7. Grammars and Parsing

7.1 Introduction

Early experiences with the kind of grammar taught in school are often confusing. Written work may be graded by a teacher who red-lines all the grammar errors they won't put up with. Like the dangling preposition in the last sentence, or sentences like this one which lack a main verb. Learning English as a second language, it may be difficult to discover which of these errors need to be fixed (or *needs* to be fixed?). Correct punctuation is something of an obsession for many writers and editors (as our own students have observed). Of course, it is all in the name of effective communication. In the following example, the interpretation of a relative clause as restrictive or non-restrictive depends on the presence of commas alone:

(1a) The presidential candidate, who was extremely popular, smiled broadly.

(1b) The presidential candidate who was extremely popular smiled broadly.

In (1a), we assume there is just one presidential candidate, and say two things about her: that she was popular and that she smiled. In (1b), 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. Others seem to be vestiges of antiquated style that preoccupies only the most crusty curmudgeons. As an example, 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 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" [languagelog.org].

When reading, we sometimes find that we have to stop and re-read a sentence in order to resolve an ambiguity. Curiously, it seems possible to combine *unambiguous* words to create ambiguous sentences:

(2a) Fighting animals could be dangerous.

(2b) Visiting relatives can be tiresome.

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:

(3a) The farmer *loaded* the cart with sand

(3b) 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.)

(4a) The farmer *filled* the cart with sand

(4b) *The farmer *filled* sand into the cart

(4c) *The farmer *dumped* the cart with sand

(4d) The farmer *dumped* sand into the cart

A further curious fact is that we are able to access the meaning of sentences we have not encountered. 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.

7.2 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.

7.2.1 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. We can construct simple examples of syntactic ambiguity involving coordinating conjunctions like *and* and *or*. Consider the following sentence:

(5) Kim left or Dana arrived and everyone cheered.

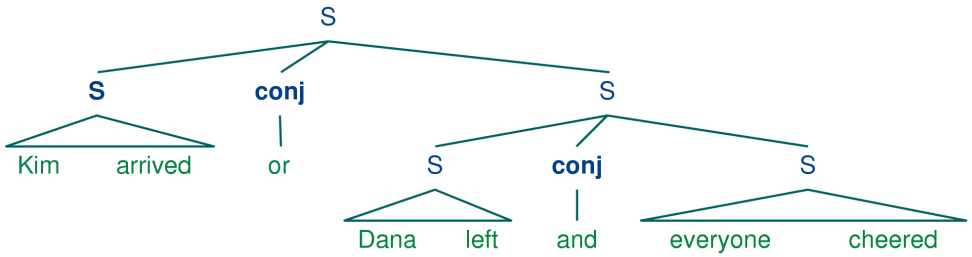
It should be obvious to you that there are two distinct interpretations of this sentence. How should we account for the difference? If you are familiar with propositional logic, you will not be surprised at the idea of using brackets to represent semantic structure:

(6a) Kim arrived or (Dana left and everyone cheered)

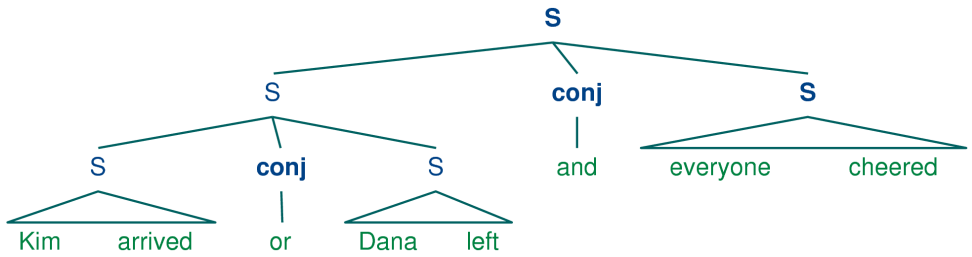
(6b) (Kim arrived or Dana left) and everyone cheered

We can describe this ambiguity in terms of the semantic **scope** of *or* and *and*: in the reading represented by (6a), the operator *or* takes the conjoined sentence *Dana arrived and everyone cheered* as one of its arguments, and therefore is said to have wider scope than *and*. Conversely, in (6b), the operator *and* has wider scope than *or*. One convenient way of representing this scope difference at a structural level is by means of a **tree diagram**.

(7a)



(7b)



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-Lite, you can easily produce trees like this yourself with the following commands:

```
>>> from nltk_lite.parse import bracket_parse
>>> sent = '(S (S Kim arrived) (conj or) (S Dana left))'
>>> tree = bracket_parse(sent)
>>> tree.draw()
```

Conveniently, the resulting tree object supports Python's standard array operations for accessing its children:

```
>>> tree[0]
(S: 'Kim' 'arrived')
```

A second example of scope ambiguity involves adjectives: *old men and women*. Does *old* have wider scope than *and*, or is it the other way round? In fact, both interpretations are possible.

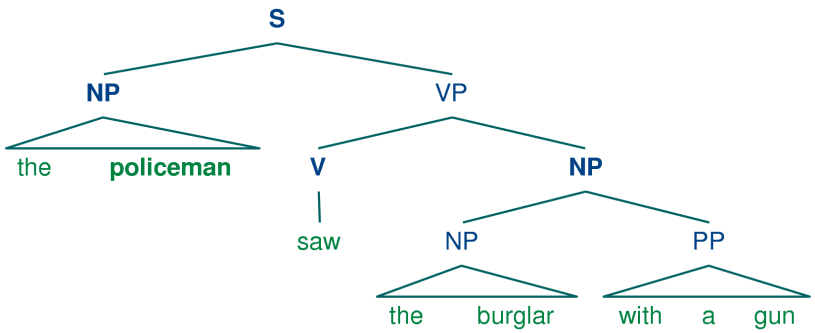
For our third illustration of ambiguity, we look at prepositional phrases. Consider a sentence like: *I saw the man with a telescope*. Who has the telescope? To clarify what is going on here, consider the following pair of sentences:

(8a) The policeman saw a burglar *with a gun*. (not some other burglar)

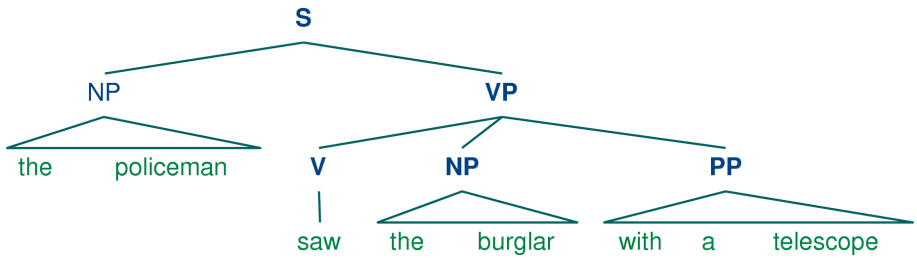
(8b) The policeman saw a burglar *with a telescope*. (not with his naked eye)

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? Again, we can represent the difference in terms of tree structure:

(9a)



(9b)



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 see that they are trees over the same sequence of words (that is, the two trees have the same **leaves**):

```
>>> tree1.leaves() == tree2.leaves
True
```

On the other hand, they 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.height() == tree2.height()
False
```

In general, how can we determine whether a prepositional phrase modifies the preceding noun or verb? This problem is often described with the label **PP attachment**. The **Prepositional Phrase Attachment Corpus**, included with NLTK-Lite as *ppattach*, 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 following sentence from the WSJ:

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

The corresponding line in the *ppattach* corpus is this:

(11)
```
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:

(12)
```
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 attachments in the above examples can also be made explicit by using phrase groupings as follows:

(13)
```
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 this corpus from NLTK-Lite as follows:

```
>>> from nltk_lite.corpora import ppattach, extract
>>> from pprint import pprint
>>> item = extract(16, ppattach.dictionary('devset'))
>>> 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 in to play. The following program uses **`MinimalSet`** to find pairs of entries in the corpus which have different attachments based on the *verb* only.

```
>>> from nltk_lite.utilities import MinimalSet
>>> ms = MinimalSet()
>>> for entry in ppattach.dictionary('training'):
...     target  = entry['attachment']
...     context = (entry['noun1'], entry['prep'], entry['noun2'])
...     display = (target, entry['verb'])
...     ms.add(context, target, display)
>>> for context in ms.contexts():
...     print context, ms.display_all(context)
```

Here is one of the pairs found by the program.

(14)
```
received (NP offer) (PP from group)
rejected (NP offer (PP from group))
```

This finding gives us clues to a structural difference: the verb *receive* usually comes with two following arguments; we receive something *from* someone. In contrast, the verb *reject* only needs a single following argument; we can reject something without needing to say where it originated from. We expect that if you look at the data, you will be able to come up with further ideas about the factors that influence PP attachment.

7.2.2 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

(15) 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:

(16)
```
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:

(17) 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:

(18) `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:

(19) `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:

(20) 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*:

(21a) *Ordinary daily multivitamin and mineral supplements* could help adults with diabetes fight off some minor infections.

(21b) *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 (22a), we have the question in (22b):

(22a) All the cakes have been eaten.

(22b) Have *all the cakes* been eaten?

Roughly speaking, if a sentence already contains an auxiliary verb, such as *has* in (22a), 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:

(23a) The fierce dog bit the man.

(23b) Did *the fierce dog* bite the man?

As we would hope, this test also confirms our earlier claim about the subject constituent of (20):

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

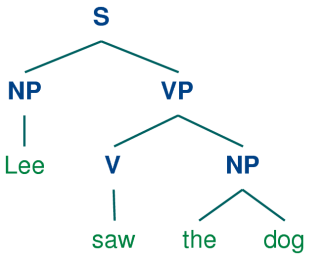
To sum up then, we have seen that the notion of constituent brings a number of benefits. By having a constituent labeled noun phrase, we can provide a unified statement of the classes of word that constitute that phrase, and reuse this statement in describing noun phrases wherever they occur in the sentence. 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.

7.2.3 More on NLTK's Trees

We have been discussing structural differences between sentences, and we have been probing these structures by substituting words and phrases. We have informally shown how sentence structures can be represented using syntactic trees, and we will now look at these structures in a bit more detail.

Consider the following example:

(25)



Terminology: 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**.

Each production in a CFG corresponds to a tree of depth one; we call these *local*:dt:' trees.

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 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[1].

Trees are created with the **`Tree`** constructor, which takes a node value and a list of zero or more children. Here's an example of a simple NLTK-Lite tree with a single child node, where the latter is a token:

```
>>> from nltk_lite.parse.tree import Tree
>>> tree1 = Tree('NP', ['John'])
>>> tree1
(NP: 'John')
```

Here is an example with two children:

```
>>> tree2 = Tree('NP', ['the', 'man'])
>>> tree2
(NP: 'the' 'man')
```

Finally, here is a more complex example, where one of the children is itself a tree:

```
>>> tree3 = Tree('VP', ['saw', tree2])
>>> tree3
(VP: 'saw' (NP: 'the' 'man'))
```

A tree's root node value is accessed with the **`node`** property, and its leaves are accessed with the **`leaves()`** method:

```
>>> tree3.node
'VP'
>>> tree3.leaves()
['saw', 'the', 'man']
```

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. Although we cannot access subjects directly, in practice we can get something similar by using **tree positions**. Consider **`tree4`** defined as follows:

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

Now we can just use indexing to access the subtrees of this tree:

```
>>> tree4[0]
(NP: 'John')
>>> tree4[1]
(VP: 'saw' (NP: 'the' 'man'))
```

Since the value of **`tree4[1]`** is itself a tree, we can index into that as well:

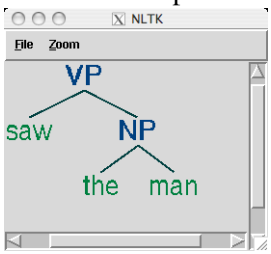
```
>>> tree4[1][0]
'saw'
>>> tree4[1][1]
(NP: 'the' 'man')
```

[1] Although the **`Tree`** class is usually used for encoding syntax trees, it can be used to encode *any* homogeneous hierarchical structure that spans a text (such as morphological structure or discourse structure). In the general case, leaves and node values do not have to be strings.

The printed representation for complex trees can be difficult to read. In these cases, the **draw** method can be very useful.

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

This method 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).

To compare multiple trees in a single window, we can use the **draw_trees()** method:

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

The **Tree** class implements a number of other useful methods. See the **Tree** reference documentation for more information about these methods.

The **nltk_lite.corpora** module defines the **treebank** corpus, which contains a collection of hand-annotated parse trees for English text, derived from the Penn Treebank.

```
>>> 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'))))
  (.: '.'))
```

7.2.4 Exercises

1. 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*.

2. 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.

7.3 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 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 VB → `'help'` means that the constituent VB can consist of the string *help*. 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.

7.3.1 A Simple Grammar

Let's start off by looking at a simple context-free grammar:

(26) S → NP VP
NP → Det N PP
NP → Det N
VP → V NP PP
VP → V NP
VP → V
PP → P NP

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

This grammar contains productions involving various syntactic categories, as laid out in the following table:

Table 1: Syntactic Categories

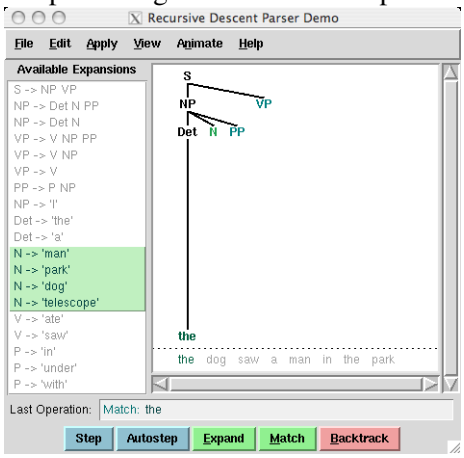
Symbol	Meaning	Example
S	sentence	*the man walked*
NP	noun phrase	*a dog*
VP	verb phrase	*saw a park*
PP	prepositional phrase	*with a telescope*
Det	determiner	
N	noun	
V	verb	
P	preposition	

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.

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, which is invoked as follows:

```
>>> from nltk_lite.draw import rdparser
>>> rdparser.demo()
```

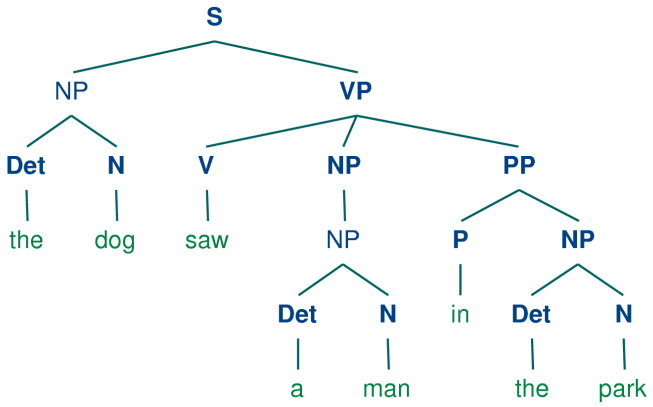
The demo opens a window which displays a list of grammar rules in the lefthand pane and the current parse diagram in the central pane:



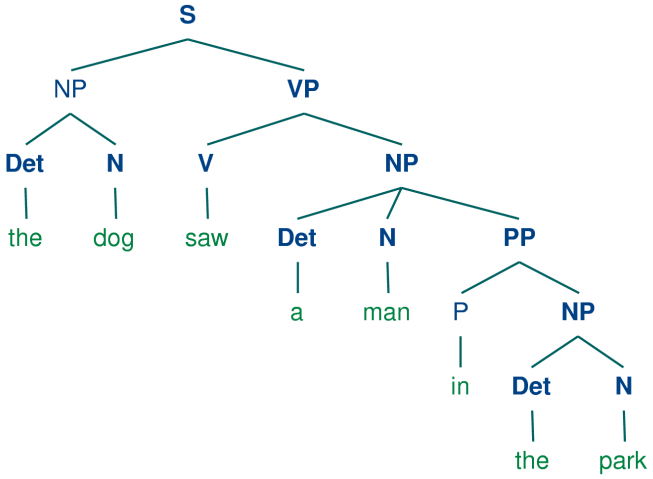
The demo comes with the grammar in (26) already loaded. We will discuss the parsing algorithm in greater detail below, but for the time being you can get a rough idea of how it works by using the *autostep* button.

If we parse the string *The dog saw a man in the park* using the grammar in (26), we end up with two trees:

(27a)



(27b)



Since our grammar assigns two distinct structures, the sentence is said to be **structurally ambiguous**. The ambiguity in question is called a **PP 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.

As we noted earlier, there is also a difference in interpretation: where the PP is attached to VP, the intended interpretation is that the event of seeing took place in the park, while if the PP is attached to NP, being in the park is a property of the NP referent; that is, the man was in the park, but the agent of the seeing — the dog — might have been somewhere else (e.g., sitting on the balcony of an apartment overlooking the park). As we will see, dealing with ambiguity is a key challenge in parsing.

7.3.2 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 (26) 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 rule, namely `NP -> Det N PP`, to `NP -> NP PP`. Using the *Step* button, try to build a parse tree. What happens?

7.3.3 Recursion

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

(28a) Jodie won the 100m freestyle

(28b) 'The Age' reported that Jodie won the 100m freestyle

(28c) Sandy said 'The Age' reported that Jodie won the 100m freestyle

(28d) 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 (26). However, grammar (29) below illustrates both kinds of recursive rule:

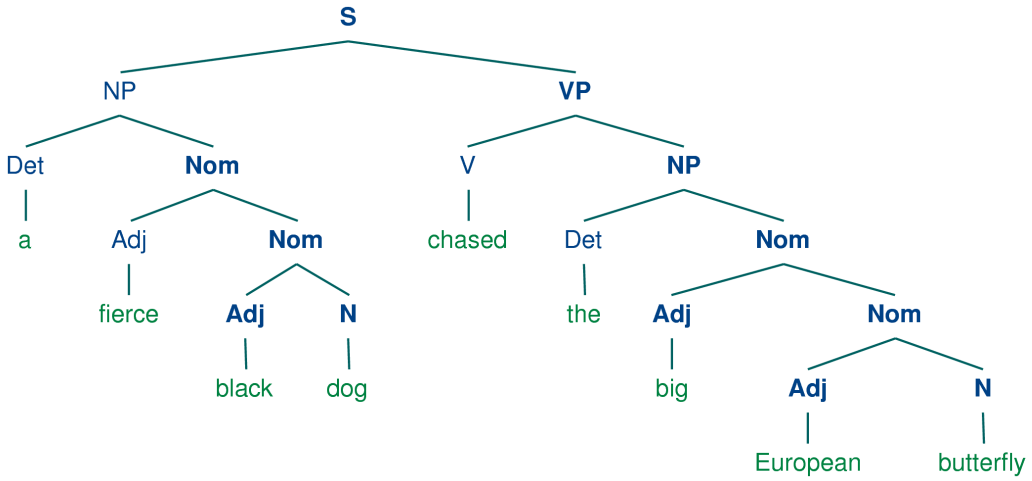
(29) S → NP VP
NP → Det Nom
NP → Det Nom PP
NP → PropN
Nom → Adj Nom
Nom → N
VP → V NP PP
VP → V NP
VP → V S
VP → V
PP → P NP

PropN → 'John' | 'Mary'
Det → 'the'
Det → '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.

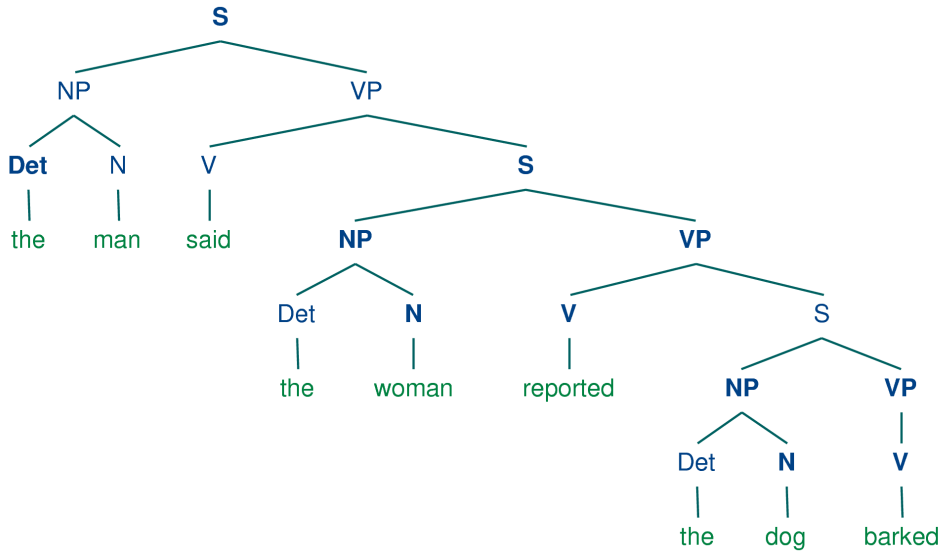
To illustrate recursion in this grammar, we show first of all a tree involving nested nominal phrases:

(30)



Next, observe how we can embed one S constituent into another:

(31)

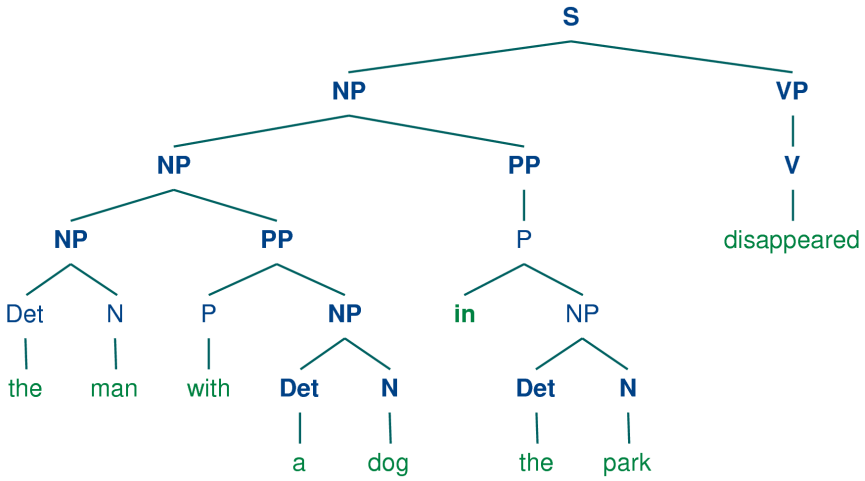


If you did the exercises for the last section, you will have noticed that the recursive descent parser fails to deal properly with the following rule:

(32) `NP → NP PP`

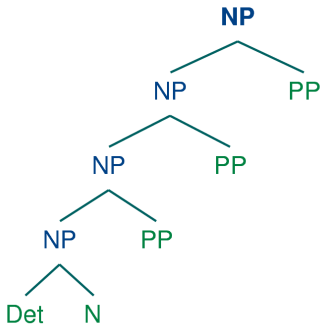
From a linguistic point of view, this rule is perfectly respectable, and will allow us to derive trees like this:

(33)



More schematically, the trees will be of the following shape:

(34)



(34) is an example of a **left recursive** structure. Such structures seem to occur rather 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, as we will discuss later in this chapter.

7.3.4 Heads, Complements and Modifiers

Let us take a closer look at verbs. The grammar (29) correctly generates examples like (35), corresponding to the four productions with VP on the lefthand side:

(35a) The woman gave the telescope to the dog.

(35b) The woman saw a man.

(35c) A man said that the woman disappeared.

(35d) 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[2]:

(36a) *The woman disappeared the telescope to the dog.

(36b) *The dog barked a man.

(36c) *A man gave that the woman disappeared.

(36d) *A man said.

How can we ensure that our grammar correctly excludes the ungrammatical examples in (36)? 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:

(37)
```
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. The following table provides more examples of labels for verb subcategories.

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*

The revised grammar for VP will now look like this:

(38)
```
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 (38), a given lexical item can belong to more than one subcategory. For example, *sang* can occur both with and without a following NP complement.

[2] 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.

7.3.5 Lexical Acquisition

We have seen increasingly detailed grammars, e.g., identifying different kinds of verbs. How are we to acquire this information in a scalable way? One method is to return to the chunking methods. For example, we saw in the Chunking chapter that it is possible to collapse chunks down to the chunk label, thus:

(39)
```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

We can use this as raw material to guide us as we manually construct more grammar productions.

7.3.6 Review of CFGs

We have seen that a CFG contains terminal and nonterminal symbols, and rules 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 $\langle N, \Sigma, P, S \rangle$, 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 $(N \cup \Sigma)$* (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, (40) is a derivation of *the dog with a telescope* for the grammar in (26).

(40)
```
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 (40) could equally have started off in the following manner:

(41)
```
NP
Det N PP
Det N P NP
Det N with NP
...
```

We can also write derivation (40) as:

(42) 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.

7.3.7 Context Free Grammars in NLTK-Lite

Context free grammars are encoded by the **`cfg.Grammar`** class. The easiest way to construct a grammar object is from the standard string representation of grammars:

```
>>> productions = '''
... 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"
... PP -> P NP
... P -> "on" | "by" | "with"
... '''
```

Now we can convert this string into a grammar object:

```
>>> from nltk_lite import parse
>>> grammar = parse.cfg.parse_grammar(productions)
>>> grammar
<Grammar with 21 productions>
```

Next, we can build a parser for this grammar:

```
>>> from nltk_lite import parse
>>> rd_parser = parse.RecursiveDescent(grammar)
```

Finally, we can use the parser to parse a sentence:

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

7.3.8 Exercises

1. Extend the grammar in (29) 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*.

2. Pick some common verbs.

   a) Write a program to find those verbs in the PP Attachment Corpus included with NLTK-Lite. 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.

3. **Lexical Acquisition:** 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.

7.4 Parsing

A **parser** is a computational system which processes input sentences according to the productions of a grammar, and builds one or more constituent structures which conform to the grammar. While a grammar is a declarative specification of well-formedness, a parser is a procedural interpretation of the grammar. We can think of the parser as searching through the space of possible trees licensed by a grammar, to find one that has the required sentence along its fringe. Following on from our description of context free grammars, we will now describe some simple parsers that work with them.

Parsing is important in linguistics and natural language processing for a variety of reasons. A parser permits a grammar to be evaluated against a potentially large collection of test sentences, helping the linguist to identify shortcomings in their analysis. A parser can also be used as a model of psycholinguistic processing, with the goal of explaining the processing difficulties that humans have with certain syntactic constructions (e.g., the so-called 'garden path' sentences). There are many NL applications which 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.

7.4.1 The Parser Interface

The `parse` module defines the `ParseI` interface, which in turn defines the two methods which all parsers should support:

1. The `parse` method returns the single best parse for a given text. The text is represented as a list of word tokens. If no parses are found for the given text, then `parse` returns `None`.

2. The `get_parse_list` method returns a list of the parses for the given text.

For example, here is what the recursive descent parser generates for a simple sentence and grammar:

```
>>> from nltk_lite import tokenize, parse
>>> sent = list(tokenize.whitespace('I saw a man in the park'))
>>> rd_parser = parse.RecursiveDescent(grammar)

>>> for p in rd_parser.get_parse_list(sent):
...     print p
(S:
  (NP: 'I')
  (VP:
    (V: 'saw')
    (NP:
      (Det: 'a')
      (N: 'man')
      (PP: (P: 'in') (NP: (Det: 'the') (N: 'park'))))))
(S:
  (NP: 'I')
  (VP:
    (V: 'saw')
    (NP: (Det: 'a') (N: 'man'))
    (PP: (P: 'in') (NP: (Det: 'the') (N: 'park')))))
```

7.4.2 Recursive Descent Parsing

The simplest kind of parser interprets the 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.

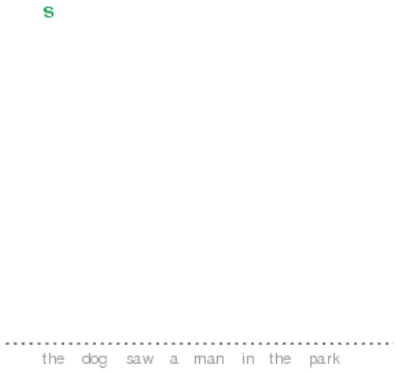
The recursive descent parser builds a parse tree during the above process. With the initial goal (find an *S*), the *S* root node is created. As the above process recursively expands its goals using the productions of the grammar, the parse tree is extended downwards (hence the name *recursive descent*). We can see this in action using the parser demonstration **`nltk_lite.draw.rdparser`**. To run this demonstration, use the following commands:

```
>>> from nltk_lite.draw import rdparser
>>> rdparser.demo()
```

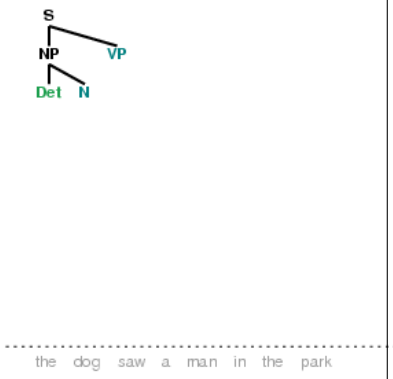
Six stages of the execution of this parser are shown below:

Six Stages of a Recursive Descent Parser

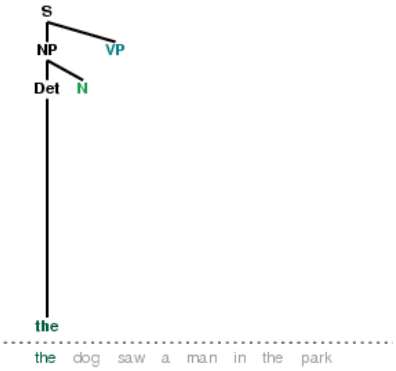
a. Initial stage



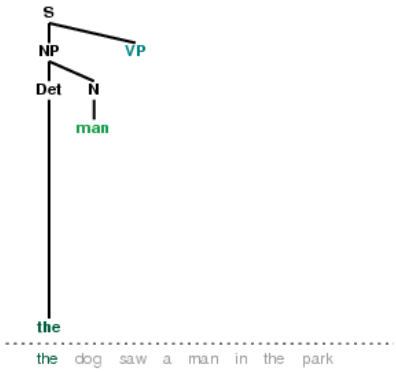
b. After two productions



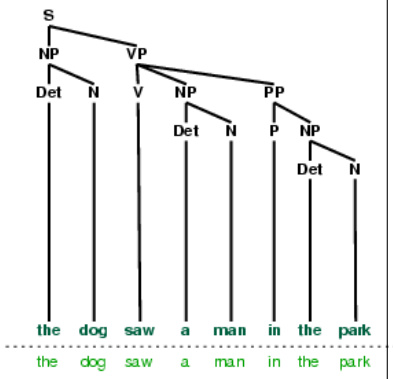
c. After matching "the"



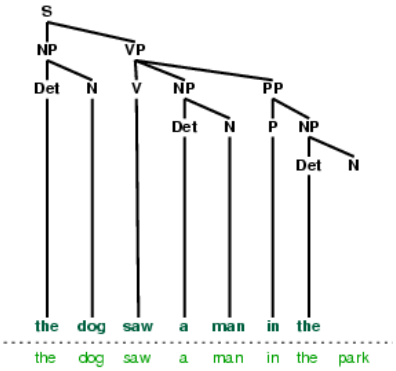
d. Failing to match "man"



e. Completed parse



f. Backtracking



During this process, the parser sometimes must 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.

7.4.3 The Recursive Descent Parser in NLTK

The `nltk_lite.parse` module defines `RecursiveDescent`, a simple recursive implementation of a top-down parser. Recursive descent parsers are created from `Grammars` by the `RecursiveDescent` constructor.

```
>>> from nltk_lite import parse
```

```
>>> rd_parser = parse.RecursiveDescent(grammar)
>>> sent = list(tokenize.whitespace('I saw a man'))
>>> rd_parser.get_parse_list(sent)
[(S: (NP: 'I') (VP: (V: 'saw') (NP: (Det: 'a') (N: 'man'))))]
```

The constructor takes an optional parameter **trace**. If **trace** is greater than zero, then the parser will describe the steps that it takes as it parses a text.

7.4.4 Problems with Recursive Descent Parsing

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.

7.4.5 Shift-Reduce Parsing

The simplest kind of bottom-up parsing is known as **shift-reduce parsing**. 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.

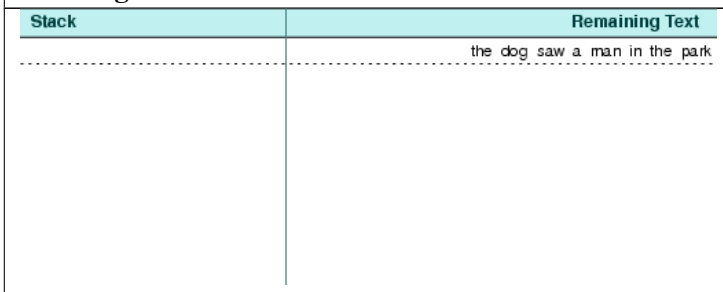
The shift-reduce parser repeatedly pushes the next input word onto a stack; this is the **shift** operation. If the top *n* items on the stack match the *n* items on the right-hand side of some production, then they are all popped off the stack, and the item on the left-hand side of the production is 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**. To run this demonstration, use the following commands:

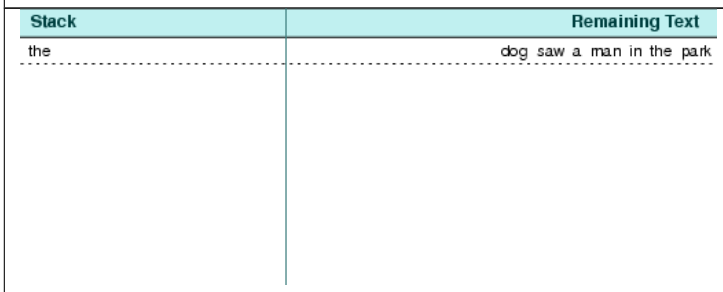
```
>>> from nltk_lite.draw import srparser
>>> srparser.demo()
```

Six stages of the execution of this parser are shown below:

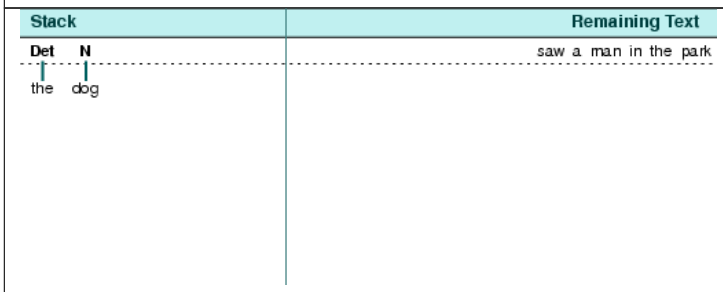
Six Stages of a Shift-Reduce Parser



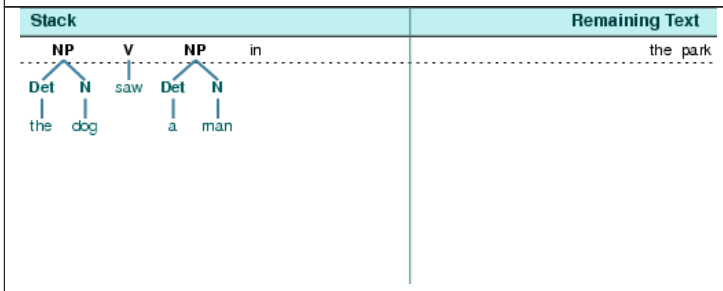
a. Initial State



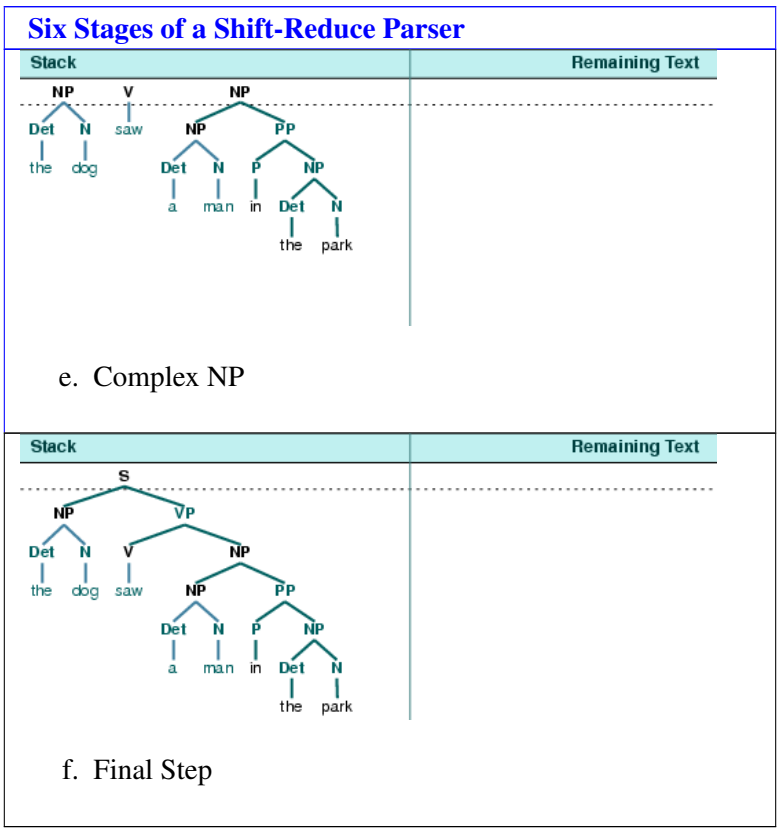
b. After one shift



c. After shift reduce shift



d. After recognizing the second NP



7.4.6 The Shift Reduce Parser in NLTK

The **`nltk_lite.parse`** module defines **`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.

Shift reduce parsers are created from **`Grammars`** by the **`ShiftReduceParse`** constructor. The constructor takes an optional parameter **`trace`**. As with the recursive descent parser, this value specifies how verbosely the parser should describe the steps that it takes as it parses a text:

```
>>> sr_parse = parse.ShiftReduce(grammar, trace=1)
```

The following example shows the trace output generated by **`sr_parser`** on a simple sentence:

```
>>> sent = list(tokenize.whitespace('I saw a man'))
>>> sr_parse.parse(sent)
Parsing 'I saw a man'
    [ * I saw a man]
  S [ 'I' * saw a man]
  R [ <NP> * saw a man]
  S [ <NP> 'saw' * a man]
  R [ <NP> <V> * a man]
  S [ <NP> <V> 'a' * man]
  R [ <NP> <V> <Det> * man]
  S [ <NP> <V> <Det> 'man' * ]
  R [ <NP> <V> <Det> <N> * ]
```

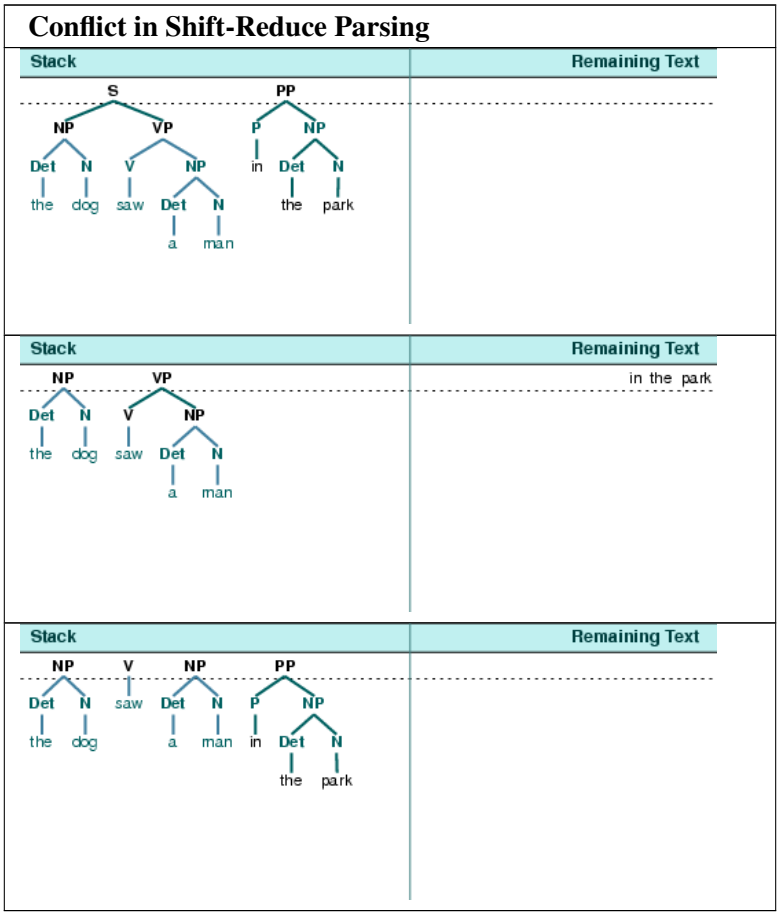
```
    R [ <NP> <V> <NP> * ]
    R [ <NP> <VP> * ]
    R [ <S> * ]
(S: (NP: 'I') (VP: (V: 'saw') (NP: (Det: 'a') (N: 'man'))))
```

NLTK also defines a graphical demonstration tool for the shift reduce parser:

```
>>> from nltk.draw.srparser import demo
>>> demo()
```

7.4.7 Problems with Shift Reduce Parser

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 the left example below. The parser entered this blind alley at an earlier stage shown in the middle example below, when it reduced instead of shifted. This situation is called a **shift-reduce conflict**. At another possible stage of processing shown in the right example below, the parser must choose between two possible reductions, both matching the top items on the stack: V → V NP PP or NP → NP PP. This situation is called a **reduce-reduce conflict**.



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.

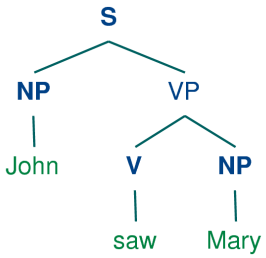
The advantages of shift-reduce parsers over recursive descent parsers is that they only build structure that corresponds to the words in the input. Furthermore, they only build each sub-structure once, e.g. `NP(Det(the), N(man))` is only built and pushed onto the stack a single time, regardless of whether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

7.4.8 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 (29) allows us to produce the following parse of *John saw Mary*:

(43)



Recall that the grammar in (29) has the following rules for expanding NP:

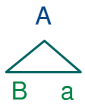
(44a) NP → Det Nom

(44b) NP → Det Nom PP

(44c) NP → PropN

Suppose we ask you to first look at tree (43), and then decide which of the NP rules you'd want a recursive descent parser to apply first — obviously, (44c) is the right choice! How do you know that it would be pointless to apply (44a) or (44b) instead? Because neither of these rules 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 $A \Rightarrow^* B\ \alpha$.

(45)



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 lc illustrates this for the grammar from (29).

Table 6: Left-Corners in (29)

Category	Left-Corners
S	NP
NP	Det, PropN
VP	V
PP	P

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.

7.4.9 Exercises

1. **Left-corner parser:** Develop a left-corner parser (inheriting from **`ParseI`**), based on the recursive descent parser.

2. Compare the performance of the top-down, bottom-up, and left-corner parsers using the same grammar and three grammatical test sentences. Use **`time.time()`** to log the amount of time each parser takes on the same sentence. 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. 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**).

7.5 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 the next chapter), 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/).

7.6 Further Reading

McCawley (1998) *The Syntactic Phenomena of English*. Chicago University Press.

Rodney D. Huddleston, Geoffrey K. Pullum (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.