑ Inspect the Prepositional Phrase Attachment Corpus and try to suggest some factors that influence PP attachment.

11. ◑ In this section we claimed that there are linguistic regularities which cannot be described simply in terms of n-grams. Consider the following sentence, particularly the position of the phrase *in his turn*. Does this illustrate a problem for an approach based on n-grams?

> *What was more, the in his turn somewhat youngish Nikolay Parfenovich also turned out to be the only person in the entire world to acquire a sincere liking to our “discriminated-against” public procurator.* (Dostoevsky: The Brothers Karamazov)

12. ◑ Write a recursive function that produces a nested bracketing for a tree, leaving out the leaf nodes, and displaying the non-terminal labels after their subtrees. So the above example about Pierre Vinken would produce: `[[[NNP NNP]NP , [ADJP [CD NNS]NP JJ]ADJP ,]NP-SBJ MD [VB [DT NN]NP [IN [DT JJ NN]NP]PP-CLR [NNP CD]NP-TMP]VP .]S` Consecutive categories should be separated by space.

1. ◑ Download several electronic books from Project Gutenberg. Write a program to scan these texts for any extremely long sentences. What is the longest sentence you can find? What syntactic construction(s) are responsible for such long sentences?

2. ★ One common way of defining the subject of a sentence S in English is as *the noun phrase that is the daughter of* S *and the sister of* VP. Write a function that takes the tree for a sentence and returns the subtree corresponding to the subject of the sentence. What should it do if the root node of the tree passed to this function is not S, or it lacks a subject?

7.4 Context Free Grammar

As we have seen, languages are infinite — there is no principled upper-bound on the length of a sentence. Nevertheless, we would like to write (finite) programs that can process well-formed sentences. It turns out that we can characterize what we mean by well-formedness using a grammar. The way that finite grammars are able to describe an infinite set uses **recursion**. (We already came across this idea when we looked at regular expressions: the finite expression `a+` is able to describe the infinite set `{a , aa, aaa, aaaa, ...}`). Apart from their compactness, grammars usually capture important

structural and distributional properties of the language, and can be used to map between sequences of words and abstract representations of meaning. Even if we were to impose an upper bound on sentence length to ensure the language was finite, we would probably still want to come up with a compact representation in the form of a grammar.

A **grammar** is a formal system which specifies which sequences of words are well-formed in the language, and which provides one or more phrase structures for well-formed sequences. We will be looking at **context-free grammar** (CFG), which is a collection of **productions** of the form S → NP VP. This says that a constituent S can consist of sub-constituents NP and VP. Similarly, the production V → `'saw' | ``'walked'` means that the constituent V can consist of the string *saw* or *walked*. For a phrase structure tree to be well-formed relative to a grammar, each non-terminal node and its children must correspond to a production in the grammar.

7.4.1 A Simple Grammar

Let's start off by looking at a simple context-free grammar. By convention, the left-hand-side of the first production is the **start-symbol** of the grammar, and all well-formed trees must have this symbol as their root label.

(39) S → NP VP
NP → Det N | Det N PP
VP → V | V NP | V NP PP
PP → P NP

Det → 'the' | 'a'
N → 'man' | 'park' | 'dog' | 'telescope'
V → 'saw' | 'walked'
P → 'in' | 'with'

This grammar contains productions involving various syntactic categories, as laid out in Table 7.1.

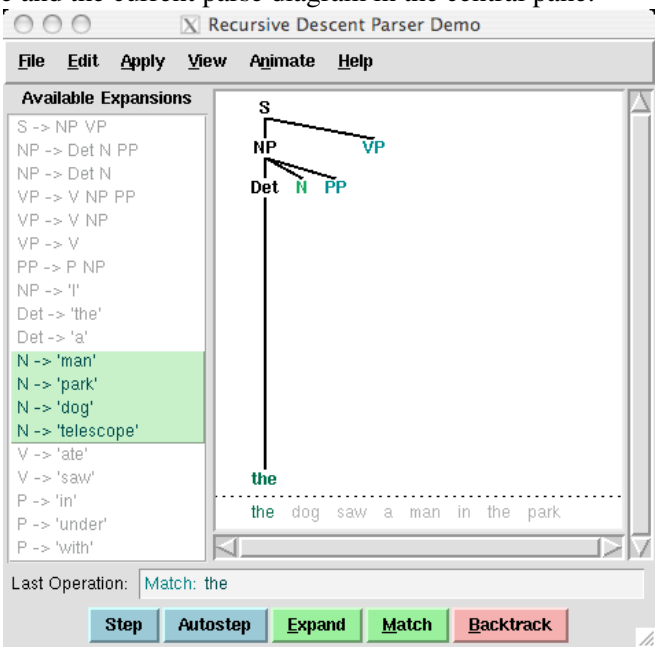
Symbol	Meaning	Example
S	sentence	*the man walked*
NP	noun phrase	*a dog*
VP	verb phrase	*saw a park*
PP	prepositional phrase	*with a telescope*
...	...	...
Det	determiner	*the*
N	noun	*dog*
V	verb	*walked*
P	preposition	*in*

Table 7.1: Syntactic Categories

In our following discussion of grammar, we will use the following terminology. The grammar consists of productions, where each production involves a single **non-terminal** (e.g. S, NP), an arrow, and one or more non-terminals and **terminals** (e.g. *walked*). The productions are often divided into two

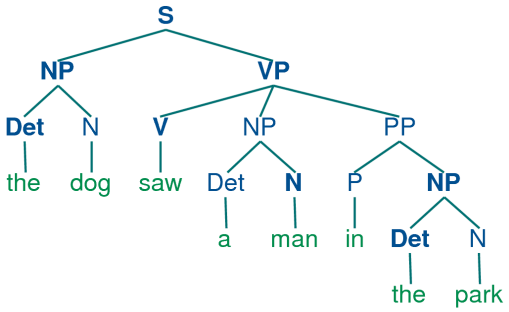
main groups. The **grammatical productions** are those without a terminal on the right-hand side. The **lexical productions** are those having a terminal on the right-hand side. A special case of non-terminals are the **pre-terminals**, which appear on the left-hand side of lexical productions. We will say that a grammar **licenses** a tree if each non-terminal X with children $Y_1 \quad Y_n$ corresponds to a production in the grammar of the form: $X \rightarrow Y_1 \quad Y_n$.

In order to get started with developing simple grammars of your own, you will probably find it convenient to play with the recursive descent parser demo, `from nltk_lite.draw.rdparser .demo().` The demo opens a window which displays a list of grammar productions in the lefthand pane and the current parse diagram in the central pane:

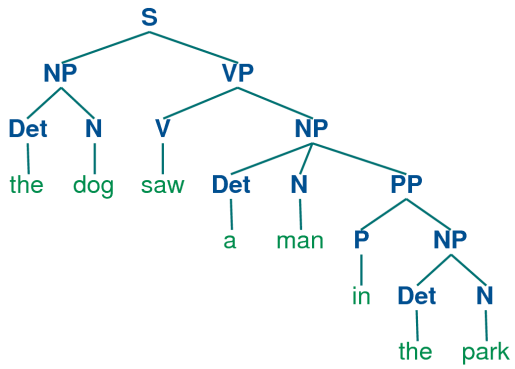


The demo comes with the grammar in (39) already loaded. We will discuss the parsing algorithm in greater detail below, but for the time being you can get an idea of how it works by using the *autostep* button. If we parse the string *The dog saw a man in the park* using the grammar in (39), we end up with two trees:

(40a)



(40b)



Since our grammar licenses two trees for this sentence, the sentence is said to be **structurally ambiguous**. The ambiguity in question is called a *prepositional phrase attachment ambiguity*, as we saw earlier in this chapter. As you may recall, it is an ambiguity about attachment since the PP *in the park* needs to be attached to one of two places in the tree: either as a daughter of VP or else as a daughter of NP. When the PP is attached to VP, the seeing event happened in the park. However, if the PP is attached to NP, then the man was in the park, and the agent of the seeing (the dog) might have been sitting on the balcony of an apartment overlooking the park. As we will see, dealing with ambiguity is a key challenge in parsing.

7.4.2 Recursion in syntactic structure

Observe that sentences can be nested within sentences, with no limit to the depth:

(41a) Jodie won the 100m freestyle

(41b) "The Age" reported that Jodie won the 100m freestyle

(41c) Sandy said "The Age" reported that Jodie won the 100m freestyle

(41d) I think Sandy said "The Age" reported that Jodie won the 100m freestyle

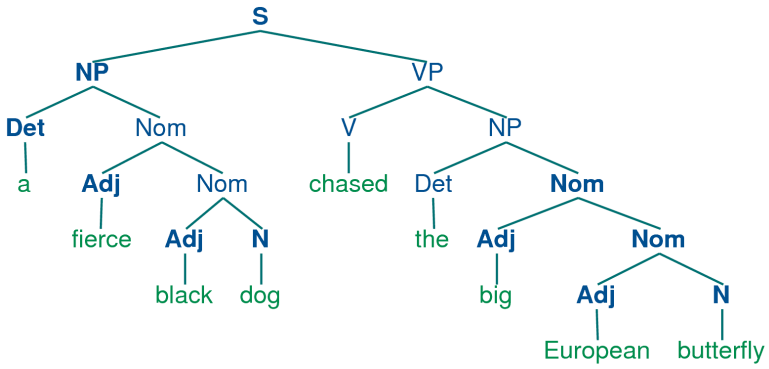
This nesting is explained in terms of **recursion**. A grammar is said to be **recursive** if a category occurring on the lefthand side of a production (such as S in this case) also appears on the righthand side of a production. If this dual occurrence takes place in *one and the same production*, then we have **direct recursion**; otherwise we have **indirect recursion**. There is no recursion in (39). However, the grammar in (42) illustrates both kinds of recursive production:

(42)
```
S  → NP VP
NP → Det Nom | Det Nom PP | PropN
Nom → Adj Nom | N
VP → V | V NP | V NP PP | V S
PP → P NP

PropN → 'John' | 'Mary'
Det → 'the' | 'a'
N → 'man' | 'woman' | 'park' | 'dog' | 'lead' | 'telescope' | 'butterfly'
Adj  → 'fierce' | 'black' |  'big' | 'European'
V → 'saw' | 'chased' | 'barked'  | 'disappeared' | 'said' | 'reported'
P → 'in' | 'with'
```

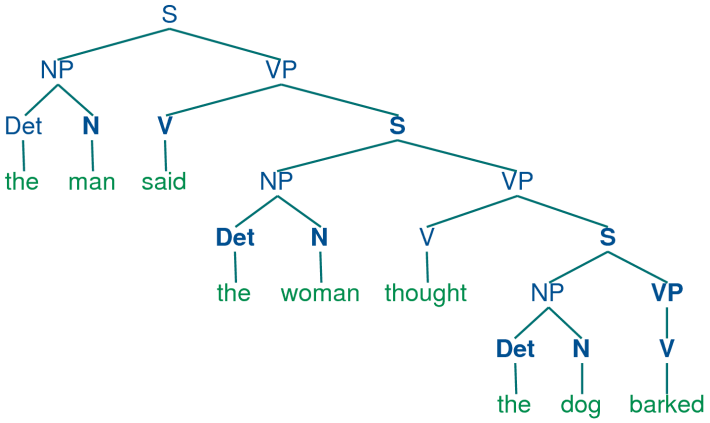
Notice that the production NOM → ADJ NOM (where NOM is the category of nominals) involves direct recursion on the category NOM, whereas indirect recursion on S arises from the combination of two productions, namely S → NP VP and VP → V S.

To see how recursion is handled in this grammar, consider the following trees. Example nested-nominals involves nested nominal phrases, while nested-sentences contains nested sentences.

(43a)

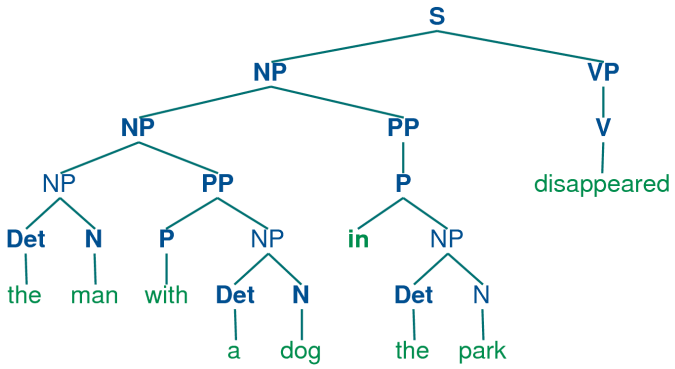


(43b)



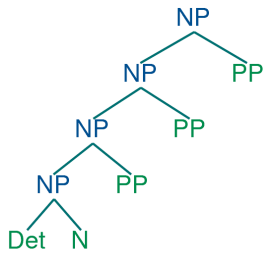
If you did the exercises for the last section, you will have noticed that the recursive descent parser fails to deal properly with the following production: NP → NP PP. From a linguistic point of view, this production is perfectly respectable, and will allow us to derive trees like this:

(44)



More schematically, the trees for these compound noun phrases will be of the following shape:

(45)



The structure in (45) is called a **left recursive** structure. These occur frequently in analyses of English, and the failure of recursive descent parsers to deal adequately with left recursion means that we will need to find alternative approaches.

7.4.3 Heads, Complements and Modifiers

Let us take a closer look at verbs. The grammar (42) correctly generates examples like (46), corresponding to the four productions with VP on the lefthand side:

(46a) The woman gave the telescope to the dog

(46b) The woman saw a man

(46c) A man said that the woman disappeared

(46d) The dog barked

That is, *gave* can occur with a following NP and PP; *saw* can occur with a following NP; *said* can occur with a following S; and *barked* can occur with no following phrase. In these cases, NP, PP and S are called **complements** of the respective verbs, and the verbs themselves are called **heads** of the verb phrase.

However, there are fairly strong constraints on what verbs can occur with what complements. Thus, we would like our grammars to mark the following examples as ungrammatical[1]:

(47a) *The woman disappeared the telescope to the dog

(47b) *The dog barked a man

(47c) *A man gave that the woman disappeared

(47d) *A man said

How can we ensure that our grammar correctly excludes the ungrammatical examples in (47)? We need some way of constraining grammar productions which expand VP so that verbs *only* cooccur with their correct complements. We do this by dividing the class of verbs into **subcategories**, each of which is associated with a different set of complements. For example, **transitive verbs** such as *saw*, *kissed* and *hit* require a following NP object complement. Borrowing from the terminology of chemistry, we sometimes refer to the **valency** of a verb, that is, its capacity to combine with a sequence of arguments and thereby compose a verb phrase.

Let's introduce a new category label for such verbs, namely TV (for Transitive Verb), and use it in the following productions:

[1]It should be borne in mind that it is possible to create examples which involve 'non-standard' but interpretable combinations of verbs and complements. Thus, we can, at a stretch, interpret *the man disappeared the dog* as meaning that the man made the dog disappear. We will ignore such examples here.

(48)
```
VP → TV NP
TV → 'saw' | 'kissed' | 'hit'
```

Now **the dog barked the man* is excluded since we haven't listed *barked* as a V_TR, but *the woman saw a man* is still allowed. Table 7.2 provides more examples of labels for verb subcategories.

Symbol	Meaning	Example
IV	intransitive verb	*barked*
TV	transitive verb	*saw a man*
DatV	dative verb	*gave a dog to a man*
SV	sentential verb	*said that a dog barked*

Table 7.2: Verb Subcategories

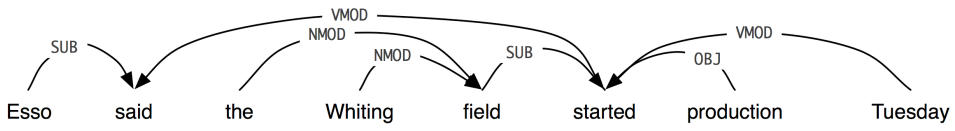
The revised grammar for VP will now look like this:

(49)
```
VP → DATV NP PP
VP → TV NP
VP → SV S
VP → IV

DATV → 'gave' | 'donated' | 'presented'
TV → 'saw' | 'kissed' | 'hit' | 'sang'
SV → 'said' | 'knew' | 'alleged'
IV → 'barked' | 'disappeared' | 'elapsed' | 'sang'
```

Notice that according to (49), a given lexical item can belong to more than one subcategory. For example, *sang* can occur both with and without a following NP complement.

7.4.4 Dependency Grammar

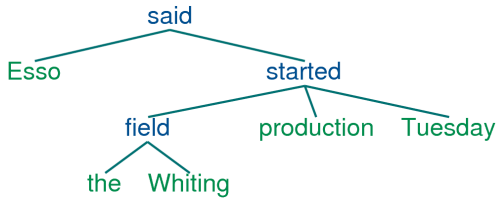
Although we concentrate on phrase structure grammars in this chapter, we should mention an alternative approach, namely **dependency grammar**. Rather than taking starting from the grouping of words into constituents, dependency grammar takes as basic the notion that one word can be dependent on another (namely, its head). The head of a sentence is usually taken to be the main verb, and every other word is either dependent on this head, or connects to it through a path of dependencies. Figure (50) illustrates a dependency graph, where the head of the arrow points to the head of a dependency.

(50)


As you will see, the arcs in Figure (50) are labeled with the particular dependency relation that holds between a dependent and its head. For example, *Esso* bears the subject relation to *said* (which is the head of the whole sentence), and *Tuesday* bears a verbal modifier (VMOD) relation to *started*.

An alternative way of representing the dependency relationships is illustrated in the tree (51), where dependents are shown as daughters of their heads.

(51)



One format for encoding dependency information places each word on a line, followed by its part-of-speech tag, the index of its head, and the label of the dependency relation (cf. [Nivre et al., 2006]). The index of a word is implicitly given by the ordering of the lines (with 1 as the first index). This is illustrated in the following code snippet:

```
>>> from nltk_lite.contrib.dependency import DepGraph
>>> dg = DepGraph().read("""Esso     NNP     2        SUB
... said     VBD     0        ROOT
... the      DT      5        NMOD
... Whiting  NNP     5        NMOD
... field    NN      6        SUB
... started  VBD     2        VMOD
... production       NN      6        OBJ
... Tuesday  NNP     6        VMOD""")
```

As you will see, this format also adopts the convention that the head of the sentence is dependent on an empty node, indexed as 0. We can use the `deptree()` method of a `DepGraph()` object to build an NLTK tree like that illustrated earlier in (51).

```
>>> tree = dg.deptree()
>>> tree.draw()
```

7.4.5 Formalizing Context Free Grammars

We have seen that a CFG contains terminal and nonterminal symbols, and productions which dictate how constituents are expanded into other constituents and words. In this section, we provide some formal definitions.

A CFG is a 4-tuple N, Σ, P, S, where:

- Σ is a set of *terminal* symbols (e.g., lexical items);
- N is a set of *non-terminal* symbols (the category labels);
- P is a set of *productions* of the form $A \rightarrow \alpha$, where
  - A is a non-terminal, and
  - α is a string of symbols from $(N \ \Sigma)^*$ (i.e., strings of either terminals or non-terminals);
- S is the *start symbol*.

A **derivation** of a string from a non-terminal A in grammar G is the result of successively applying productions from G to A. For example, (52) is a derivation of *the dog with a telescope* for the grammar in (39).

(52)
```
NP
Det N PP
the N PP
the dog PP
the dog P NP
the dog with NP
the dog with Det N
the dog with a N
the dog with a telescope
```

Although we have chosen here to expand the leftmost non-terminal symbol at each stage, this is not obligatory; productions can be applied in any order. Thus, derivation (52) could equally have started off in the following manner:

(53)
```
NP
Det N PP
Det N P NP
Det N with NP
...
```

We can also write derivation (52) as:

(54) NP ⇒ DET N PP ⇒ *the* N PP ⇒ *the dog* PP ⇒ *the dog* P NP ⇒ *the dog with* NP ⇒ *the dog with a* N ⇒ *the dog with a telescope*

where ⇒ means "derives in one step". We use ⇒* to mean "derives in zero or more steps":

- α ⇒* α for any string α, and
- if α ⇒* β and β ⇒ γ, then α ⇒* γ.

We write A ⇒* α to indicate that α can be derived from A.

In NLTK, context free grammars are defined in the `parse.cfg` module. The easiest way to construct a grammar object is from the standard string representation of grammars. In Listing 7.2 we define a grammar and use it to parse a simple sentence. You will learn more about parsing in the next section.

7.4.6 Exercises

1. ☼ In the recursive descent parser demo, experiment with changing the sentence to be parsed by selecting *Edit Text* in the *Edit* menu.
2. ☼ Can the grammar in (39) be used to describe sentences which are more than 20 words in length?
3. ◑ You can modify the grammar in the recursive descent parser demo by selecting *Edit Grammar* in the *Edit* menu. Change the first expansion production, namely `NP -> Det N PP`, to `NP -> NP PP`. Using the *Step* button, try to build a parse tree. What happens?
4. ◑ Extend the grammar in (42) with productions which expand prepositions as intransitive, transitive and requiring a PP complement. Based on these productions, use the method of the preceding exercise to draw a tree for the sentence *Lee ran away home*.

Listing 30 Context Free Grammars in NLTK

```
from nltk_lite import parse
grammar = parse.cfg.parse_grammar("""
  S -> NP VP
  VP -> V NP | V NP PP
  V -> "saw" | "ate"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "dog" | "cat" | "cookie" | "park"
  PP -> P NP
  P -> "in" | "on" | "by" | "with"
  """)

>>> from nltk_lite import tokenize
>>> sent = list(tokenize.whitespace("Mary saw Bob"))
>>> rd_parser = parse.RecursiveDescent(grammar)
>>> for p in rd_parser.get_parse_list(sent):
...     print p
(S: (NP: 'Mary') (VP: (V: 'saw') (NP: 'Bob')))
```

5. ◑ Pick some common verbs and complete the following tasks:

   a) Write a program to find those verbs in the PP Attachment Corpus included with NLTK. Find any cases where the same verb exhibits two different attachments, but where the first noun, or second noun, or preposition, stay unchanged (as we saw in the PP Attachment Corpus example data above).

   b) Devise CFG grammar productions to cover some of these cases.

6. ★ **Lexical Acquisition:** As we saw in Chapter 5, it is possible to collapse chunks down to their chunk label. When we do this for sentences involving the word *gave*, we find patterns such as the following:

```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

   a) Use this method to study the complementation patterns of a verb of interest, and write suitable grammar productions.

   b) Identify some English verbs that are near-synonyms, such as the *dumped/filled/loaded* example from earlier in this chapter. Use the chunking method to study the complementation patterns of these verbs. Create a grammar to cover these cases. Can the verbs be freely substituted for each other, or are their constraints? Discuss your findings.

7.5 Parsing

A **parser** processes input sentences according to the productions of a grammar, and builds one or more constituent structures which conform to the grammar. A grammar is a declarative specification of well-formedness. In NLTK, it is just a multi-line string; it is not itself a program that can be used for anything. A parser is a procedural interpretation of the grammar. It searches through the space of trees licensed by a grammar to find one that has the required sentence along its fringe.

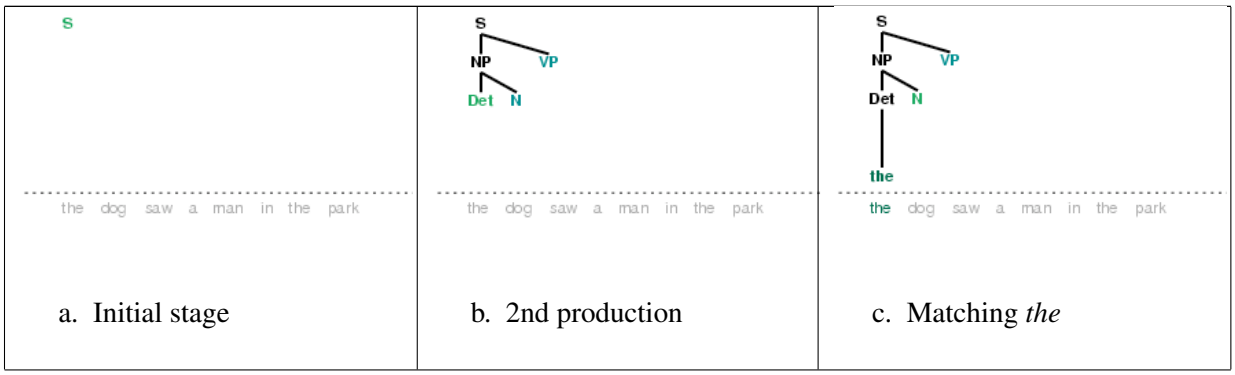
Parsing is important in both linguistics and natural language processing. A parser permits a grammar to be evaluated against a potentially large collection of test sentences, helping linguists to find any problems in their grammatical analysis. A parser can serve as a model of psycholinguistic processing, helping to explain the difficulties that humans have with processing certain syntactic constructions. Many natural language applications involve parsing at some point; for example, we would expect the natural language questions submitted to a question-answering system to undergo parsing as an initial step.

In this section we see two simple parsing algorithms, a top-down method called recursive descent parsing, and a bottom-up method called shift-reduce parsing.

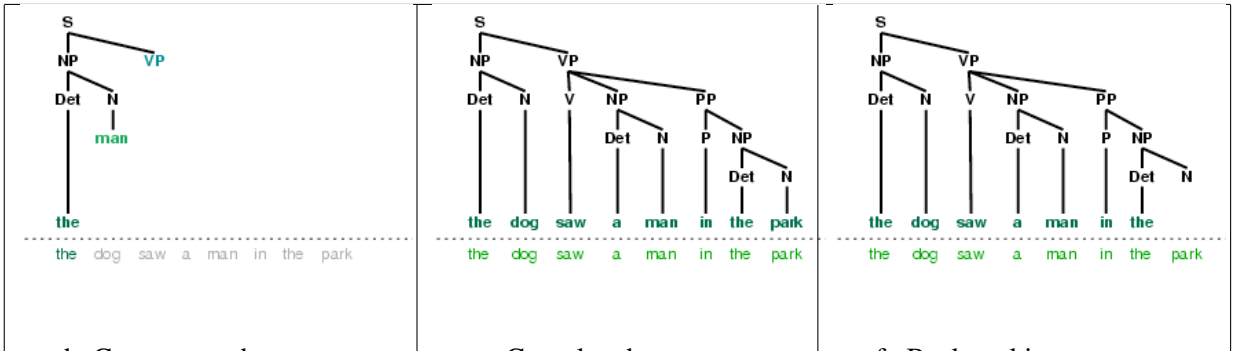
7.5.1 Recursive Descent Parsing

The simplest kind of parser interprets a grammar as a specification of how to break a high-level goal into several lower-level subgoals. The top-level goal is to find an S. The S → NP VP production permits the parser to replace this goal with two subgoals: find an NP, then find a VP. Each of these subgoals can be replaced in turn by sub-sub-goals, using productions that have NP and VP on their left-hand side. Eventually, this expansion process leads to subgoals such as: find the word *telescope*. Such subgoals can be directly compared against the input string, and succeed if the next word is matched. If there is no match the parser must back up and try a different alternative.

The recursive descent parser builds a parse tree during the above process. With the initial goal (find an S), the S root node is created. As the above process recursively expands its goals using the productions of the grammar, the parse tree is extended downwards (hence the name *recursive descent*). We can see this in action using the parser demonstration `nltk_lite.draw.rdparser.demo()`. Six stages of the execution of this parser are shown in Table 7.3.



a. Initial stage b. 2nd production c. Matching *the*



d. Cannot match *man* e. Completed parse f. Backtracking

Table 7.3: Six Stages of a Recursive Descent Parser

During this process, the parser is often forced to choose between several possible productions. For example, in going from step c to step d, it tries to find productions with N on the left-hand side. The first of these is N → *man*. When this does not work it *backtracks*, and tries other N productions in order, under it gets to N → *dog*, which matches the next word in the input sentence. Much later, as shown in step e, it finds a complete parse. This is a tree which covers the entire sentence, without any dangling edges. Once a parse has been found, we can get the parser to look for additional parses. Again it will backtrack and explore other choices of production in case any of them result in a parse.

NLTK provides a recursive descent parser:

```
>>> from nltk_lite import parse
>>> rd_parser = parse.RecursiveDescent(grammar)
>>> sent = list(tokenize.whitespace('Mary saw a dog'))
>>> rd_parser.get_parse_list(sent)
[('S': ('NP': 'Mary') ('VP': ('V': 'saw') ('NP': ('Det': 'a') ('N': 'dog'))))]
```

Note

`parse.RecursiveDescent()` takes an optional parameter `trace`. If `trace` is greater than zero, then the parser will report the steps that it takes as it parses a text.

Recursive descent parsing has three key shortcomings. First, left-recursive productions like NP → NP PP send it into an infinite loop. Second, the parser wastes a lot of time considering words and structures that do not correspond to the input sentence. Third, the backtracking process may discard parsed constituents that will need to be rebuilt again later. For example, backtracking over VP → V NP will discard the subtree created for the NP. If the parser then proceeds with VP → V NP PP, then the NP subtree must be created all over again.

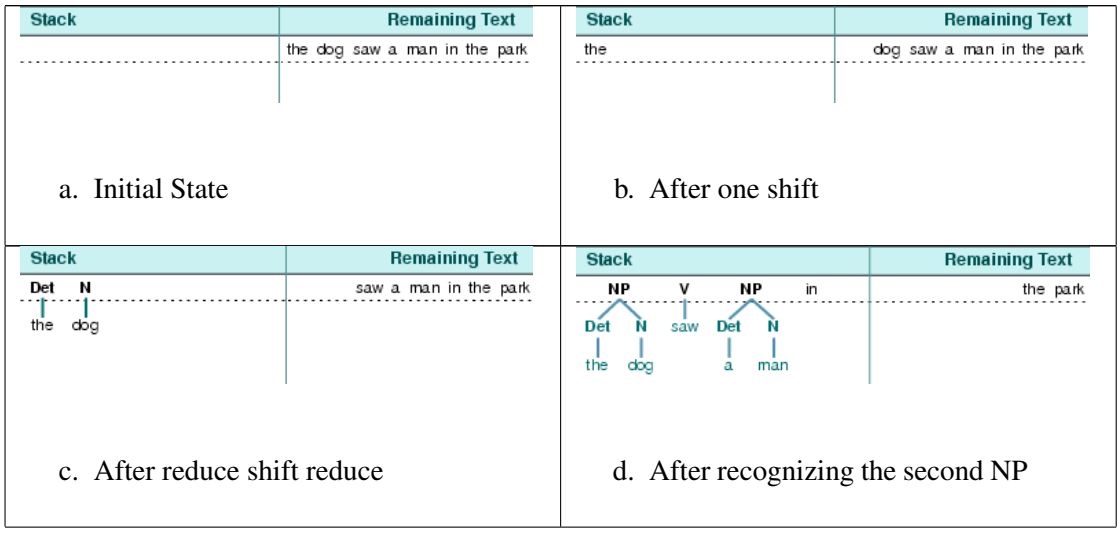
Recursive descent parsing is a kind of **top-down parsing**. Top-down parsers use a grammar to *predict* what the input will be, before inspecting the input! However, since the input is available to the parser all along, it would be more sensible to consider the input sentence from the very beginning. This approach is called **bottom-up parsing**, and we will see an example in the next section.

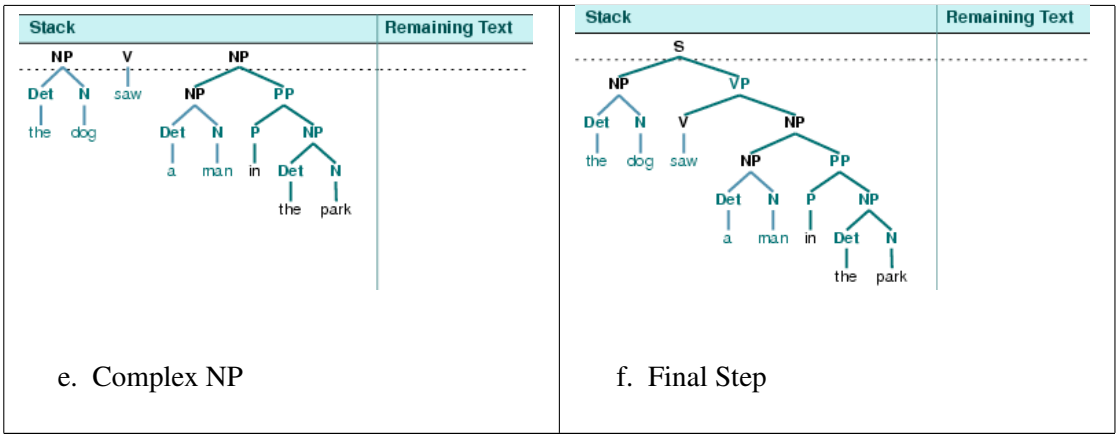
7.5.2 Shift-Reduce Parsing

A simple kind of bottom-up parser is the **shift-reduce parser**. In common with all bottom-up parsers, a shift-reduce parser tries to find sequences of words and phrases that correspond to the *right-hand* side of a grammar production, and replace them with the left-hand side, until the whole sentence is reduced to an S.

The shift-reduce parser repeatedly pushes the next input word onto a stack (Section 6.2.4); this is the **shift** operation. If the top *n* items on the stack match the *n* items on the right-hand side of some production, then they are all popped off the stack, and the item on the left-hand side of the production is pushed on the stack. This replacement of the top *n* items with a single item is the **reduce** operation. (This reduce operation may only be applied to the top of the stack; reducing items lower in the stack must be done before later items are pushed onto the stack.) The parser finishes when all the input is consumed and there is only one item remaining on the stack, a parse tree with an S node as its root.

The shift-reduce parser builds a parse tree during the above process. If the top of stack holds the word *dog*, and if the grammar has a production N → *dog*, then the reduce operation causes the word to be replaced with the parse tree for this production. For convenience we will represent this tree as `N(dog)`. At a later stage, if the top of the stack holds two items `Det(the) N(dog)` and if the grammar has a production NP → DET N then the reduce operation causes these two items to be replaced with `NP(Det(the), N(dog))`. This process continues until a parse tree for the entire sentence has been constructed. We can see this in action using the parser demonstration `nltk_lite.draw.srparser.demo()`. Six stages of the execution of this parser are shown in Figure 7.4.





e. Complex NP f. Final Step

Table 7.4: Six Stages of a Shift-Reduce Parser

NLTK provides `parse.ShiftReduce()`, a simple implementation of a shift-reduce parser. This parser does not implement any backtracking, so it is not guaranteed to find a parse for a text, even if one exists. Furthermore, it will only find at most one parse, even if more parses exist. We can provide an optional `trace` parameter, which controls how verbosely the parser reports the steps that it takes as it parses a text:

```
>>> sr_parse = parse.ShiftReduce(grammar, trace=2)
>>> sent = list(tokenize.whitespace('Mary saw a dog'))
>>> sr_parse.parse(sent)
Parsing 'Mary saw a dog'
    [ * Mary saw a dog]
  S [ 'Mary' * saw a dog]
  R [ <NP> * saw a dog]
  S [ <NP> 'saw' * a dog]
  R [ <NP> <V> * a dog]
  S [ <NP> <V> 'a' * dog]
  R [ <NP> <V> <Det> * dog]
  S [ <NP> <V> <Det> 'dog' * ]
  R [ <NP> <V> <Det> <N> * ]
  R [ <NP> <V> <NP> * ]
  R [ <NP> <VP> * ]
  R [ <S> * ]
  ('S': ('NP': 'Mary') ('VP': ('V': 'saw') ('NP': ('Det': 'a') ('N': 'dog'))))
```

Shift-reduce parsers have a number of problems. A shift-reduce parser may fail to parse the sentence, even though the sentence is well-formed according to the grammar. In such cases, there are no remaining input words to shift, and there is no way to reduce the remaining items on the stack, as exemplified in Table 7.5a. The parser entered this blind alley at an earlier stage shown in Table 7.5b, when it reduced instead of shifted. This situation is called a **shift-reduce conflict**. At another possible stage of processing shown in Table 7.5c, the parser must choose between two possible reductions, both matching the top items on the stack: VP → VP NP PP or NP → NP PP. This situation is called a **reduce-reduce conflict**.

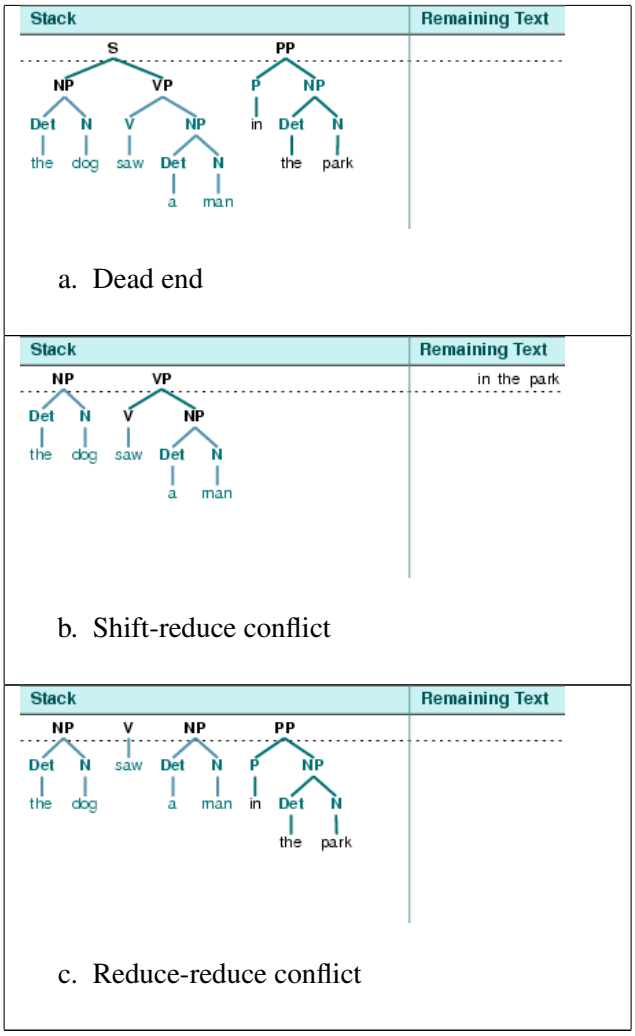


Table 7.5: Conflict in Shift-Reduce Parsing

Shift-reduce parsers may implement policies for resolving such conflicts. For example, they may address shift-reduce conflicts by shifting only when no reductions are possible, and they may address reduce-reduce conflicts by favouring the reduction operation that removes the most items from the stack. No such policies are failsafe however.

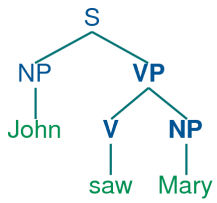
The advantages of shift-reduce parsers over recursive descent parsers is that they only build structure that corresponds to the words in the input. Furthermore, they only build each sub-structure once, e.g. `NP(Det(the), N(man))` is only built and pushed onto the stack a single time, regardless of whether it will later be used by the VP → V NP PP reduction or the NP → NP PP reduction.

7.5.3 The Left-Corner Parser

One of the problems with the recursive descent parser is that it can get into an infinite loop. This is because it applies the grammar productions blindly, without considering the actual input sentence. A left-corner parser is a hybrid between the bottom-up and top-down approaches we have seen.

Grammar (42) allows us to produce the following parse of *John saw Mary*:

(55)



Recall that the grammar in (42) has the following productions for expanding NP:

(56a) NP → DT NOM

(56b) NP → DT NOM PP

(56c) NP → PROPN

Suppose we ask you to first look at tree (55), and then decide which of the NP productions you'd want a recursive descent parser to apply first — obviously, (56c) is the right choice! How do you know that it would be pointless to apply (56a) or (56b) instead? Because neither of these productions will derive a string whose first word is *John*. That is, we can easily tell that in a successful parse of *John saw Mary*, the parser has to expand NP in such a way that NP derives the string *John* α. More generally, we say that a category *B* is a **left-corner** of a tree rooted in *A* if $A \Rightarrow^* B\ \alpha$.

(57)

A **left-corner parser** is a top-down parser with bottom-up filtering. Unlike an ordinary recursive descent parser, it does not get trapped in left recursive productions. Before starting its work, a left-corner parser preprocesses the context-free grammar to build a table where each row contains two cells, the first holding a non-terminal, and the second holding the collection of possible left corners of that non-terminal. Table 7.6 illustrates this for the grammar from (42).

Category	Left-Corners (pre-terminals)
S	NP
NP	Det, PropN
VP	V
PP	P

Table 7.6: Left-Corners in (42)

Each time a production is considered by the parser, it checks that the next input word is compatible with at least one of the pre-terminal categories in the left-corner table.

[TODO: *explain how this effects the action of the parser, and why this solves the problem.*]

7.5.4 Exercises

1. ☼ With pen and paper, manually trace the execution of a recursive descent parser and a shift-reduce parser, for a CFG you have already seen, or one of your own devising.

2. ◑ Compare the performance of the top-down, bottom-up, and left-corner parsers using the same grammar and three grammatical test sentences. Use `timeit` to log the amount of time each parser takes on the same sentence (Section 6.5.4). Write a function which runs all three parsers on all three sentences, and prints a 3-by-3 grid of times, as well as row and column totals. Discuss your findings.

3. ◑ Read up on "garden path" sentences. How might the computational work of a parser relate to the difficulty humans have with processing these sentences? `http://en.wikipedia.org/wiki/Garden_path_sentence`

4. ★ **Left-corner parser:** Develop a left-corner parser based on the recursive descent parser, and inheriting from `ParseI`. (Note, this exercise requires knowledge of Python classes, covered in Chapter 10.)

5. ★ Extend NLTK's shift-reduce parser to incorporate backtracking, so that it is guaranteed to find all parses that exist (i.e. it is **complete**).

7.6 Conclusion

We began this chapter talking about confusing encounters with grammar at school. We just wrote what we wanted to say, and our work was handed back with red marks showing all our grammar mistakes. If this kind of "grammar" seems like secret knowledge, the linguistic approach we have taken in this chapter is quite the opposite: grammatical structures are made explicit as we build trees on top of sentences. We can write down the grammar productions, and parsers can build the trees automatically. This thoroughly objective approach is widely referred to as **generative grammar**.

Note that we have only considered "toy grammars," small grammars that illustrate the key aspects of parsing. But there is an obvious question as to whether the general approach can be scaled up to cover large corpora of natural languages. How hard would it be to construct such a set of productions by hand? In general, the answer is: *very hard*. Even if we allow ourselves to use various formal devices that give much more succinct representations of grammar productions (some of which will be discussed in Chapter 8), it is still extremely difficult to keep control of the complex interactions between the many productions required to cover the major constructions of a language. In other words, it is hard to modularize grammars so that one portion can be developed independently of the other parts. This in turn means that it is difficult to distribute the task of grammar writing across a team of linguists. Another difficulty is that as the grammar expands to cover a wider and wider range of constructions, there is a corresponding increase in the number of analyses which are admitted for any one sentence. In other words, ambiguity increases with coverage.

Despite these problems, there are a number of large collaborative projects which have achieved interesting and impressive results in developing rule-based grammars for several languages. Examples are the Lexical Functional Grammar (LFG) Pargram project (http://www2.parc.com/istl/groups/nltt/pargram/), the Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework (http://www.delph-in.net/matrix/), and the Lexicalized Tree Adjoining Grammar XTAG Project (http://www.cis.upenn.edu/~xtag/).

7.7 Summary (notes)

- Sentences have internal organization, or constituent structure, which can be represented using a tree; notable features of constituent structure are: recursion, heads, complements, modifiers

- A grammar is a compact characterization of a potentially infinite set of sentences; we say that a tree is well-formed according to a grammar, or that a grammar licenses a tree.
- Syntactic ambiguity arises when one sentence has more than one syntactic structure (e.g. prepositional phrase attachment ambiguity).
- A parser is a procedure for finding one or more trees corresponding to a grammatically well-formed sentence.
- A simple top-down parser is the recursive descent parser (summary, problems)
- A simple bottom-up parser is the shift-reduce parser (summary, problems)
- It is difficult to develop a broad-coverage grammar...

7.8 Further Reading

There are many introductory books on syntax. [O'Grady1989LI]_ is a general introduction to linguistics, while [Radford, 1988] provides a gentle introduction to transformational grammar, and can be recommended for its coverage of transformational approaches to unbounded dependency constructions.

[Burton-Roberts, 1997] is very practically oriented textbook on how to analyse constituency in English, with extensive exemplification and exercises. [Huddleston and Pullum, 2002] provides an up-to-date and comprehensive analysis of syntactic phenomena in English.

- LALR(1)
- Marcus parser
- Lexical Functional Grammar (LFG)
  - Pargram project
  - LFG Portal
- Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework
- Lexicalized Tree Adjoining Grammar XTAG Project

Chapter 8

Advanced Parsing

8.1 Introduction

Chapter 7 started with an introduction to constituent structure in English, showing how words in a sentence group together in predictable ways. We showed how to describe this structure using syntactic tree diagrams, and observed that it is sometimes desirable to assign more than one such tree to a given string. In this case, we said that the string was structurally ambiguous; and example was *old men and women*.

Treebanks are language resources in which the syntactic structure of a corpus of sentences has been annotated, usually by hand. However, we would also like to be able to produce trees algorithmically. A context-free phrase structure grammar (CFG) is a formal model for describing whether a given string can be assigned a particular constituent structure. Given a set of syntactic categories, the CFG uses a set of productions to say how a phrase of some category A can be analysed into a sequence of smaller parts α_1 ... α_n. But a grammar is a static description of a set of strings; it does not tell us what sequence of steps we need to take to build a constituent structure for a string. For this, we need to use a parsing algorithm. We presented two such algorithms: Top-Down Recursive Descent (7.5.1) and Bottom-Up Shift-Reduce (7.5.2). As we pointed out, both parsing approaches suffer from important shortcomings. The Recursive Descent parser cannot handle left-recursive productions (e.g., productions such as NP → NP PP), and blindly expands categories top-down without checking whether they are compatible with the input string. The Shift-Reduce parser is not guaranteed to find a valid parse for the input even if one exists, and builds substructure without checking whether it is globally consistent with the grammar. As we will describe further below, the Recursive Descent parser is also inefficient in its search for parses.

So, parsing builds trees over sentences, according to a phrase structure grammar. Now, all the examples we gave in Chapter 7 only involved toy grammars containing a handful of productions. What happens if we try to scale up this approach to deal with realistic corpora of language? Unfortunately, as the coverage of the grammar increases and the length of the input sentences grows, the number of parse trees grows rapidly. In fact, it grows at an astronomical rate.

Let's explore this issue with the help of a simple example. The word *fish* is both a noun and a verb. We can make up the sentence *fish fish fish*, meaning *fish like to fish for other fish*. (Try this with *police* if you prefer something more sensible.) Here is a toy grammar for the "fish" sentences.

```
>>> from nltk_lite.parse import cfg, chart
>>> grammar = cfg.parse_grammar("""
... S -> NP V NP
... NP -> NP Sbar
```

```
... Sbar -> NP V
... NP -> 'fish'
... V -> 'fish'
... """)
```

Now we can try parsing a longer sentence, *fish fish fish fish fish*, which amongst other things, means 'fish that other fish fish are in the habit of fishing fish themselves'. We use the NLTK chart parser, which is presented later on in this chapter. This sentence has two readings.

```
>>> tokens = ["fish"] * 5
>>> cp = chart.ChartParse(grammar, chart.TD_STRATEGY)
>>> for tree in cp.get_parse_list(tokens):
...     print tree
(S:
  (NP: 'fish')
  (V: 'fish')
  (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish'))))
(S:
  (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish')))
  (V: 'fish')
  (NP: 'fish'))
```

As the length of this sentence goes up (3, 5, 7, ...) we get the following numbers of parse trees: 1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ... (These are the *Catalan numbers*, which we saw in an exercise in Section 6.5). The last of these is for a sentence of length 23, the average length of sentences in the WSJ section of Penn Treebank. For a sentence of length 50 there would be over 10^{12} parses, and this is only half the length of the Piglet sentence (Section 7.2), which young children process effortlessly. No practical NLP system could construct all millions of trees for a sentence and choose the appropriate one in the context. It's clear that humans don't do this either!

Note that the problem is not with our choice of example. [Church and Patil, 1982] point out that the syntactic ambiguity of PP attachment in sentences like (58) also grows in proportion to the Catalan numbers.

(58) Put the block in the box on the table.

So much for structural ambiguity; what about lexical ambiguity? As soon as we try to construct a broad-coverage grammar, we are forced to make lexical entries highly ambiguous for their part of speech. In a toy grammar, *a* is only a determiner, *dog* is only a noun, and *runs* is only a verb. However, in a broad-coverage grammar, *a* is also a noun (e.g. *part a*), *dog* is also a verb (meaning to follow closely), and *runs* is also a noun (e.g. *ski runs*). In fact, all words can be referred to by name: e.g. *the verb 'ate' is spelled with three letters*; in speech we do not need to supply quotation marks. Furthermore, it is possible to *verb* most nouns. Thus a parser for a broad-coverage grammar will be overwhelmed with ambiguity. Even complete gibberish will often have a reading, e.g. *the a are of I*. As [Abney, 1996b] has pointed out, this is not word salad but a grammatical noun phrase, in which *are* is a noun meaning a hundredth of a hectare (or 100 sq m), and *a* and *I* are nouns designating coordinates, as shown in Figure 8.1.

Even though this phrase is unlikely, it is still grammatical and a a broad-coverage parser should be able to construct a parse tree for it. Similarly, sentences which seem to be unambiguous, such as *John saw Mary*, turn out to have other readings we would not have anticipated (as Abney explains). This ambiguity is unavoidable, and leads to horrendous inefficiency in parsing seemingly inoccuous sentences.

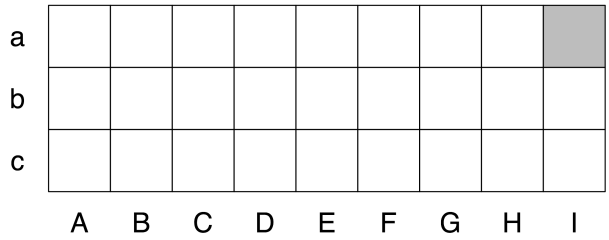


Figure 8.1: The a are of I

Let's look more closely at this issue of efficiency. The top-down recursive-descent parser presented in Chapter 7 can be very inefficient, since it often builds and discards the same sub-structure many times over. We see this in Figure 8.1, where a phrase *the block* is identified as a noun phrase several times, and where this information is discarded each time we backtrack.

Note

You should try the recursive-descent parser demo if you haven't already: `nltk_lite.draw.srparser.demo()`

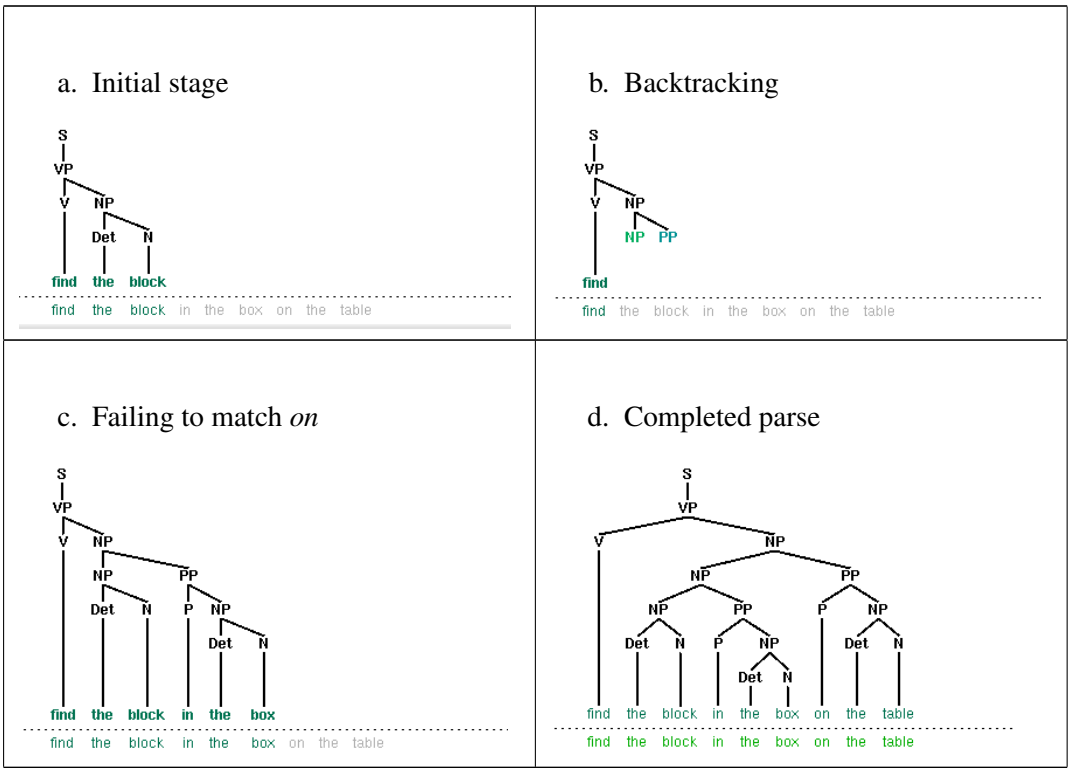


Table 8.1: Backtracking and Repeated Parsing of Subtrees

In this chapter, we will present two independent methods for dealing with ambiguity. The first is *chart parsing*, which uses the algorithmic technique of dynamic programming to derive the parses of an ambiguous sentence more *efficiently*. The second is *probabilistic parsing*, which allows us to *rank* the parses of an ambiguous sentence on the basis of evidence from corpora.

8.2 Chart Parsing

In the introduction to this chapter, we pointed out that the simple parsers discussed in Chapter 7 suffered from limitations in both completeness and efficiency. In order to remedy these, we will apply the algorithm design technique of *dynamic programming* to the parsing problem. As we saw in Section 6.5.3, dynamic programming stores intermediate results and re-uses them when appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing, allowing us to store

partial solutions to the parsing task and then look them up as necessary in order to efficiently arrive at a complete solution. This approach to parsing is known as **chart parsing**, and is the focus of this section.

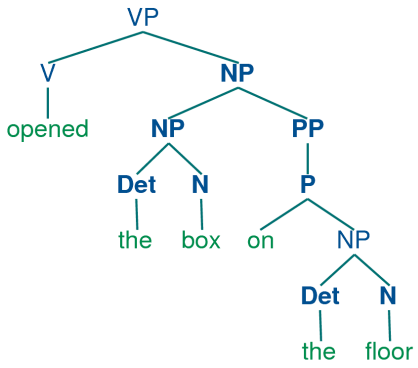
8.2.1 Well-formed Substring Tables

Let's start off by defining a simple grammar.

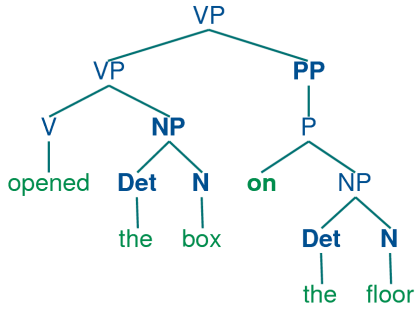
```
>>> from nltk_lite.parse import cfg
>>> grammar = cfg.parse_grammar("""
... S -> NP VP
... PP -> P NP
... NP -> Det N | NP PP
... VP -> V NP | VP PP
... Det -> 'the'
... N -> 'kids' | 'box' | 'floor'
... V -> 'opened'
... P -> 'on'
... """)
```

As you can see, this grammar allows the VP *opened the box on the floor* to be analysed in two ways, depending on where the PP is attached.

(59a)



(59b)



Dynamic programming allows us to build the PP *on the floor* just once. The first time we build it we save it in a table, then we look it up when we need to use it as a subconstituent of either the object NP or the higher VP. This table is known as a **well-formed substring table** (or WFST for short). We will show how to construct the WFST bottom-up so as to systematically record what syntactic constituents have been found.

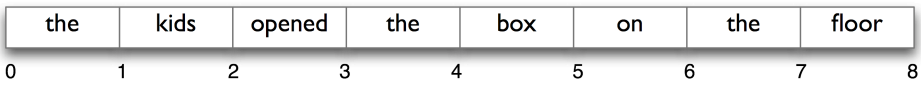


Figure 8.2: Slice Points in the Input String

Let's set our input to be the sentence *the kids opened the box on the floor*. It is helpful to think of the input as being indexed like a Python list. We have illustrated this in Figure 8.2.
This allows us to say that, for instance, the word *opened* spans (2, 3) in the input. This is reminiscent of the slice notation:

```
>>> tokens = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]
>>> tokens[2:3]
['opened']
```

In a WFST, we record the position of the words by filling in cells in a triangular matrix: the vertical axis will denote the start position of a substring, while the horizontal axis will denote the end position (thus *opened* will appear in the cell with coordinates (2, 3)). To simplify this presentation, we will assume each word has a unique lexical category, and we will store this (not the word) in the matrix. So cell (2, 3) will contain the entry V. More generally, if our input string is $a_1 a_2 \quad a_n$, and our grammar contains a production of the form $A \rightarrow a_i$, then we add A to the cell (i-1, i).

So, for every word in `tokens`, we can look up in our grammar what category it belongs to.

```
>>> grammar.productions(rhs=tokens[2])
[V -> 'opened']
```

For our WFST, we create an $(n-1) \times (n-1)$ matrix as a list of lists in Python, and initialize it with the lexical categories of each token, in the `init_wfst()` function in Listing 8.1. We also define a utility function `display()` to pretty-print the WFST for us. As expected, there is a V in cell (2, 3).

Returning to our tabular representation, given that we have DET in cell (0, 1), and N in cell (1, 2), what should we put into cell (0, 2)? In other words, what syntactic category derives *the kids*? We have already established that DET derives *the* and N derives *kids*, so we need to find a production of the form $A \rightarrow$ DET N, that is, a production whose right hand side matches the categories in the cells we have already found. From the grammar, we know that we can enter NP in cell (0,2).

More generally, we can enter A in (i, j) if there is a production $A \rightarrow B\ C$, and we find nonterminal B in (i, k) and C in (k, j). Listing 8.1 uses this inference step to complete the WFST.

> **Note**
>
> To help us easily retrieve productions by their right hand sides, we create an index for the grammar. This is an example of a space-time trade-off: we do a reverse lookup on the grammar, instead of having to check through entire list of productions each time we want to look up via the right hand side.

We conclude that there is a parse for the whole input string once we have constructed an S node that covers the whole input, from position *0* to position *8*; i.e., we can conclude that S $\Rightarrow^* a_1 a_2 \quad a_n$.

Notice that we have not used any built-in parsing functions here. We've implemented a complete, primitive chart parser from the ground up!

Listing 31 Acceptor Using Well-Formed Substring Table (based on CYK algorithm)

```
def init_wfst(tokens, grammar):
    numtokens = len(tokens)
    wfst = [['.' for i in range(numtokens+1)] for j in range(numtokens+1)]
    for i in range(numtokens):
        productions = grammar.productions(rhs=tokens[i])
        wfst[i][i+1] = productions[0].lhs()
    return wfst
def complete_wfst(wfst, tokens, trace=False):
    index = {}
    for prod in grammar.productions():
        index[prod.rhs()] = prod.lhs()
    numtokens = len(tokens)
    for span in range(2, numtokens+1):
        for start in range(numtokens+1-span):
            end = start + span
            for mid in range(start+1, end):
                nt1, nt2 = wfst[start][mid], wfst[mid][end]
                if (nt1,nt2) in index:
                    if trace:
                        print "[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \
                        (start, nt1, mid, nt2, end, start, index[(nt1,nt2)], end)
                    wfst[start][end] = index[(nt1,nt2)]
    return wfst
def display(wfst, tokens):
    print '\nWFST ' + ' '.join([("%-4d" % i) for i in range(1, len(wfst))])
    for i in range(len(wfst)-1):
        print "%d   " % i,
        for j in range(1, len(wfst)):
            print "%-4s" % wfst[i][j],
        print

>>> wfst0 = init_wfst(tokens, grammar)
>>> display(wfst0, tokens)
WFST 1    2    3    4    5    6    7    8
0    Det  .    .    .    .    .    .    .
1    .    N    .    .    .    .    .    .
2    .    .    V    .    .    .    .    .
3    .    .    .    Det  .    .    .    .
4    .    .    .    .    N    .    .    .
5    .    .    .    .    .    P    .    .
6    .    .    .    .    .    .    Det  .
7    .    .    .    .    .    .    .    N
>>> wfst1 = complete_wfst(wfst0, tokens)
>>> display(wfst1, tokens)
WFST 1    2    3    4    5    6    7    8
0    Det  NP   .    .    S    .    .    S
1    .    N    .    .    .    .    .    .
2    .    .    V    .    VP   .    .    VP
3    .    .    .    Det  NP   .    .    NP
4    .    .    .    .    N    .    .    .
5    .    .    .    .    .    P    .    PP
6    .    .    .    .    .    .    Det  NP
7    .    .    .    .    .    .    .    N
```

8.2.2 Charts

By setting `trace` to `True` when calling the function `complete_wfst()`, we get additional output.

```
>>> wfst1 = complete_wfst(wfst0, tokens, trace=True)
[0] Det [1]   N [2] ==> [0]  NP [2]
[3] Det [4]   N [5] ==> [3]  NP [5]
[6] Det [7]   N [8] ==> [6]  NP [8]
[2]   V [3]  NP [5] ==> [2]  VP [5]
[5]   P [6]  NP [8] ==> [5]  PP [8]
[0]  NP [2]  VP [5] ==> [0]   S [5]
[3]  NP [5]  PP [8] ==> [3]  NP [8]
[2]   V [3]  NP [8] ==> [2]  VP [8]
[2]  VP [5]  PP [8] ==> [2]  VP [8]
[0]  NP [2]  VP [8] ==> [0]   S [8]
```

For example, this says that since we found `Det` at `wfst[0][1]` and `N` at `wfst[1][2]`, we can add `NP` to `wfst[0][2]`. The same information can be represented in a directed acyclic graph, as shown in Figure 8.2(a). This graph is usually called a **chart**. Figure 8.2(b) is the corresponding graph representation, where we add a new edge labeled NP to cover the input from *0* to *2*.

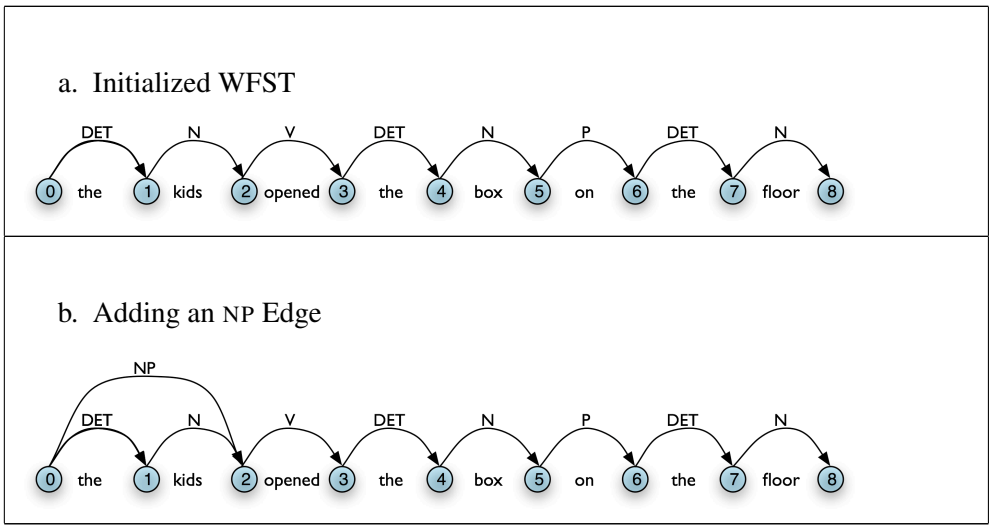


Table 8.2: A Graph Representation for the WFST

(Charts are more general than the WFSTs we have seen, since they can hold multiple hypotheses for a given span.)

A WFST is a data structure that can be used by a variety of parsing algorithms. The particular method for constructing a WFST that we have just seen and has some shortcomings. First, as you can see, the WFST is not itself a parse tree, so the technique is strictly speaking **recognizing** that a sentence is admitted by a grammar, rather than parsing it. Second, it requires every non-lexical grammar production to be *binary* (see Section 8.5.1). Although it is possible to convert an arbitrary CFG into this form, we would prefer to use an approach without such a requirement. Third, as a bottom-up approach it is potentially wasteful, being able to propose constituents in locations that would not be licensed by the grammar. Finally, the WFST did not represent the structural ambiguity in the sentence (i.e. the two

verb phrase readings). The VP in cell (*2,8*) was actually entered twice, once for a V NP reading, and oce for a VP PP reading. In the next section we will address these issues.

8.2.3 Exercises

1. ☼ Consider the sequence of words: *Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo*. This is a grammatically correct sentence, as explained at `http://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffa`. Consider the tree diagram presented on this Wikipedia page, and write down a suitable grammar. Normalise case to lowercase, to simulate the problem that a listener has when hearing this sentence. Can you find other parses for this sentence? How does the number of parse trees grow as the sentence gets longer? (More examples of these sentences can be found at `http://en.wikipedia.org/wiki/List_of_homophonous_phrases`).

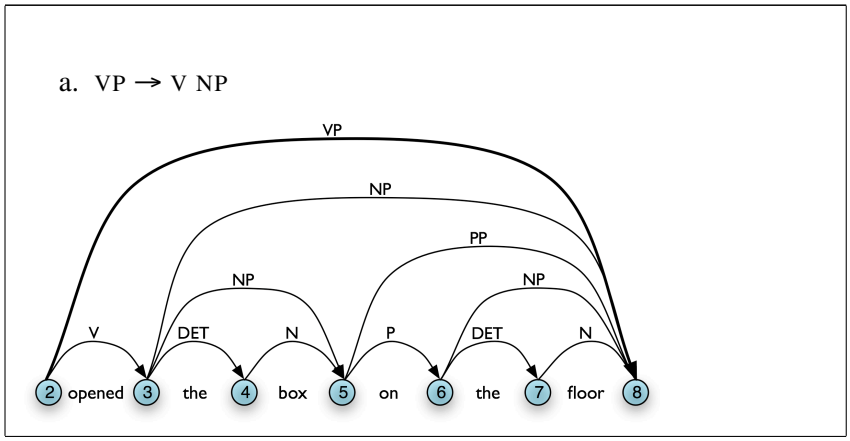
2. ◑ Consider the algorithm in Listing 8.1. Can you explain why parsing context-free grammar is proportional to n^3?

3. ◑ Modify the functions `init_wfst()` and `complete_wfst()` so that the contents of each cell in the WFST is a set of non-terminal symbols rather than a single non-terminal.

4. ★ Modify the functions `init_wfst()` and `complete_wfst()` so that when a non-terminal symbol is added to a cell in the WFST, it includes a record of the cells from which it was derived. Implement a function which will convert a WFST in this form to a parse tree.

8.3 Active Charts

One important aspect of the tabular approach to parsing can be seen more clearly if we look at the graph representation: given our grammar, there are two different ways to derive a top-level VP for the input, as shown in Table 8.3(a,b). In our graph representation, we simply combine the two sets of edges to yield Table 8.3(c).



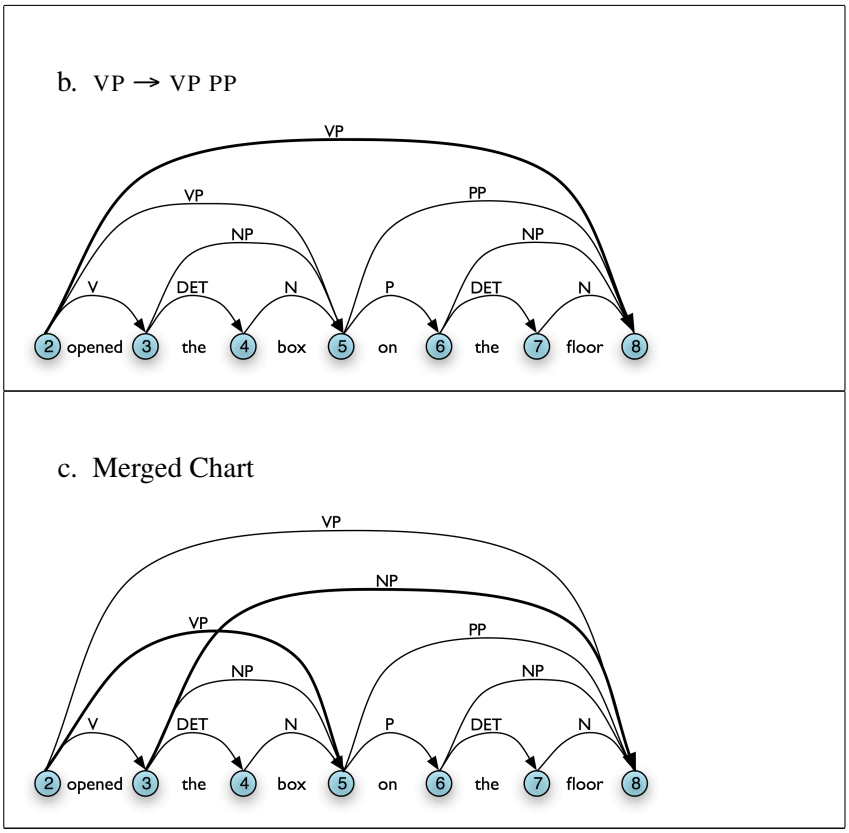


Table 8.3: Combining Multiple Parses in a Single Chart

However, given a WFST we cannot necessarily read off the justification for adding a particular edge. For example, in 8.3(b), `[Edge: VP, 2:8]` might owe its existence to a production VP → V NP PP. Unlike phrase structure trees, a WFST does not encode a relation of immediate dominance. In order to make such information available, we can label edges not just with a non-terminal category, but with the whole production which justified the addition of the edge. This is illustrated in Figure 8.3.

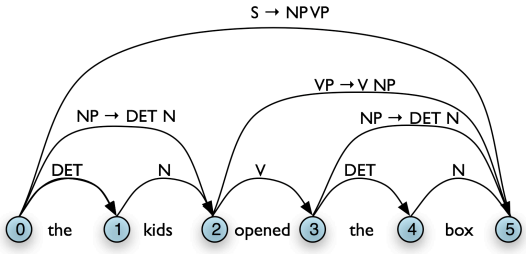


Figure 8.3: Chart Annotated with Productions

In general, a chart parser hypothesizes constituents (i.e. adds edges) based on the grammar, the tokens, and the constituents already found. Any constituent that is compatible with the current knowledge can be hypothesized; even though many of these hypothetical constituents will never be used in the final result. A WFST just records these hypotheses.

All of the edges that we've seen so far represent complete constituents. However, as we will see, it is helpful to hypothesize *incomplete* constituents. For example, the work done by a parser in processing the production *VP → V NP PP* can be reused when processing *VP → V NP*. Thus, we will record the hypothesis that "the V constituent *likes* is the beginning of a VP."

We can record such hypotheses by adding a **dot** to the edge's right hand side. Material to the left of the dot specifies what the constituent starts with; and material to the right of the dot specifies what still needs to be found in order to complete the constituent. For example, the edge in the Figure 8.4 records the hypothesis that "a VP starts with the V *likes*, but still needs an NP to become complete":

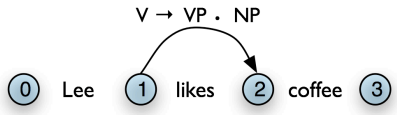


Figure 8.4: Chart Containing Incomplete VP Edge

These **dotted edges** are used to record all of the hypotheses that a chart parser makes about constituents in a sentence. Formally a dotted edge $[A \rightarrow c_1 \ldots c_d \bullet c_{d+1} \ldots c_n, (i, j)]$ records the hypothesis that a constituent of type A with span (i, j) starts with children $c_1 \ldots c_d$, but still needs children $c_{d+1} \ldots c_n$ to be complete ($c_1 \ldots c_d$ and $c_{d+1} \ldots c_n$ may be empty). If $d = n$, then $c_{d+1} \ldots c_n$ is empty and the edge represents a complete constituent and is called a **complete edge**. Otherwise, the edge represents an incomplete constituent, and is called an **incomplete edge**. In Figure 8.4(a), [VP → V NP •, (1, 3)] is a complete edge, and [VP → V • NP, (1, 2)] is an incomplete edge.

If $d = 0$, then $c_1 \ldots c_n$ is empty and the edge is called a **self-loop edge**. This is illustrated in Table 8.4(b). If a complete edge spans the entire sentence, and has the grammar's start symbol as its left-hand side, then the edge is called a **parse edge**, and it encodes one or more parse trees for the sentence. In Table 8.4(c), [S → NP VP •, (0, 3)] is a parse edge.

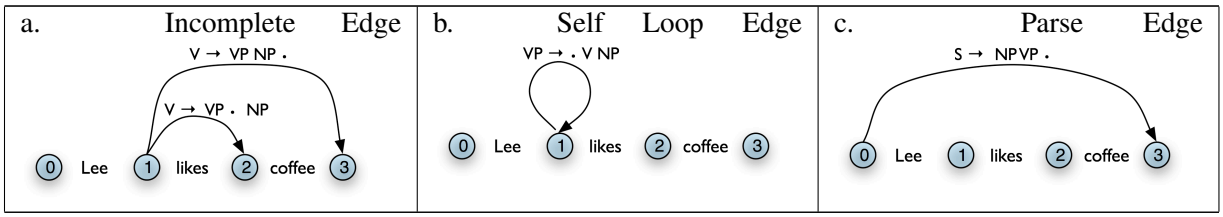


Table 8.4: Chart Terminology

8.3.1 The Chart Parser

To parse a sentence, a chart parser first creates an empty chart spanning the sentence. It then finds edges that are licensed by its knowledge about the sentence, and adds them to the chart one at a time until one or more parse edges are found. The edges that it adds can be licensed in one of three ways:

1. The *input* can license an edge. In particular, each word w_i in the input licenses the complete edge $[w_i \rightarrow \bullet, (i, i{+}1)]$.
2. The *grammar* can license an edge. In particular, each grammar production A → α licenses the self-loop edge $[A \rightarrow \bullet\ \alpha, (i, i)]$ for every i, $0 \leq i < n$.
3. The *current chart contents* can license an edge.

However, it is not wise to add *all* licensed edges to the chart, since many of them will not be used in any complete parse. For example, even though the edge in the following chart is licensed (by the grammar), it will never be used in a complete parse:

Figure 8.5: Chart Containing Redundant Edge

Chart parsers therefore use a set of **rules** to heuristically decide when an edge should be added to a chart. This set of rules, along with a specification of when they should be applied, forms a **strategy**.

8.3.2 The Fundamental Rule

One rule is particularly important, since it is used by every chart parser: the **Fundamental Rule**. This rule is used to combine an incomplete edge that's expecting a nonterminal *B* with a following, complete edge whose left hand side is *B*.

(60) **Fundamental Rule**

```
If the chart contains the edges
  [A → α • B β , (i, j)]
  [B → γ • , (j, k)]
then add the new edge
  [A → α B • β , (i, k)]
where α, β, and γ are (possibly empty) sequences
of terminals or non-terminals
```

Note that the dot has moved one place to the right, and the span of this new edige is the combined span of the other two. Note also that in adding this new edge we do not remove the other two, because they might be used again.

A somewhat more intuitive version of the operation of the Fundamental Rule can be given using chart diagrams. Thus, if we have a chart of the form shown in Table 8.5(a), then we can add a new complete edge as shown in Table 8.5(b).

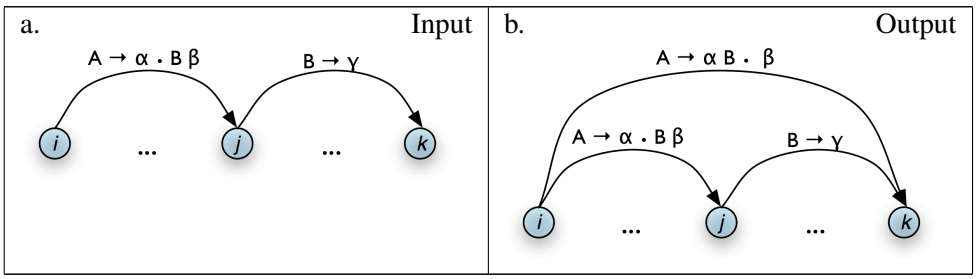


Table 8.5: Fundamental Rule

[2]The Fundamental Rule corresponds to the Completer function in the Earley algorithm; cf. [Jurafsky and Martin, 2000].

8.3.3 Bottom-Up Parsing

As we saw in Chapter 7, bottom-up parsing starts from the input string, and tries to find sequences of words and phrases that correspond to the *right hand* side of a grammar production. The parser then replaces these with the left-hand side of the production, until the whole sentence is reduced to an S. Bottom-up chart parsing is an extension of this approach in which hypotheses about structure are recorded as edges on a chart. In terms of our earlier terminology, bottom-up chart parsing can be seen as a parsing strategy; in other words, bottom-up is a particular choice of heuristics for adding new edges to a chart.

The general procedure for chart parsing is inductive: we start with a base case, and then show how we can move from a given state of the chart to a new state. Since we are working bottom-up, the base case for our induction will be determined by the words in the input string, so we add new edges for each word. Now, for the induction step, suppose the chart contains an edge labeled with constituent A. Since we are working bottom-up, we want to build constituents which can have an A as a daughter. In other words, we are going to look for productions of the form $B \rightarrow A\ \beta$ and use these to label new edges.

Let's look at the procedure a bit more formally. To create a bottom-up chart parser, we add to the Fundamental Rule two new rules: the **Bottom-Up Initialization Rule**; and the **Bottom-Up Predict Rule**. The Bottom-Up Initialization Rule says to add all edges licensed by the input.

(61) **Bottom-Up Initialization Rule**

For every word w_i add the edge
$[w_i \rightarrow \bullet\ ,\ (i,\ i+1)]$

Table 8.6(a) illustrates this rule using the chart notation, while Table 8.6(b) shows the bottom-up initialization for the input *Lee likes coffee*.

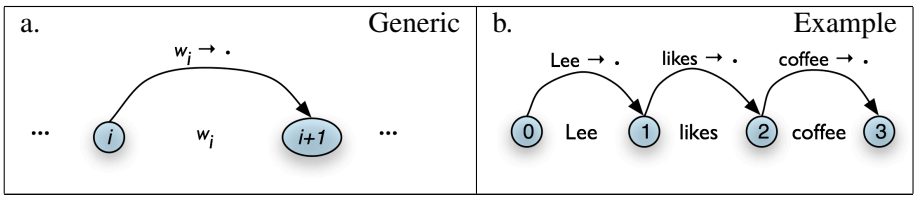


Table 8.6: Bottom-Up Initialization Rule

Notice that the dot on the right hand side of these productions is telling us that we have complete edges for the lexical items. By including this information, we can give a uniform statement of how the Fundamental Rule operates in Bottom-Up parsing, as we will shortly see.

Next, suppose the chart contains a complete edge e whose left hand category is A. Then the Bottom-Up Predict Rule requires the parser to add a self-loop edge at the left boundary of e for each grammar production whose right hand side begins with category A.

(62) **Bottom-Up Predict Rule**

If the chart contains the complete edge
$[A \rightarrow \alpha\ \bullet\ ,\ (i,\ j)]$
and the grammar contains the production
$B \rightarrow A\ \beta$
then add the self-loop edge

[B → • A β , (i, i)]

Graphically, if the chart looks as in Figure 8.7(a), then the Bottom-Up Predict Rule tells the parser to augment the chart as shown in Figure 8.7(b).

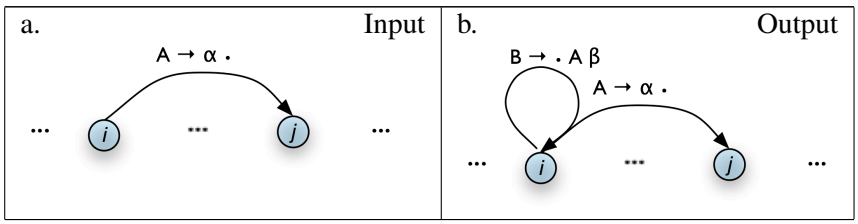


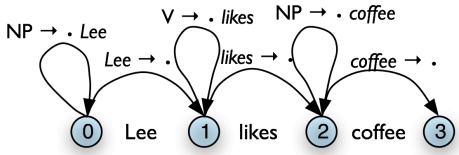
Table 8.7: Bottom-Up Prediction Rule

To continue our earlier example, let's suppose that our grammar contains the lexical productions shown in (63a). This allows us to add three self-loop edges to the chart, as shown in (63b).

(63a) NP → *Lee* | *coffee*

V → *likes*

(63b)



Once our chart contains an instance of the pattern shown in Figure 8.7(b), we can use the Fundamental Rule to add an edge where we have "moved the dot" one position to the right, as shown in Figure 8.8 (we have omitted the self-loop edges for simplicity.)

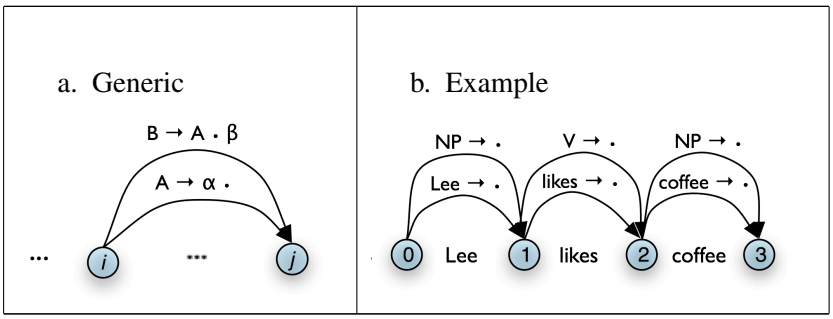


Table 8.8: Fundamental Rule used in Bottom-Up Parsing

We will now be able to add new self-loop edges such as [S → • NP VP, (0, 0)] and [VP → • VP NP, (1, 1)], and use these to build more complete edges.

Using these three productions, we can parse a sentence as shown in (64).

(64) **Bottom-Up Strategy**

Create an empty chart spanning the sentence.

```
Apply the Bottom-Up Initialization Rule to each word.
Until no more edges are added:
  Apply the Bottom-Up Predict Rule everywhere it applies.
  Apply the Fundamental Rule everywhere it applies.
Return all of the parse trees corresponding to the parse edges in the chart
```

NLTK provides a useful interactive tool for visualizing the way in which charts are built, `nltk_lite.draw.chart.demo()`. The tool comes with a pre-defined input string and grammar, but both of these can be readily modified with options inside the *Edit* menu. Figure 8.6 illustrates a window after the grammar has been updated:

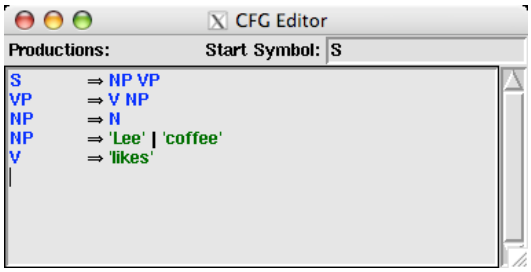


Figure 8.6: Modifing the *demo()* grammar

Note

To get the symbol ⇒ illustrated in Figure 8.6. you just have to type the keyboard characters *'->'*.

Figure 8.7 illustrates the tool interface. In order to invoke a rule, you simply click one of the green buttons at the bottom of the window. We show the state of the chart on the input *Lee likes coffee* after three applications of the Bottom-Up Initialiation Rule, followed by successive applications of the Bottom-Up Predict Rule and the Fundamental Rule.

Notice that in the topmost pane of the window, there is a partial tree showing that we have constructed an S with an NP subject in the expectation that we will be able to find a VP.

8.3.4 Top-Down Parsing

Top-down chart parsing works in a similar way to the recursive descent parser discussed in Chapter 7, in that it starts off with the top-level goal of finding an S. This goal is then broken into the subgoals of trying to find constituents such as NP and VP which can be immediatedly dominated by S. To create a top-down chart parser, we use the Fundamental Rule as before plus three other rules: the **Top-Down Initialization Rule**, the **Top-Down Expand Rule**, and the **Top-Down Match Rule**. The Top-Down Initialization Rule in (65) captures the fact that the root of any parse must be the start symbol S. It is illustrated graphically in Table 8.9.

(65) **Top-Down Initialization Rule**

```
For every grammar production of the form:
  s → α
add the self-loop edge:
  [s →  • α, (0, 0)]
```

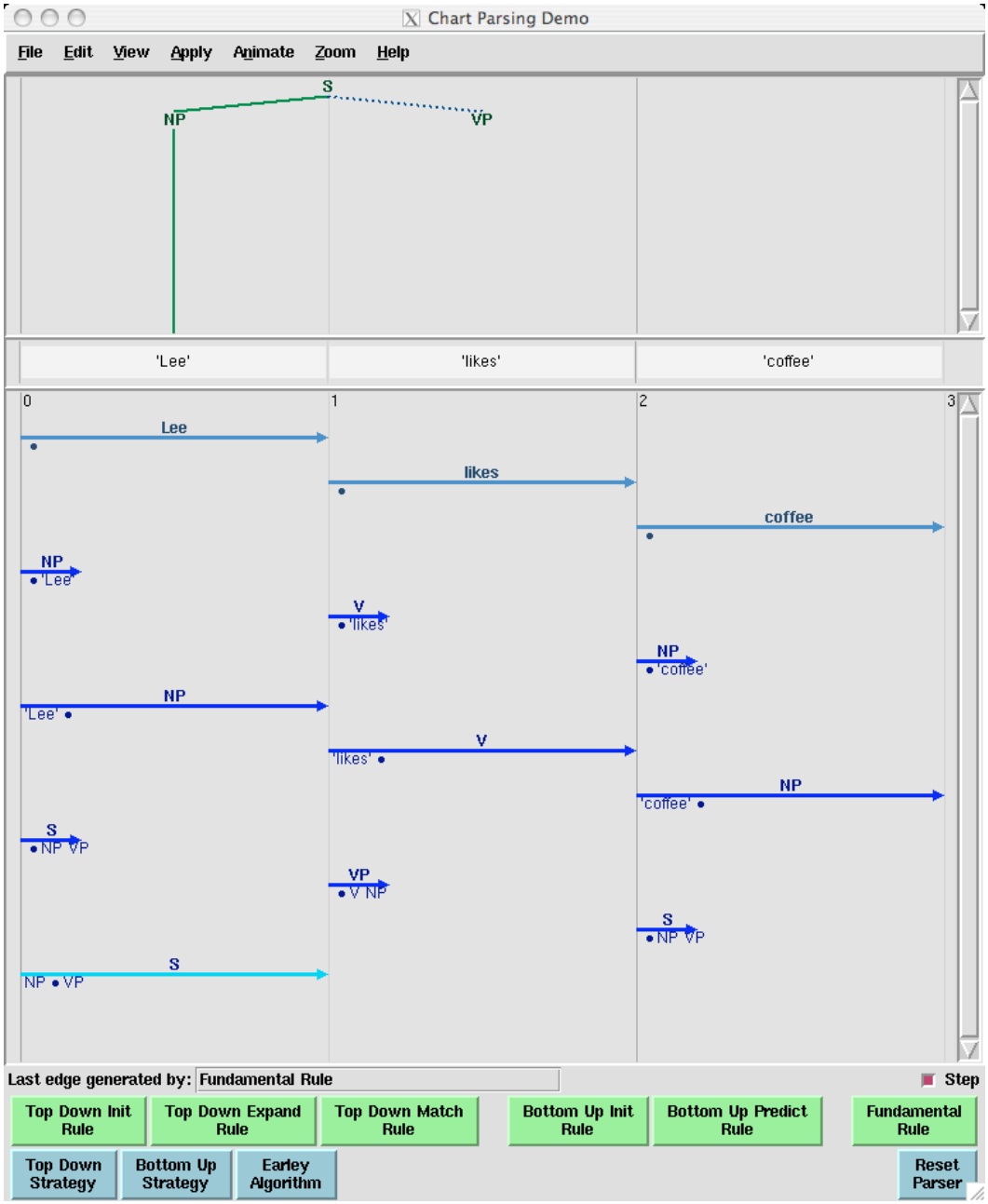


Figure 8.7: Incomplete chart for *Lee likes coffee*

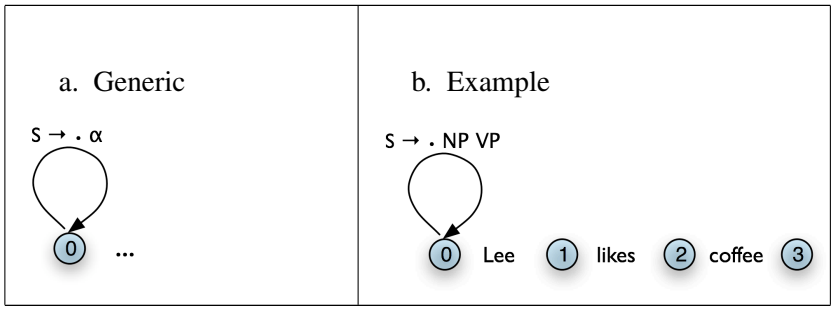


Table 8.9: Top-Down Initialization Rule

As we mentioned before, the dot on the right hand side of a production records how far our goals have been satisfied. So in Figure 8.9(b), we are predicting that we will be able to find an NP and a VP, but have not yet satisfied these subgoals. So how do we pursue them? In order to find an NP, for instance, we need to invoke a production which has NP on its left hand side. The step of adding the required edge to the chart is accomplished with the Top-Down Expand Rule (66). This tells us that if our chart contains an incomplete edge whose dot is followed by a nonterminal *B*, then the parser should add any self-loop edges licensed by the grammar whose left-hand side is *B*.

(66) **Top-Down Expand Rule**

```
If the chart contains the incomplete edge
  [A → α • B β , (i, j)]
then for each grammar production
  B → γ
add the edge
  [B → • γ , (j, j)]
```

Thus, given a chart that looks like the one in Table 8.10(a), the Top-Down Expand Rule augments it with the edge shown in Table 8.10(b). In terms of our running example, we now have the chart shown in Table 8.10(c).

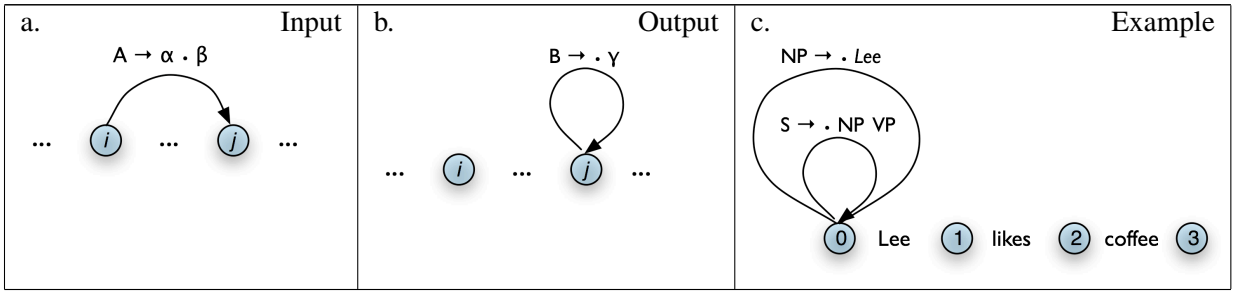


Table 8.10: Top-Down Expand Rule

The Top-Down Match rule allows the predictions of the grammar to be matched against the input string. Thus, if the chart contains an incomplete edge whose dot is followed by a terminal *w*, then the parser should add an edge if the terminal corresponds to the current input symbol.

(67) **Top-Down Match Rule**

```
If the chart contains the incomplete edge
  [A → α • wj β, (i, j)],
where wj is the j th word of the input,
then add a new complete edge
  [wj → • , (j, j+1)]
```

Graphically, the Top-Down Match rule takes us from Table 8.11(a), to Table 8.11(b).

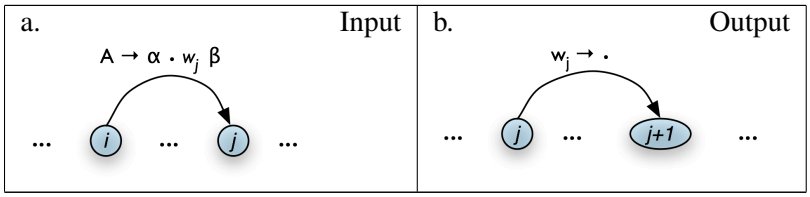


Table 8.11: Top-Down Match Rule

Figure 8.12(a) illustrates how our example chart after applying the Top-Down Match rule. What rule is relevant now? The Fundamental Rule. If we remove the self-loop edges from Figure 8.12(a) for simplicity, the Fundamental Rule gives us Figure 8.12(b).

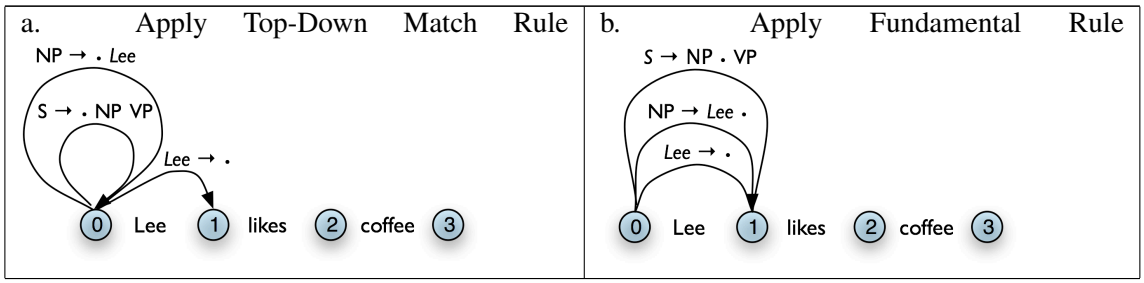


Table 8.12: Top-Down Example (cont)

Using these four rules, we can parse a sentence top-down as shown in (68).

(68) **Top-Down Strategy**

```
Create an empty chart spanning the sentence.
Apply the Top-Down Initialization Rule.
Until no more edges are added:
  Apply the Top-Down Expand Rule everywhere it applies.
  Apply the Top-Down Match Rule everywhere it applies.
  Apply the Fundamental Rule everywhere it applies.
Return all of the parse trees corresponding to the parse edges in
the chart.
```

We encourage you to experiment with the NLTK chart parser demo, as before, in order to test out the top-down strategy yourself.

8.3.5 The Earley Algorithm

The Earley algorithm [Earley, 1970] is a parsing strategy that resembles the Top-Down Strategy, but deals more efficiently with matching against the input string. Table 8.13 shows the correspondence between the parsing rules introduced above and the rules used by the Earley algorithm.

Top-Down/Bottom-Up	**Earley**
Top-Down Initialization Rule Top-Down Expand Rule	Predictor Rule
Top-Down/Bottom-Up Match Rule	Scanner Rule
Fundamental Rule	Completer Rule

Table 8.13: Terminology for rules in the Earley algorithm

Let's look in more detail at the Scanner Rule. Suppose the chart contains an incomplete edge with a lexical category P immediately after the dot, the next word in the input is w, P is a part-of-speech label for w. Then the Scanner Rule admits a new complete edge in which P dominates w. More precisely:

(69) **Scanner Rule**

If the chart contains the incomplete edge
  [$A \rightarrow \alpha \bullet P\ \beta$, ($i$, j)]
and w_j is the j^{th} word of the input,
and P is a valid part of speech for w_j,
then add the new complete edges
  [$P \rightarrow w_j \bullet$, ($j$, j+1)]
  [$w_j \rightarrow \bullet$, (j, j+1)]

To illustrate, suppose the input is of the form *I saw* ..., and the chart already contains the edge [VP → • V ..., (1, 1)]. Then the Scanner Rule will add to the chart the edges [V -> 'saw', (1, 2)] and ['saw'→ •, (1, 2)]. So in effect the Scanner Rule packages up a sequence of three rule applications: the Bottom-Up Initialization Rule for [$w \rightarrow \bullet$, (j, j+1)], the Top-Down Expand Rule for [$P \rightarrow \bullet\ w_j$, ($j$, j)], and the Fundamental Rule for [$P \rightarrow w_j \bullet$, ($j$, j+1))]. This is considerably more efficient than the Top-Down Strategy, which adds a new edge of the form [$P \rightarrow \bullet\ w$, (j, j)] for *every* lexical rule $P \rightarrow w$, regardless of whether w can be found in the input. By contrast with Bottom-Up Initialization, however, the Earley algorithm proceeds strictly left-to-right through the input, applying all applicable rules at that point in the chart, and never backtracking. The NLTK chart parser demo, described above, allows the option of parsing according to the Earley algorithm.

8.3.6 Chart Parsing in NLTK

`nltk_lite.parse.chart` defines a simple yet flexible chart parser, `ChartParse`. A new chart parser is constructed from a grammar and a list of chart rules (also known as a *strategy*). These rules will be applied, in order, until no new edges are added to the chart. In particular, `ChartParse` uses the algorithm shown in (70).

(70)
```
Until no new edges are added:
  For each chart rule R:
    Apply R to any applicable edges in the chart.
    Return any complete parses in the chart.
```

`nltk_lite.parse.chart` defines two ready-made strategies: `TD_STRATEGY`, a basic top-down strategy; and `BU_STRATEGY`, a basic bottom-up strategy. When constructing a chart parser, you can use either of these strategies, or create your own.

The following example illustrates the use of the chart parser. We start by defining a simple grammar, and tokenizing a sentence. We make sure it is a list (not an iterator), since we wish to use the same tokenized sentence several times.

Listing 32

```
from nltk_lite.parse import cfg, ChartParse, BU_STRATEGY
from nltk_lite import tokenize

grammar = cfg.parse_grammar('''
  NP  -> NNS | JJ NNS | NP CC NP
  NNS -> "men" | "women" | "children" | NNS CC NNS
  JJ  -> "old" | "young"
  CC  -> "and" | "or"
  ''')
parser = ChartParse(grammar, BU_STRATEGY)

>>> sent = list(tokenize.whitespace('old men and women'))
>>> for tree in parser.get_parse_list(sent):
...     print tree
(NP: (JJ: 'old') (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women')))
(NP: (NP: (JJ: 'old') (NNS: 'men')) (CC: 'and') (NP: (NNS: 'women')))
```

The `trace` parameter can be specified when creating a parser, to turn on tracing (higher trace levels produce more verbose output). Example 8.3 shows the trace output for parsing a sentence with the bottom-up strategy. Notice that in this output, `'[-----]'` indicates a complete edge, `'>'` indicates a self-loop edge, and `'[----->'` indicates an incomplete edge.

8.3.7 Exercises

1. ☼ Use the graphical chart-parser interface to experiment with different rule invocation strategies. Come up with your own strategy which you can execute manually using the graphical interface. Describe the steps, and report any efficiency improvements it has (e.g. in terms of the size of the resulting chart). Do these improvements depend on the structure of the grammar? What do you think of the prospects for significant performance boosts from cleverer rule invocation strategies?

2. ☼ We have seen that a chart parser adds but never removes edges from a chart. Why?

3. ◑ Write a program to compare the efficiency of a top-down chart parser compared with a recursive descent parser (Section 7.5.1). Use the same grammar and input sentences for both. Compare their performance using the `timeit` module (Section 6.5.4).

Listing 33 Trace of Bottom-Up Parser

```
>>> parser = ChartParse(grammar, BU_STRATEGY, trace=2)
>>> trees = parser.get_parse_list(sent)
|.   old   .   men   .   and   .  women  .|
Bottom Up Init Rule:
|[---------]         .         .         .| [0:1] 'old'
|.         [---------]         .         .| [1:2] 'men'
|.         .         [---------]         .| [2:3] 'and'
|.         .         .         [---------]| [3:4] 'women'
Bottom Up Predict Rule:
|>         .         .         .         .| [0:0] JJ -> * 'old'
|.         >         .         .         .| [1:1] NNS -> * 'men'
|.         .         >         .         .| [2:2] CC -> * 'and'
|.         .         .         >         .| [3:3] NNS -> * 'women'
Fundamental Rule:
|[---------]         .         .         .| [0:1] JJ -> 'old' *
|.         [---------]         .         .| [1:2] NNS -> 'men' *
|.         .         [---------]         .| [2:3] CC -> 'and' *
|.         .         .         [---------]| [3:4] NNS -> 'women' *
Bottom Up Predict Rule:
|>         .         .         .         .| [0:0] NP -> * JJ NNS
|.         >         .         .         .| [1:1] NP -> * NNS
|.         >         .         .         .| [1:1] NNS -> * NNS CC NNS
|.         .         .         >         .| [3:3] NP -> * NNS
|.         .         .         >         .| [3:3] NNS -> * NNS CC NNS
Fundamental Rule:
|[---------]         .         .         .| [0:1] NP -> JJ * NNS
|.         [---------]         .         .| [1:2] NP -> NNS *
|.         [--------->         .         .| [1:2] NNS -> NNS * CC NNS
|[-------------------]         .         .| [0:2] NP -> JJ NNS *
|.         [------------------->         .| [1:3] NNS -> NNS CC * NNS
|.         .         .         [---------]| [3:4] NP -> NNS *
|.         .         .         [--------->| [3:4] NNS -> NNS * CC NNS
|.         [---------------------------]| [1:4] NNS -> NNS CC NNS *
|.         [---------------------------]| [1:4] NP -> NNS *
|.         [--------------------------->| [1:4] NNS -> NNS * CC NNS
|[=========================================]| [0:4] NP -> JJ NNS *
Bottom Up Predict Rule:
|.         >         .         .         .| [1:1] NP -> * NP CC NP
|>         .         .         .         .| [0:0] NP -> * NP CC NP
|.         .         .         >         .| [3:3] NP -> * NP CC NP
Fundamental Rule:
|.         [--------->         .         .| [1:2] NP -> NP * CC NP
|[------------------->         .         .| [0:2] NP -> NP * CC NP
|.         .         .         [--------->| [3:4] NP -> NP * CC NP
|.         [---------------------------->| [1:4] NP -> NP * CC NP
|[----------------------------------------->| [0:4] NP -> NP * CC NP
|.         [------------------->         .| [1:3] NP -> NP CC * NP
|[---------------------------->         .| [0:3] NP -> NP CC * NP
|.         [---------------------------]| [1:4] NP -> NP CC NP *
|[=========================================]| [0:4] NP -> NP CC NP *
|.         [---------------------------->| [1:4] NP -> NP * CC NP
|[----------------------------------------->| [0:4] NP -> NP * CC NP
```

8.4 Probabilistic Parsing

As we pointed out in the introduction to this chapter, dealing with ambiguity is a key challenge to broad coverage parsers. We have shown how chart parsing can help improve the efficiency of computing multiple parses of the same sentences. But the sheer number of parses can be just overwhelming. We will show how probabilistic parsing helps to manage a large space of parses. However, before we deal with these parsing issues, we must first back up and introduce weighted grammars.

8.4.1 Weighted Grammars

We begin by considering the verb *give*. This verb requires both a direct object (the thing being given) and an indirect object (the recipient). These complements can be given in either order, as illustrated in example (71). In the "prepositional dative" form, the indirect object appears last, and inside a prepositional phrase, while in the "double object" form, the indirect object comes first:

(71a) Kim gave a bone to the dog

(71b) Kim gave the dog a bone

Using the Penn Treebank sample, we can examine all instances of prepositional dative and double object constructions involving *give*, as shown in Listing 8.4.

We can observe a strong tendency for the shortest complement to appear first. However, this does not account for a form like `give NP: federal judges / NP: a raise`, where animacy may be playing a role. In fact there turn out to be a large number of contributing factors, as surveyed by [Bresnan and Hay, 2006].

How can such tendencies be expressed in a conventional context free grammar? It turns out that they cannot. However, we can address the problem by adding weights, or probabilities, to the productions of a grammar.

A **probabilistic context free grammar** (or *PCFG*) is a context free grammar that associates a probability with each of its productions. It generates the same set of parses for a text that the corresponding context free grammar does, and assigns a probability to each parse. The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

The simplest way to define a PCFG is to load it from a specially formatted string consisting of a sequence of weighted productions, where weights appear in brackets, as shown in Listing 8.5.

It is sometimes convenient to combine multiple productions into a single line, e.g. `VP -> 'saw' NP [0.4] | 'ate' [0.3] | 'gave' NP NP [0.3]`. In order to ensure that the trees generated by the grammar form a probability distribution, PCFG grammars impose the constraint that all productions with a given left-hand side must have probabilities that sum to one. The grammar in Listing 8.5 obeys this constraint: for `S`, there is only one production, with a probability of 1.0; for `VP`, 0.4+0.3+0.3=1.0; and for `NP`, 0.8+0.2=1.0. The parse tree returned by `get_parse()` includes probabilities:

```
>>> from nltk_lite.parse import ViterbiParse
>>> viterbi_parser = ViterbiParse(grammar)
>>> print viterbi_parser.get_parse(['Jack', 'saw', 'the', 'telescope'])
(S: (NP: 'Jack') (VP: 'saw' (NP: 'the' 'telescope'))) (p=0.064)
```

Listing 34

```
from nltk_lite.corpora import treebank
from string import join
def give(t):
    return t.node == 'VP' and len(t) > 2 and t[1].node == 'NP'\
           and (t[2].node == 'PP-DTV' or t[2].node == 'NP')\
           and ('give' in t[0].leaves() or 'gave' in t[0].leaves())
def sent(t):
    return join(token for token in t.leaves() if token[0] not in '*-0')
def print_node(t, width):
        output = "%s %s: %s / %s: %s" %\
            (sent(t[0]), t[1].node, sent(t[1]), t[2].node, sent(t[2]))
        if len(output) > width:
            output = output[:width] + "..."
        print output

>>> for tree in treebank.parsed():
...     for t in tree.subtrees(give):
...         print_node(t, 72)
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a cal...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than ``...
give NP: the president / NP: line-item veto power
```

Listing 35 Defining a Probabilistic Context Free Grammar (PCFG)

```
from nltk_lite.parse import pcfg
grammar = pcfg.parse_grammar("""
    S  -> NP VP              [1.0]
    VP -> 'saw' NP           [0.4]
    VP -> 'ate'              [0.3]
    VP -> 'gave' NP NP       [0.3]
    NP -> 'the' 'telescope'  [0.8]
    NP -> 'Jack'             [0.2]
    """)

>>> print grammar
Grammar with 6 productions (start state = S)
    S -> NP VP [1.0]
    VP -> 'saw' NP [0.4]
    VP -> 'ate' [0.3]
    VP -> 'gave' NP NP [0.3]
    NP -> 'the' 'telescope' [0.8]
    NP -> 'Jack' [0.2]
```

The next two sections introduce two probabilistic parsing algorithms for PCFGs. The first is an A* parser that uses Viterbi-style dynamic programming to find the single most likely parse for a given text. Whenever it finds multiple possible parses for a subtree, it discards all but the most likely parse. The second is a bottom-up chart parser that maintains a queue of edges, and adds them to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, allowing the parser to expand more likely edges before less likely ones. Different queue orderings are used to implement a variety of different search strategies. These algorithms are implemented in the `nltk_lite.parse.viterbi` and `nltk_lite.parse.pchart` modules.

8.4.2 A* Parser

An **A* Parser** is a bottom-up PCFG parser that uses dynamic programming to find the single most likely parse for a text [Klein and Manning, 2003]. It parses texts by iteratively filling in a **most likely constituents table**. This table records the most likely tree for each span and node value. For example, after parsing the sentence "I saw the man with the telescope" with the grammar `pcfg.toy1`, the most likely constituents table contains the following entries (amongst others):

Span	Node	Tree	Prob
[0:1]	NP	(NP: I)	0.15
[6:7]	NP	(NN: telescope)	0.5
[5:7]	NP	(NP: the telescope)	0.2
[4:7]	PP	(PP: with (NP: the telescope))	0.122
[0:4]	S	(S: (NP: I) (VP: saw (NP: the man)))	0.01365
[0:7]	S	(S: (NP: I) (VP: saw (NP: (NP: the man) (PP: with (NP: the telescope)))))	0.0004163250

Span	Node	Tree	Prob

Table 8.14: Fragment of Most Likely Constituents Table

Once the table has been completed, the parser returns the entry for the most likely constituent that spans the entire text, and whose node value is the start symbol. For this example, it would return the entry with a span of [0:6] and a node value of "S".

Note that we only record the *most likely* constituent for any given span and node value. For example, in the table above, there are actually two possible constituents that cover the span [1:6] and have "VP" node values.

1. "saw the man, who has the telescope":

   (VP: saw (NP: (NP: John) (PP: with (NP: the telescope))))

2. "used the telescope to see the man":

   (VP: saw (NP: John) (PP: with (NP: the telescope)))

Since the grammar we are using to parse the text indicates that the first of these tree structures has a higher probability, the parser discards the second one.

Filling in the Most Likely Constituents Table: Because the grammar used by `ViterbiParse` is a PCFG, the probability of each constituent can be calculated from the probabilities of its children. Since a constituent's children can never cover a larger span than the constituent itself, each entry of the most likely constituents table depends only on entries for constituents with *shorter* spans (or equal spans, in the case of unary and epsilon productions).

`ViterbiParse` takes advantage of this fact, and fills in the most likely constituent table incrementally. It starts by filling in the entries for all constituents that span a single element of text. After it has filled in all the table entries for constituents that span one element of text, it fills in the entries for constituents that span two elements of text. It continues filling in the entries for constituents spanning larger and larger portions of the text, until the entire table has been filled.

To find the most likely constituent with a given span and node value, `ViterbiParse` considers all productions that could produce that node value. For each production, it checks the most likely constituents table for sequences of children that collectively cover the span and that have the node values specified by the production's right hand side. If the tree formed by applying the production to the children has a higher probability than the current table entry, then it updates the most likely constituents table with the new tree.

Handling Unary Productions and Epsilon Productions: A minor difficulty is introduced by unary productions and epsilon productions: an entry of the most likely constituents table might depend on another entry with the same span. For example, if the grammar contains the production `V` → `VP`, then the table entries for `VP` depend on the entries for `V` with the same span. This can be a problem if the constituents are checked in the wrong order. For example, if the parser tries to find the most likely constituent for a `VP` spanning [1:3] before it finds the most likely constituents for `V` spanning [1:3], then it can't apply the `V` → `VP` production.

To solve this problem, `ViterbiParse` repeatedly checks each span until it finds no new table entries. Note that cyclic grammar productions (e.g. `V` → `V`) will *not* cause this procedure to enter an

infinite loop. Since all production probabilities are less than or equal to 1, any constituent generated by a cycle in the grammar will have a probability that is less than or equal to the original constituent; so `ViterbiParse` will discard it.

In NLTK, we create Viterbi parsers using `ViterbiParse()`. Note that since `ViterbiParse` only finds the single most likely parse, that `get_parse_list` will never return more than one parse.

Listing 36

```
from nltk_lite.parse import pcfg, ViterbiParse
from nltk_lite import tokenize
grammar = pcfg.parse_grammar('''
  NP  -> NNS [0.5] | JJ NNS [0.3] | NP CC NP [0.2]
  NNS -> "men" [0.1] | "women" [0.2] | "children" [0.3] | NNS CC NNS [0.4]
  JJ  -> "old" [0.4] | "young" [0.6]
  CC  -> "and" [0.9] | "or" [0.1]
  ''')
viterbi_parser = ViterbiParse(grammar)

>>> sent = list(tokenize.whitespace('old men and women'))
>>> print viterbi_parser.parse(sent)
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
```

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays the constituents that are considered, and indicates which ones are added to the most likely constituent table. It also indicates the likelihood for each constituent.

```
>>> viterbi_parser.trace(3)
>>> tree = viterbi_parser.parse(sent)
Inserting tokens into the most likely constituents table...
   Insert: |=...| old
   Insert: |.=..| men
   Insert: |..=.| and
   Insert: |...=| women
Finding the most likely constituents spanning 1 text elements...
   Insert: |=...| JJ -> 'old' [0.4]                0.4000000000
   Insert: |.=..| NNS -> 'men' [0.1]               0.1000000000
   Insert: |.=..| NP -> NNS [0.5]                  0.0500000000
   Insert: |..=.| CC -> 'and' [0.9]                0.9000000000
   Insert: |...=| NNS -> 'women' [0.2]             0.2000000000
   Insert: |...=| NP -> NNS [0.5]                  0.1000000000
Finding the most likely constituents spanning 2 text elements...
   Insert: |==..| NP -> JJ NNS [0.3]               0.0120000000
Finding the most likely constituents spanning 3 text elements...
   Insert: |.===| NP -> NP CC NP [0.2]             0.0009000000
   Insert: |.===| NNS -> NNS CC NNS [0.4]          0.0072000000
   Insert: |.===| NP -> NNS [0.5]                  0.0036000000
  Discard: |.===| NP -> NP CC NP [0.2]             0.0009000000
  Discard: |.===| NP -> NP CC NP [0.2]             0.0009000000
Finding the most likely constituents spanning 4 text elements...
```

```
   Insert: |====| NP -> JJ NNS [0.3]                 0.0008640000
  Discard: |====| NP -> NP CC NP [0.2]               0.0002160000
  Discard: |====| NP -> NP CC NP [0.2]               0.0002160000
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
```

8.4.3 A Bottom-Up PCFG Chart Parser

The *A* parser* described in the previous section finds the single most likely parse for a given text. However, when parsers are used in the context of a larger NLP system, it is often necessary to produce several alternative parses. In the context of an overall system, a parse that is assigned low probability by the parser might still have the best overall probability.

For example, a probabilistic parser might decide that the most likely parse for "I saw John with the cookie" is is the structure with the interpretation "I used my cookie to see John"; but that parse would be assigned a low probability by a semantic system. Combining the probability estimates from the parser and the semantic system, the parse with the interpretation "I saw John, who had my cookie" would be given a higher overall probability.

This section describes a probabilistic bottom-up chart parser. It maintains an **edge queue**, and adds these edges to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, and this allows the parser to insert the most probable edges first. Each time an edge is added to the chart, it may become possible to insert new edges, so these are added to the queue. The bottom-up chart parser continues adding the edges in the queue to the chart until enough complete parses have been found, or until the edge queue is empty.

Like an edge in a regular chart, a probabilistic edge consists of a dotted production, a span, and a (partial) parse tree. However, unlike ordinary charts, this time the tree is weighted with a probability. Its probability is the product of the probability of the production that generated it and the probabilities of its children. For example, the probability of the edge `[Edge: S → NP • VP, 0:2]` is the probability of the PCFG production `S → NP VP` multiplied by the probability of its `NP` child. (Note that an edge's tree only includes children for elements to the left of the edge's dot. Thus, the edge's probability does *not* include probabilities for the constituents to the right of the edge's dot.)

8.4.4 Bottom-Up PCFG Strategies

The *edge queue* is a sorted list of edges that can be added to the chart. It is initialized with a single edge for each token in the text, with the form `[Edge: token |rarr| |dot|]`. As each edge from the queue is added to the chart, it may become possible to add further edges, according to two rules: (i) the Bottom-Up Initialization Rule can be used to add a self-loop edge whenever an edge whose dot is in position 0 is added to the chart; or (ii) the Fundamental Rule can be used to combine a new edge with edges already present in the chart. These additional edges are queued for addition to the chart.

By changing the sort order used by the queue, we can control the strategy that the parser uses to explore the search space. Since there are a wide variety of reasonable search strategies, `BottomUpChartParse()` does not define any sort order. Instead, different strategies are implemented in subclasses of `BottomUpChartParse()`.

Lowest Cost First: The simplest way to order the edge queue is to sort edges by the probabilities of their associated trees (`nltk_lite.parse.InsideParse()`). This ordering concentrates the efforts of the parser on those edges that are more likely to be correct analyses of their underlying tokens.

The probability of an edge's tree provides an upper bound on the probability of any parse produced using that edge. The probabilistic "cost" of using an edge to form a parse is one minus its tree's probability. Thus, inserting the edges with the most likely trees first results in a **lowest-cost-first search strategy**. Lowest-cost-first search is optimal: the first solution it finds is guaranteed to be the best solution.

However, lowest-cost-first search can be rather inefficient. Recall that a tree's probability is the product of the probabilities of all the productions used to generate it. Consequently, smaller trees tend to have higher probabilities than larger ones. Thus, lowest-cost-first search tends to work with edges having small trees before considering edges with larger trees. Yet any complete parse of the text will necessarily have a large tree, and so this strategy will tend to produce complete parses only once most other edges are processed.

Let's consider this problem from another angle. The basic shortcoming with lowest-cost-first search is that it ignores the probability that an edge's tree will be part of a complete parse. The parser will try parses that are locally coherent even if they are unlikely to form part of a complete parse. Unfortunately, it can be quite difficult to calculate the probability that a tree is part of a complete parse. However, we can use a variety of techniques to approximate that probability.

Best-First Search: This method sorts the edge queue in descending order of the edges' span, no the assumption that edges having a larger span are more likely to form part of a complete parse. Thus, `LongestParse` employs a **best-first search strategy**, where it inserts the edges that are closest to producing complete parses before trying any other edges. Best-first search is *not* an optimal search strategy: the first solution it finds is not guaranteed to be the best solution. However, it will usually find a complete parse much more quickly than lowest-cost-first search.

Beam Search: When large grammars are used to parse a text, the edge queue can grow quite long. The edges at the end of a large well-sorted queue are unlikely to be used. Therefore, it is reasonable to remove (or *prune*) these edges from the queue. This strategy is known as **beam search**; it only keeps the best partial results. The bottom-up chart parsers take an optional parameter `beam_size`; whenever the edge queue grows longer than this, it is pruned. This parameter is best used in conjunction with `InsideParse()`. Beam search reduces the space requirements for lowest-cost-first search, by discarding edges that are not likely to be used. But beam search also loses many of lowest-cost-first search's more useful properties. Beam search is not optimal: it is not guaranteed to find the best parse first. In fact, since it might prune a necessary edge, beam search is not even *complete*: it is not guaranteed to return a parse if one exists.

In NLTK we can construct these parsers using `InsideParse`, `LongestParse`, `RandomParse`.

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays edges as they are added to the chart, and shows the probability for each edges' tree.

```
>>> inside_parser.trace(3)
>>> trees = inside_parser.get_parse_list(sent)
  |. . . [-]| [3:4] 'women'                    [1.0]
  |. . [-] .| [2:3] 'and'                      [1.0]
  |. [-] . .| [1:2] 'men'                      [1.0]
  |[-] . . .| [0:1] 'old'                      [1.0]
  |. . [-] .| [2:3] CC -> 'and' *              [0.9]
  |. . > . .| [2:2] CC -> * 'and'              [0.9]
  |[-] . . .| [0:1] JJ -> 'old' *              [0.4]
  |> . . . .| [0:0] JJ -> * 'old'              [0.4]
```

Listing 37

```
from nltk_lite.parse import pchart
inside_parser = pchart.InsideParse(grammar)
longest_parser = pchart.LongestParse(grammar)
beam_parser = pchart.InsideParse(grammar, beam_size=20)

>>> print inside_parser.parse(sent)
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
>>> for tree in inside_parser.get_parse_list(sent):
...     print tree
(NP:
  (JJ: 'old')
  (NNS: (NNS: 'men') (CC: 'and') (NNS: 'women'))) (p=0.000864)
(NP:
  (NP: (JJ: 'old') (NNS: 'men'))
  (CC: 'and')
  (NP: (NNS: 'women'))) (p=0.000216)
```

```
  |> . . . .| [0:0] NP -> * JJ NNS                  [0.3]
  |. . . [-]| [3:4] NNS -> 'women' *                 [0.2]
  |. . . > .| [3:3] NP -> * NNS                      [0.5]
  |. . . > .| [3:3] NNS -> * NNS CC NNS              [0.4]
  |. . . > .| [3:3] NNS -> * 'women'                 [0.2]
  |[-> . . .| [0:1] NP -> JJ * NNS                   [0.12]
  |. . . [-]| [3:4] NP -> NNS *                      [0.1]
  |. . . > .| [3:3] NP -> * NP CC NP                 [0.2]
  |. [-] . .| [1:2] NNS -> 'men' *                   [0.1]
  |. > . . .| [1:1] NP -> * NNS                      [0.5]
  |. > . . .| [1:1] NNS -> * NNS CC NNS              [0.4]
  |. > . . .| [1:1] NNS -> * 'men'                   [0.1]
  |. . . [->| [3:4] NNS -> NNS * CC NNS              [0.08]
  |. [-] . .| [1:2] NP -> NNS *                      [0.05]
  |. > . . .| [1:1] NP -> * NP CC NP                 [0.2]
  |. [-> . .| [1:2] NNS -> NNS * CC NNS              [0.04]
  |. [---> .| [1:3] NNS -> NNS CC * NNS              [0.036]
  |. . . [->| [3:4] NP -> NP * CC NP                 [0.02]
  |[---] . .| [0:2] NP -> JJ NNS *                   [0.012]
  |> . . . .| [0:0] NP -> * NP CC NP                 [0.2]
  |. [-> . .| [1:2] NP -> NP * CC NP                 [0.01]
  |. [---> .| [1:3] NP -> NP CC * NP                 [0.009]
  |. [-----]| [1:4] NNS -> NNS CC NNS *              [0.0072]
  |. [-----]| [1:4] NP -> NNS *                      [0.0036]
  |. [-----># [1:4] NNS -> NNS * CC NNS             [0.00288]
  |[---> . .| [0:2] NP -> NP * CC NP                 [0.0024]
  |[-----> .| [0:3] NP -> NP CC * NP                 [0.00216]
  |. [-----]| [1:4] NP -> NP CC NP *                 [0.0009]
  |[=======]| [0:4] NP -> JJ NNS *                   [0.000864]
```

```
|. [----->| [1:4] NP -> NP * CC NP                    [0.00072]
|[=======]| [0:4] NP -> NP CC NP *                    [0.000216]
|. [----->| [1:4] NP -> NP * CC NP                    [0.00018]
|[------->| [0:4] NP -> NP * CC NP                    [0.0001728]
|[------->| [0:4] NP -> NP * CC NP                    [4.32e-05]
```

8.5 Grammar Induction

As we have seen, PCFG productions are just like CFG productions, adorned with probabilities. So far, we have simply specified these probabilities in the grammar. However, it is more usual to *estimate* these probabilities from training data, namely a collection of parse trees or *treebank*.

The simplest method uses *Maximum Likelihood Estimation*, so called because probabilities are chosen in order to maximize the likelihood of the training data. The probability of a production `VP` → `V NP PP` is *p(V,NP,PP | VP)*. We calculate this as follows:

$$P(V,NP,PP \mid VP) = \frac{count(VP \rightarrow V\ NP\ PP)}{count(VP \rightarrow \ldots)}$$

Here is a simple program that induces a grammar from the first three parse trees in the Penn Treebank corpus:

```
>>> from nltk_lite.corpora import treebank
>>> from itertools import islice
>>> productions = []
>>> S = cfg.Nonterminal('S')
>>> for tree in islice(treebank.parsed(),3):
...     productions += tree.productions()
>>> grammar = pcfg.induce(S, productions)
>>> for production in grammar.productions()[:10]:
...     print production
PP -> IN NP [1.0]
NNP -> 'Nov.' [0.0714285714286]
NNP -> 'Agnew' [0.0714285714286]
JJ -> 'industrial' [0.142857142857]
NP -> CD NNS [0.133333333333]
, -> ',' [1.0]
CC -> 'and' [1.0]
NNP -> 'Pierre' [0.0714285714286]
NP -> NNP NNP NNP NNP [0.0666666666667]
NNP -> 'Rudolph' [0.0714285714286]
```

8.5.1 Normal Forms

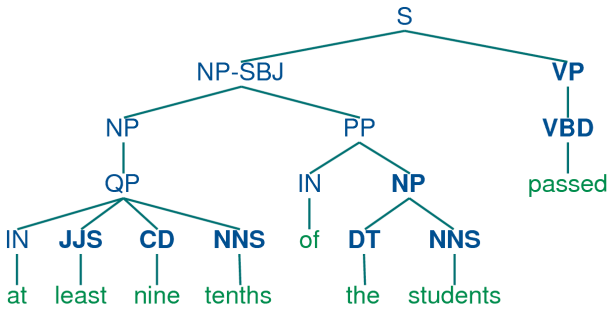
Grammar induction usually involves normalizing the grammar in various ways. The `nltk_lite.parse.treetransforms` module supports binarization (Chomsky Normal Form), parent annotation, Markov order-N smoothing, and unary collapsing. This information can be accessed by importing `treetransforms` from `nltk_lite.parse`, then calling `help(treetransforms)`.

```
>>> from nltk_lite.parse import bracket_parse
>>> from nltk_lite.parse import treetransforms
```

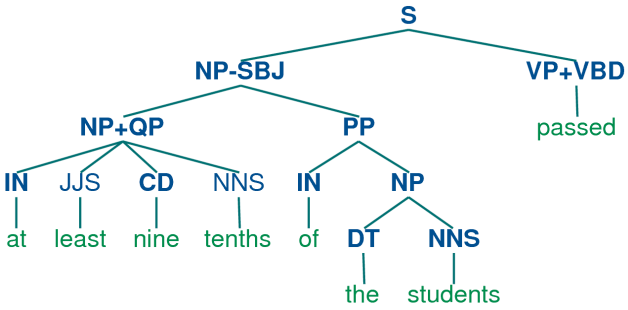
```
>>> treebank_string = """(S (NP-SBJ (NP (QP (IN at) (JJS least) (CD nine) (NNS tent
...     (PP (IN of) (NP (DT the) (NNS students) ))) (VP (VBD passed)))"""
>>> t = bracket_parse(treebank_string)
>>> print t
(S:
  (NP-SBJ:
    (NP: (QP: (IN: 'at') (JJS: 'least') (CD: 'nine') (NNS: 'tenths')))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP: (VBD: 'passed')))
>>> treetransforms.collapseUnary(t, collapsePOS=True)
>>> print t
(S:
  (NP-SBJ:
    (NP+QP: (IN: 'at') (JJS: 'least') (CD: 'nine') (NNS: 'tenths'))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP+VBD: 'passed'))
>>> treetransforms.chomskyNormalForm(t)
>>> print t
(S:
  (NP-SBJ:
    (NP+QP:
      (IN: 'at')
      (NP+QP|<JJS-CD-NNS>:
        (JJS: 'least')
        (NP+QP|<CD-NNS>: (CD: 'nine') (NNS: 'tenths'))))
    (PP: (IN: 'of') (NP: (DT: 'the') (NNS: 'students'))))
  (VP+VBD: 'passed'))
```

These trees are shown in (72).

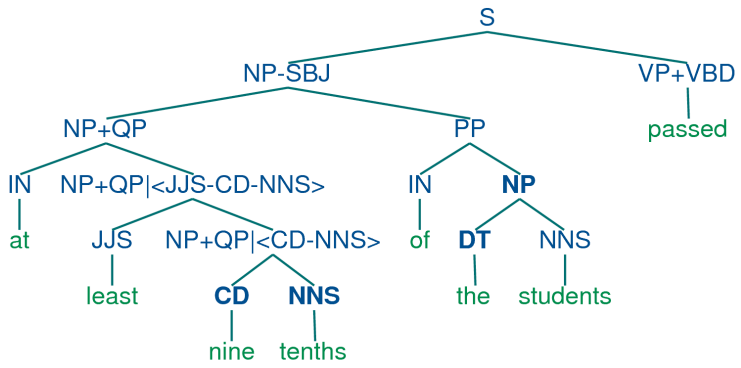
(72a)



(72b)



(72c)



8.6 Conclusion

8.7 Further Reading

- [Manning and Schutze, 1999] (esp chapter 12).
- [Klein and Manning, 2003]

Chapter 9

Feature Based Grammar

9.1 Introduction

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

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

9.2 Decomposing Lingustic Categories

9.2.1 Syntactic Agreement

Consider the following contrasts:

(73a) this dog

(73b) *these dog

(74a) these dogs

(74b) *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 (73) and (74) 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:

(75a) the dog runs

(75b) *the dog run

(76a) the dogs run

(76b) *the dogs runs

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

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

Table 9.1: Agreement Paradigm for English Regular Verbs

We can make the role of morphological properties a bit more explicit as illustrated in (77) and (78). These representations indicate that the verb agrees with its subject in person and number. (We use '3' as an abbreviation for 3rd person, 'SG' for singular and 'PL' for plural.)

	the	dog	run-s
(77a)		dog.3.SG	run-3.SG

	the	dog-s	run
(78a)		dog.3.PL	run-3.PL

Despite the undoubted interest of agreement as a topic in its own right, we have introduced it here for another reason: we want to look at what happens when we try encode agreement constraints in a context-free grammar. Suppose we take as our starting point the very simple CFG in (79).

(79)
```
S → NP VP
NP → DET N
VP → V

DET → 'this'
N → 'dog'
V → 'runs'
```

(79) 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):

(80)
```
S_SG → NP_SG VP_SG
S_PL → NP_PL VP_PL
NP_SG → DET_SG N_SG
NP_PL → DET_PL N_PL
VP_SG → V_SG
VP_PL → V_PL

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

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

9.2.2 Using Attributes and Constraints

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

(81)
```
N[NUM pl]
```

In (81), 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:

(82)
```
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 (80). Things become more interesting when we allow *variables* over feature values, and use these to state constraints. This is illustrated in (83).

(83a)
```
S → NP[NUM ?n] VP[NUM ?n]
```

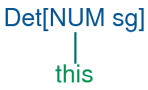
(83b)
```
NP[NUM ?n] → DET[NUM ?n] N[NUM ?n]
```

(83c)
```
VP[NUM ?n] → V[NUM ?n]
```

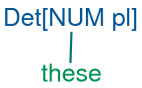
We are using '?*n*' as a variable over values of NUM; it can be instantiated either to *sg* or *pl*. Its scope is limited to individual rules. That is, within (83a), for example, ?*n* must be instantiated to the same constant value; we can read the rule as saying that whatever value NP takes for the 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):

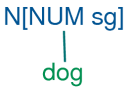
(84a)



(84b)



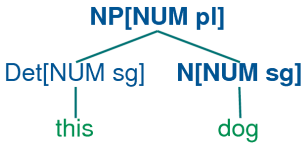
(85a)



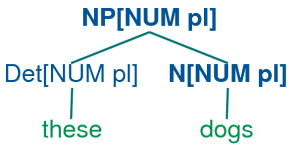
(85b)

Now (83b) says that whatever the NUM values of N and DET are, they have to be the same. Consequently, (83b) will permit (84a) and (85a) to be combined into an NP as shown in (86a) and it will also allow (84b) and (85b) to be combined, as in (86b). By contrast, (87a) and (87b) are prohibited because the roots of their constituent local trees differ in their values for the NUM feature.

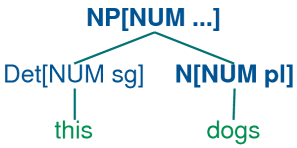
(86a)



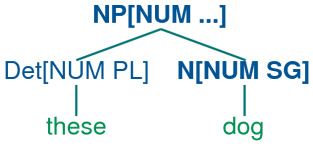
(86b)



(87a)

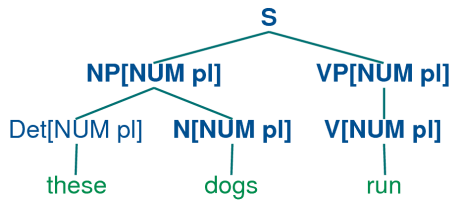


(87b)



Rule (83c) can be thought of as saying that the NUM value of the head verb has to be the same as the NUM value of the VP mother. Combined with (83a), 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.

(88)



Grammar (89) illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones. As you will see, the format of feature specifications in these productions inserts a `'=` between the feature and its value. In our exposition, we will stick to our earlier convention of just leaving space between the feature and value, except when we are directly referring to the NLTK grammar formalism.

(89)

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

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

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

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

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

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

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

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

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

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

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

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

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

A final detail about (89) is the statment `%start S`. This a 'directive' which tells the parser to take S as the start symbol for the grammar.

In general, when we are trying to develop even a very small grammar, it is convenient to put the rules in a file where they can be edited, tested and revised. Assuming we have saved (89) as a file named `'feat0.cfg'`, the function `GrammarFile.read_file()` allows us to read the grammar into NLTK, ready for use in parsing.

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

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

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

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

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

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

Listing 38 Trace of Feature-Based Chart Parser

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

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

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

9.2.3 Terminology

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

(90)
```
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:

(91)
```
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 (92).

(92)
```
N[NUM sg]
```

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

(93)
```
[POS N, NUM sg]
```

In fact, we regard (93) as our 'official' representation of a feature-based linguistic category, and (92) 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 (94).

(94)
$$\begin{bmatrix} \text{POS} & N \\ \text{AGR} & \begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & pl \\ \text{GND} & fem \end{bmatrix} \end{bmatrix}$$

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

There is no particular significance to the *order* of features in a feature structure. So (94) is equivalent to (94).

(95)
$$\begin{bmatrix} & \begin{bmatrix} \text{NUM} & pl \\ \text{PER} & 3 \\ \text{GND} & fem \end{bmatrix} \\ \text{AGR} & \\ \text{POS} & N \end{bmatrix}$$

Once we have the possibility of using features like AGR, we can refactor a grammar like (89) so that agreement features are bundled together. A tiny grammar illustrating this point is shown in (96).

(96)
```
S → NP[AGR ?n] VP[AGR ?n]
NP[AGR ?n] → PROPN[AGR ?n]
VP[TENSE ?t, AGR ?n] → COP[TENSE ?t, AGR ?n] Adj
COP[TENSE pres, AGR [NUM sg, PER 3]] → 'is'
PROPN[AGR [NUM sg, PER 3]] → 'Kim'
ADJ → 'happy'
```

9.2.4 Exercises

1. ☼ What constraints are required to correctly parse strings like *I am happy* and *she is happy* but not **you is happy* or **they am happy*? Implement two solutions for the present tense paradigm of the verb *be* in English, first taking Grammar (80) as your starting point, and then taking Grammar (96) as the starting point.

2. ☼ Develop a variant of grammar (89) which uses a COUNT to make the distinctions shown below:

(97a) The boy sings.

(97b) *Boy sings.

(98a) The boys sing.

(98b) Boys sing.

(99a) The boys sing.

(99b) Boys sing.

(100a) The water is precious.

(100b) Water is precious.

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

(101a)

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

'a beautiful picture'

(101b)	un-os INDEF-PL.MASC	cuadro-s picture-PL	hermos-os beautiful-PL.MASC
	'beautiful pictures'		
(101c)	un-a INDEF.SG.FEM	cortina curtain	hermos-a beautiful-SG.FEM
	'a beautiful curtain'		
(101d)	un-as INDEF.PL.FEM	cortina-s curtain	hermos-as beautiful-PL.FEM
	'beautiful curtains'		

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

9.3 Computing with Feature Structures

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

9.3.1 Feature Structures in NLTK

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

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

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

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

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

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

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

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

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

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

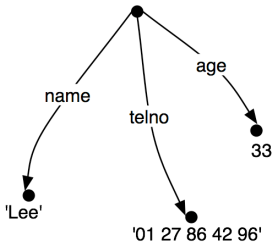
9.3.2 Feature Structures as Graphs

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

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

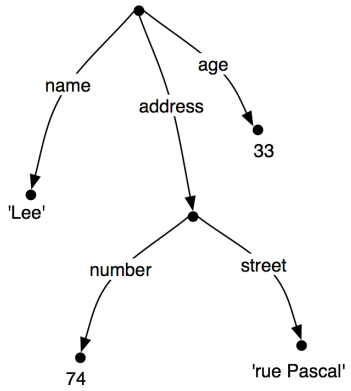
(102)
$$\begin{bmatrix} \text{NAME} & \textit{'Lee'} \\ \text{TELNO} & \textit{01 27 86 42 96} \\ \text{AGE} & \textit{33} \end{bmatrix}$$

It is sometimes helpful to view feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (103) is equivalent to the AVM (102).

(103)


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.

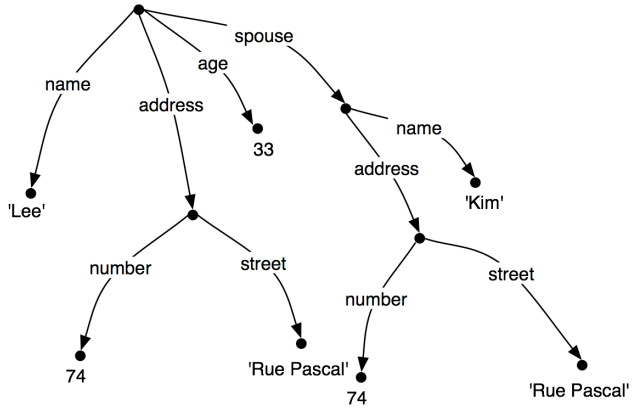
Just as before, feature values can be complex:



(104)

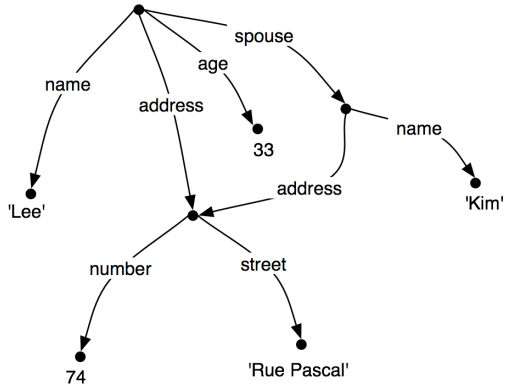
When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths in NLTK as tuples. Thus, `('address', 'street')` is a feature path whose value in (104) is the string 'rue 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 (105).



(105)

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



(106)

In other words, the value of the path `('ADDRESS')` in (106) is identical to the value of the path `('SPOUSE', 'ADDRESS')`. DAGs such as (106) are said to involve **structure sharing** or **reentrancy**. When two paths have the same value, they are said to be **equivalent**.

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

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

This is similar to more conventional displays of AVMs, as shown in (107).

(107)
$$\begin{bmatrix} \text{ADDRESS} & \boxed{1}\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

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

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

(108)
$$\begin{bmatrix} \text{A} & \text{'a'} \\ \text{B} & \boxed{1}\begin{bmatrix} \text{C} & \text{'c'} \end{bmatrix} \\ \text{D} & \boxed{1} \\ \text{E} & \boxed{1} \end{bmatrix}$$

9.3.3 Subsumption and Unification

It is standard to think of feature structures as providing **partial information** about some object, in the sense that we can order feature structures according to how general they are. For example, (109a) is more general (less specific) than (109b), which in turn is more general than (109c).

(109a)
$$\begin{bmatrix} \text{NUMBER} & 74 \end{bmatrix}$$

(109b)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix}$$

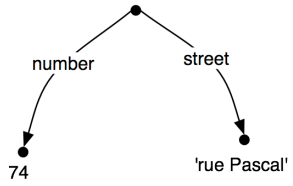
(109c)
$$\begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \\ \text{CITY} & \text{'Paris'} \end{bmatrix}$$

This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write FS_0 FS_1), then FS_1 must have all the paths and path equivalences of FS_0, and may have additional paths and equivalences as well. Thus, (105) subsumes (106), 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, (110) neither subsumes nor is subsumed by (109a).

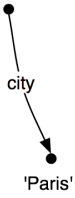
(110) $\left[\text{TELNO} \quad 01\ 27\ 86\ 42\ 96\right]$

So we have seen that some feature structures are more specific than others. How do we go about specialising a given feature structure? For example, we might decide that addresses should consist of not just a street number and a street name, but also a city. That is, we might want to *merge* graph (111b) with (111a) to yield (111c).

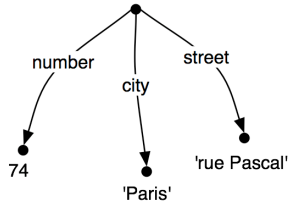
(111a)



(111b)



(111c)



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

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

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

(112) FS_0 $FS_1 = FS_1$ FS_0.

The same is true in NLTK:

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

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

(113) If FS_0 FS_1, then FS_0 $FS_1 = FS_1$

For example, the result of unifying (109b) with (109c) is (109c).

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 (105).

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

(114)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

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

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

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

(115)
$$\begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Lee'} \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \begin{bmatrix} \text{CITY} & \text{'Paris'} \\ \text{NUMBER} & 74 \\ \text{STREET} & \text{'rue Pascal'} \end{bmatrix} \\ \text{NAME} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

By contrast, the result is very different if `fs1` is unified with the structure-sharing version `fs2` (also shown as (106)):

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

(116)
$$\begin{bmatrix} & & \begin{bmatrix} \text{CITY} & \textit{'Paris'} \\ \text{NUMBER} & \textit{74} \\ \text{STREET} & \textit{'rue Pascal'} \end{bmatrix} \\ \text{ADDRESS} & \boxed{1} & \\ \text{NAME} & \textit{'Lee'} & \\ \text{SPOUSE} & \begin{bmatrix} \text{ADDRESS} & \boxed{1} \\ \text{NAME} & \textit{'Kim'} \end{bmatrix} & \end{bmatrix}$$

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

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

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

9.3.4 Exercises

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

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

Listing 39

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

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

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

Check your answers on the computer.

3. ◑ List two feature structures which subsume [A=?x, B=?x].
4. ◑ Ignoring structure sharing, give an informal algorithm for unifying two feature structures.

9.4 Extending a Feature-Based Grammar

9.4.1 Subcategorization

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

(117)
```
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 (89), shown in (118).

(118)
```
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=0, TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=1, TENSE=?t, NUM=?n] NP
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=2, TENSE=?t, NUM=?n] Sbar

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

V[SUBCAT=0, TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
V[SUBCAT=1, TENSE=pres, NUM=pl] -> 'see' | 'like'
V[SUBCAT=2, TENSE=pres, NUM=pl] -> 'say' | 'claim'

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

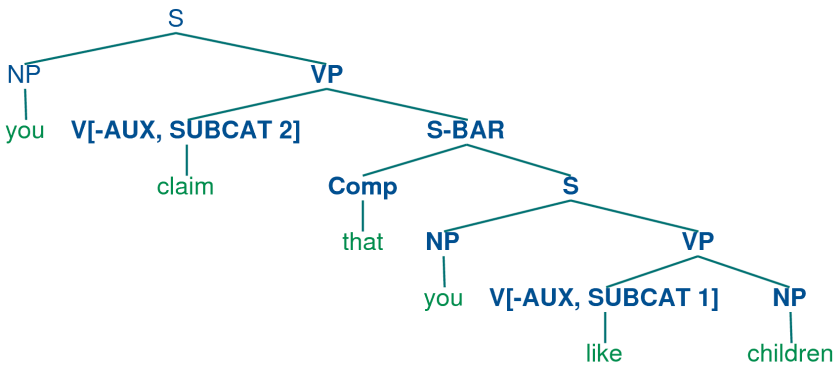
When we see a lexical category like V[SUBCAT *1*], we can interpret the SUBCAT specification as a pointer to the rule in which V[SUBCAT *1*] is introduced as the head daughter in a VP expansion rule. By convention, there is a one-to-one correspondence between SUBCAT values and rules which introduce lexical heads. It's worth noting that the choice of integer which acts as a value for SUBCAT is completely arbitrary — we could equally well have chosen 3999, 113 and 57 as our two values in (118). On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP.

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

(119)
```
S-BAR -> Comp S
Comp -> 'that'
```

The resulting structure is the following.

(120)



An alternative treatment of subcategorization, due originally to a framework known as categorial grammar, is represented in feature-based frameworks such as PATR and Head-driven Phrase Structure Grammar. Rather than using SUBCAT values as a way of indexing rules, the SUBCAT value directly encodes the valency of a head (the list of arguments that it can combine with). For example, a verb like *put* which takes NP and PP complements (*put the book on the table*) might be represented as (121):

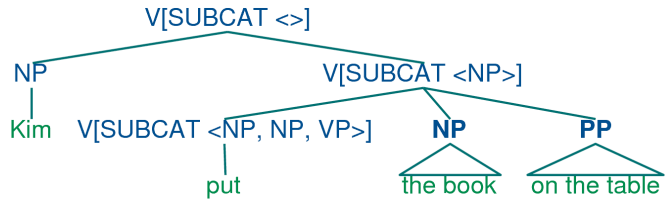
(121) V[SUBCAT NP, NP, PP]

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.

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

(123)

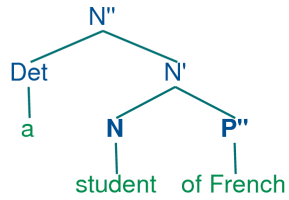


9.4.2 Heads Revisited

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

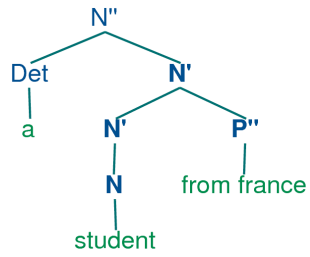
X-bar syntax (cf. [Chomsky, 1970], [Jackendoff, 1977]) addresses this issue by abstracting out the notion of **phrasal level**. It is usual to recognise three such levels. If N represents the lexical level, then N' represents the next level up, corresponding to the more traditional category NOM, while N" represents the phrasal level, corresponding to the category NP. (The primes here replace the typographically more demanding horizontal bars of [Chomsky, 1970]). (124) illustrates a representative structure.

(124)



The head of the structure (124) is N while N' and N" are called **(phrasal) projections** of N. N" is the **maximal projection**, and N is sometimes called the **zero projection**. One of the central claims of X-bar syntax is that all constituents share a structural similarity. Using X as a variable over N, V, A and P, we say that directly subcategorized *complements* of the head are always placed as sisters of the lexical head, whereas *adjuncts* are placed as sisters of the intermediate category, X'. Thus, the configuration of the P" adjunct in (125) contrasts with that of the complement P" in (124).

(125)



The productions in (126) illustrate how bar levels can be encoded using feature structures.

(126)
```
S → N[BAR 2] V[BAR 2]
N[BAR 2] → DET N[BAR 1]
N[BAR 1] → N[BAR 1] P[BAR 2]
N[BAR 1] → N[BAR 0] P[BAR 2]
```

9.4.3 Auxiliary verbs and Inversion

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

(127a) Do you like children?

(127b) Can Jody walk?

(128a) Rarely do you see Kim.

(128b) Never have I seen this dog.

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

(129a) *Like you children?

(129b) *Walks Jody?

(130a) *Rarely see you Kim.

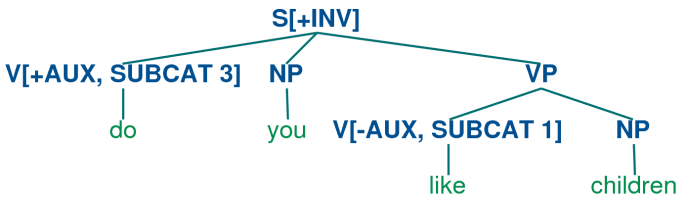
(130b) *Never saw I this dog.

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

(131)
```
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.) (132) illustrates the structure of an inverted clause.

(132)



9.4.4 Unbounded Dependency Constructions

Consider the following contrasts:

(133a) You like Jody.

(133b) *You like.

(134a) You put the card into the slot.

(134b) *You put into the slot.

(134c) *You put the card.

(134d) *You put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. Examples (133) and (134) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (135) and (136) illustrate.

(135a) Kim knows who you like.

(135b) This music, you really like.

(136a) Which card do you put into the slot?

(136b) 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 (135a), the preposed topic *this music* in (135b), or the *wh* phrases *which card/slot* in (136). It is common to say that sentences like (135) – (136) contain **gaps** where the obligatory complements have been omitted, and these gaps are sometimes made explicit using an underscore:

(137a) Which card do you put __ into the slot?

(137b) 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.

(138a) *Kim knows who you like Jody.

(138b) *This music, you really like hip-hop.

(139a) *Which card do you put this into the slot?

(139b) *Which slot do you put the card into this one?

The mutual co-occurence between filler and gap leads to (135) – (136) is sometimes termed a 'dependency'. One issue of considerable importance in theoretical linguistics has been the nature of the material that can intervene between a filler and the gap that it licenses; in particular, can we simply list a finite set of strings that separate the two? The answer is No: there is no upper bound on the distance between filler and gap. This fact can be easily illustrated with constructions involving sentential complements, as shown in (140).

(140a) Who do you like __?

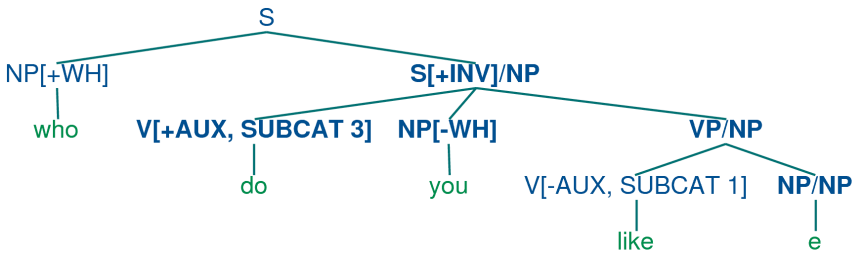
(140b) Who do you claim that you like __?

(140c) Who do you claim that Jody says that you like __?

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

A variety of mechanisms have been suggested for handling unbounded dependencies in formal grammars; we shall adopt an approach due to Generalized Phrase Structure Grammar that involves something called **slash categories**. A slash category is something of the form Y/XP; we interpret this as a phrase of category Y which is somewhere missing a sub-constituent of category XP. For example, S/NP is an S which is missing an NP. The use of slash categories is illustrated in (141).

(141)



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

Do we need to think of slash categories as a completely new kind of object in our grammars? Fortunately, no, we don't — in fact, we can accommodate them within our existing feature-based framework. We do this by treating slash as a feature, and the category to its right as a value. In other words, our 'official' notation for S/NP will be S[SLASH = *NP*]. Once we have taken this step, it is straightforward to write a small grammar in NLTK for analyzing unbounded dependency constructions. (142) illustrates the main principles of slash categories, and also includes rules for inverted clauses. To simplify presentation, we have omitted any specification of tense on the verbs.

(142)

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

```
S[-INV] -> NP  S/NP
S[-INV]/?x -> NP VP/?x
S[+INV]/?x -> V[+AUX] NP VP/?x
S-BAR/?x -> Comp S[-INV]/?x

NP/NP ->

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

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

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

Comp -> 'that'
```

(142) contains one gap-introduction rule, namely

(143) `S[-INV] → NP S/NP`

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

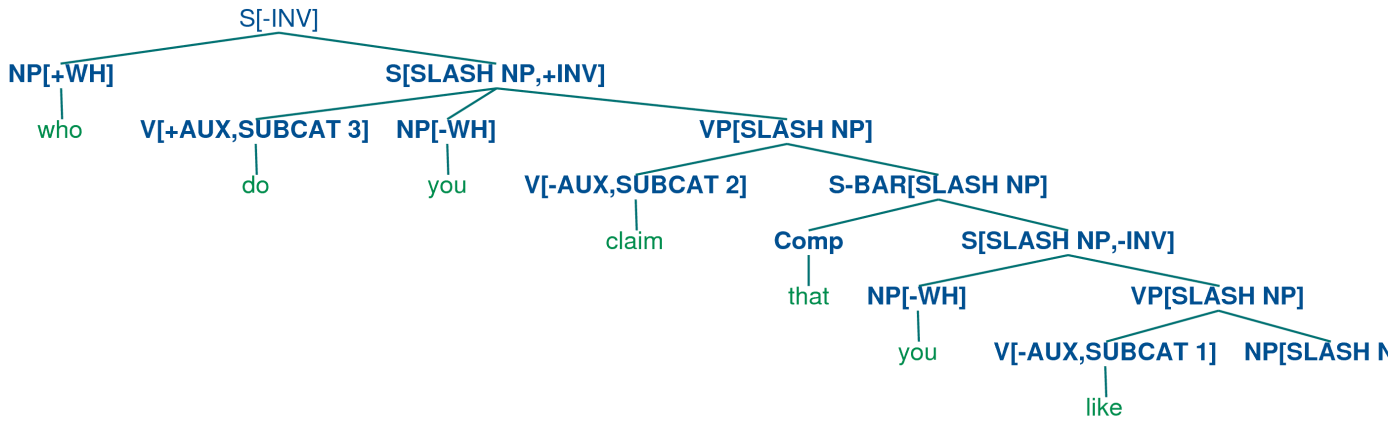
(144) `VP/?X → V S-BAR/?X`

says that a slash value can be specified on the VP mother of a constituent if the same value is also specified on the S-BAR daughter. Finally, (145) allows the slash information on NP to be discharged as the empty string.

(145) `NP/NP →`

Using (142), we can parse the string *who do you claim that you like* into the tree shown in (146).

(146)



9.4.5 Case and Gender in German

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

Case	Masc	Fem	Neut	Plural
Nom	der	die	das	die
Acc	dem	die	das	die
Dat	den	der	dem	den
Gen	des	der	des	der

Table 9.2: Morphological Paradigm for the German definite Article

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

(147a)	Die the.NOM.FEM.SG 'the cat sees the dog'	katze cat.3.FEM.SG	sieht see.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG
(147b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	sieht see.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
(148a)	Die the.NOM.FEM.SG 'the cat helps the dog'	katze cat.3.FEM.SG	hilft help.3.SG	den the.DAT.MASC.SG	hund dog.3.MASC.SG
(148b)	*Die the.NOM.FEM.SG	katze cat.3.FEM.SG	hilft help.3.SG	dem the.ACC.MASC.SG	hund dog.3.MASC.SG

The grammar (149) illustrates the interaction of agreement (comprising person, number and gender) with case.

(149)

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

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

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

###############
# Lexical Rules
###############
```

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

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

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

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

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

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

9.4.6 Exercises

1. ☼ Modify the grammar illustrated in (118) to incorporate a BAR feature for dealing with phrasal projections.

2. ☼ Modify the German grammar in (149) to incorporate the treatment of subcategorization presented in 9.4.1.

3. ◑ Extend the German grammar in (149) so that it can handle so-called verb-second structures like the following:

   (150) Heute sieht der hund die katze.

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

5. ★ So-called **head features** are shared between the mother and head daughter. For example, TENSE is a head feature that is shared between a VP and its head V daughter. See

[Gazdar et al., 1985] for more details. Most of the features we have looked at are head features — exceptions are SUBCAT and SLASH. Since the sharing of head features is predictable, it should not need to be stated explicitly in the grammar rules. Develop an approach which automatically accounts for this regular behaviour of head features.

9.5 Summary

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

9.6 Further Reading

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

Coming more from the perspective of computational linguistics, [Kay, 1985] proposed that functional aspects of language could be captured by unification of attribute-value structures, and a similar approach was elaborated by [Shieber et al., 1983] within the PATR-II formalism. Early work in Lexical-Functional grammar (LFG; [Kaplan and Bresnan, 1982]) introduced the notion of an **f-structure** which was primarily intended to represent the grammatical relations and predicate-argument structure associated with a constituent structure parse. [Shieber, 1986] provides an excellent introduction to this phase of research into feature-based grammars.

One conceptual difficulty with algebraic approaches to feature structures arose when researchers attempted to model negation. An alternative perspective, pioneered by [Kasper and 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. [Kaplan, 1989], and was also adopted by later versions of Head-Driven Phrase Structure Grammar (HPSG; [Sag and Wasow, 1999]).

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

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

There is a copious literature on the analysis of German within feature-based grammar frameworks. [Nerbonne et al., 1994] is a good starting point for the HPSG literature on this topic, while [Müller, 1999] gives a very extensive and detailed analysis of German syntax in HPSG.