

Chapter 7

Context Free 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 that 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:

- (1) a. The presidential candidate, who was extremely popular, smiled broadly.
- b. 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 which of several possible 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 & 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 (2b).

- (2) a. Fighting animals could be dangerous.
b. 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:

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

- (4) a. The farmer *filled* the cart with sand
b. *The farmer *filled* sand into the cart
c. *The farmer *dumped* the cart with sand
d. 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 modeling 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.

Note

Remember that our program samples assume you begin your interactive session or your program with: `import nltk, re, pprint`

7.3 What's the Use of Syntax?

Earlier chapters focused on words: how to identify them, how to analyze 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 that 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 that are much easier to capture once we have established a level of syntactic structure.

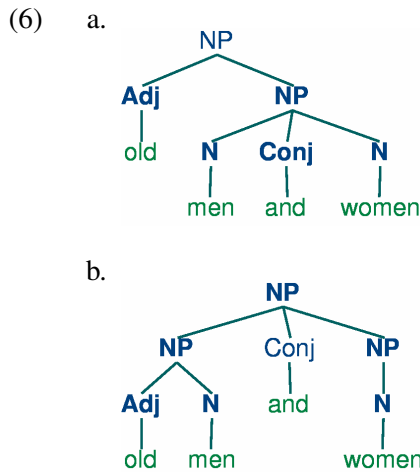
7.3.1 Syntactic Ambiguity

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:

- (5) a. old (men and women)
- b. (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 (6b).



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:

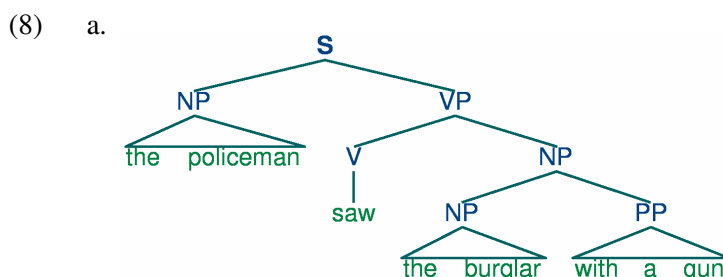
```
>>> tree = nltk.bracket_parse('(NP (Adj old) (NP (N men) (Conj and) (N women)))')
>>> tree.draw()
```

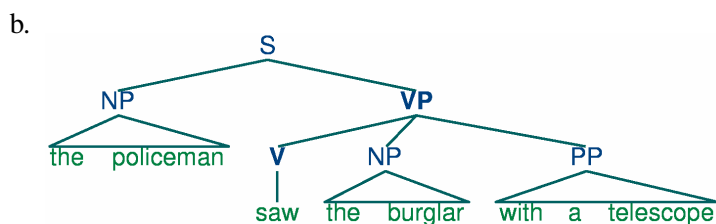
We can construct other examples of syntactic ambiguity involving the coordinating conjunctions *and* and *or*, e.g. *Kim left or Dana arrived and everyone cheered*. We can describe this ambiguity in terms of the relative semantic **scope** of *or* and *and*.

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:

- (7) a. The policeman saw a burglar *with a gun*. (not some other burglar)
 b. 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:





In (8b)a, the PP attaches to the NP, while in (8b)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 = nltk.bracket_parse(s1)
>>> tree2 = nltk.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 (except for the last word). 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()[:-1] == tree2.leaves()[:-1]
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 (9a). The corresponding line in the Prepositional Phrase Attachment Corpus is shown in (9b).

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

- (10) 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 (10) can also be made explicit by using phrase groupings as in (11).

- (11) 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:

```
>>> nltk.corpus.ppattach.tuples('training') [9]
('16', 'including', 'three', 'with', 'cancer', 'N')
```

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 (12).

- (12) 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 6):

- (13) 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:

- (14) 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:

- (15) 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:

- (16) 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 esthetic: 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:

(17) 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 italicized string in the following example is a constituent, since it can be replaced by *they*:

- (18) a. *Ordinary daily multivitamin and mineral supplements* could help adults with diabetes fight off some minor infections.
- b. *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 (19a), we have the question in (19b):

- (19) a. All the cakes have been eaten.
- b. Have *all the cakes* been eaten?

Roughly speaking, if a sentence already contains an auxiliary verb, such as *has* in (19a), 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:

- (20) a. The fierce dog bit the man.
- b. Did *the fierce dog* bite the man?

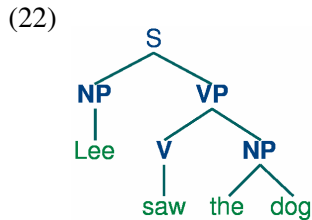
As we would hope, this test also confirms our earlier claim about the subject constituent of (17):

(21) 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:



Although it is helpful to represent trees in a graphical format, for computational purposes we usually need a more text-oriented representation. We will use the same format as the Penn Treebank, a combination of brackets and labels:

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

Here, the node value is a constituent type (e.g., NP or VP), and 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:

```
>>> tree1 = nltk.Tree('NP', ['John'])
>>> print tree1
(NP John)
>>> tree2 = nltk.Tree('NP', ['the', 'man'])
>>> print tree2
(NP the man)
```

We can incorporate these into successively larger trees as follows:

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

Here are some of the methods available for tree objects:

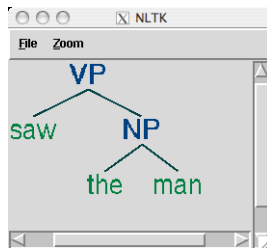
```
>>> print tree4[1]
(VP saw (NP the man))
>>> tree4[1].node
```



```
'VP'
>>> tree4.leaves()
['John', 'saw', 'the', 'man']
>>> tree4[1,1,1]
'man'
```

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

```
>>> print nltk.corpus.treebank.parsed_sents('wsj_0001.mrg')[0]
(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))))
  (. .))
```

Listing 7.1 prints a tree object using whitespace formatting.

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.

```
>>> nltk.corpus.sinica_treebank.parsed_sents()[3450].draw()
```

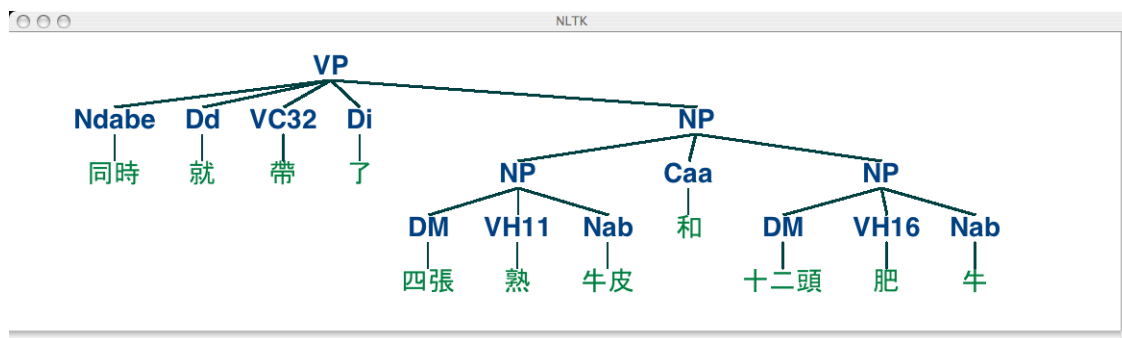
Listing 7.1

```

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 = nltk.corpus.treebank.parsed_sents('wsj_0001.mrg')[0]
>>> 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
      .
      .

```



(23)

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

```
>>> print nltk.corpus.treebank.tagged_sents('wsj_0001.mrg')[0]
[('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

- ✧ Can you come up with grammatical sentences that have probably never been uttered before? (Take turns with a partner.) What does this tell you about human language?
- ✧ 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?
- ✧ 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.
- ✧ 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)`.
- ✧ **Building trees:**
 - Write code to produce two trees, one for each reading of the phrase *old men and women*
 - Encode any of the trees presented in this chapter as a labeled bracketing and use `nltk.bracket_parse()` to check that it is well-formed. Now use `draw()` to display the tree.
 - As in (a) above, draw a tree for *The woman saw a man last Thursday*.
- ✧ 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.)
- ✧ 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.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 that 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 $\{a, aa, aaa, aaaa, \dots\}$). 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 that specifies which sequences of words are well-formed in the language, and that 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 $S \rightarrow NP VP$. This says that a constituent S can consist of sub-constituents NP and VP . Similarly, the production $V \rightarrow 'saw' \mid 'walked'$ 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.

(24) $S \rightarrow NP VP$
 $NP \rightarrow Det N \mid Det N PP$
 $VP \rightarrow V \mid V NP \mid V NP PP$
 $PP \rightarrow P NP$

$Det \rightarrow 'the' \mid 'a'$
 $N \rightarrow 'man' \mid 'park' \mid 'dog' \mid 'telescope'$
 $V \rightarrow 'saw' \mid 'walked'$
 $P \rightarrow 'in' \mid 'with'$

This grammar contains productions involving various syntactic categories, as laid out in [Table 7.1](#).

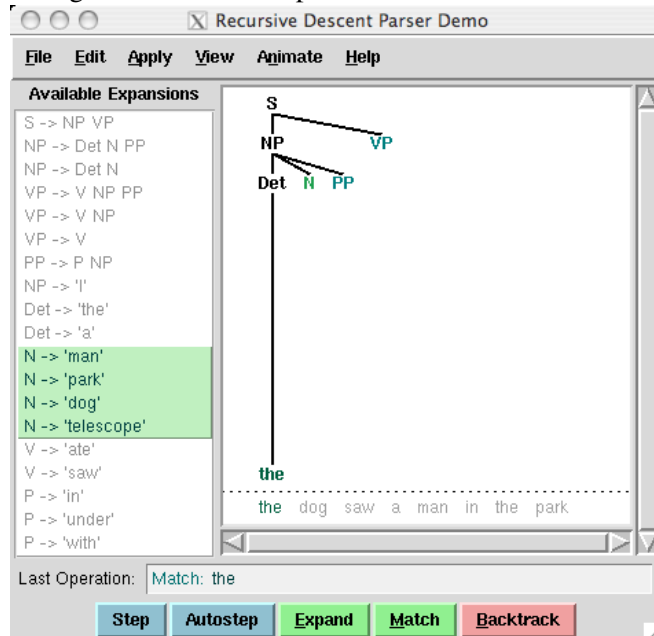
Symbol	Meaning	Example
S	sentence	<i>the man walked</i>
NP	noun phrase	<i>a dog</i>
VP	verb phrase	<i>saw a park</i>
PP	prepositional phrase	<i>with a telescope</i>
...
Det	determiner	<i>the</i>
N	noun	<i>dog</i>
V	verb	<i>walked</i>
P	preposition	<i>in</i>

Table 7.1: Syntactic Categories

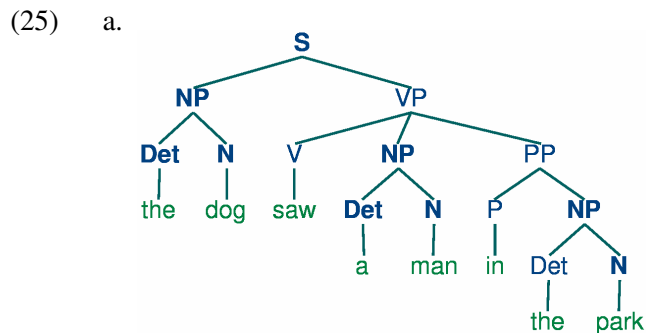
In our following discussion of grammar, we will use the following terminology. The grammar consists of productions, where each production involves a single **non-terminal** (e.g. S , NP), an arrow, and one or more non-terminals and **terminals** (e.g. *walked*). The productions are often divided into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

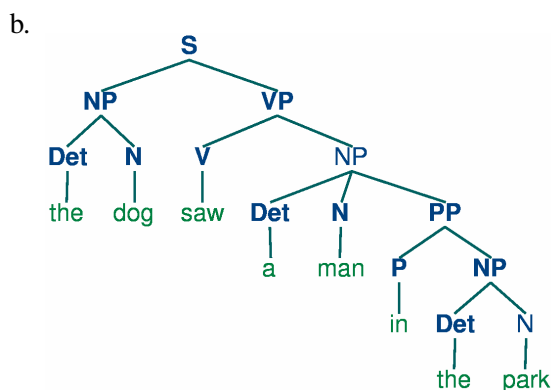
grammar **licenses** a tree if each non-terminal X with children $Y_1 \dots Y_n$ corresponds to a production in the grammar of the form: $X \rightarrow Y_1 \dots Y_n$.

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, `nltk.draw.rdparser.demo()`. The demo opens a window that displays a list of grammar productions in the left hand pane and the current parse diagram in the central pane:



The demo comes with the grammar in (24) 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 (24), we end up with two trees:





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:

- (26)
- a. Jodie won the 100m freestyle
 - b. “The Age” reported that Jodie won the 100m freestyle
 - c. Sandy said “The Age” reported that Jodie won the 100m freestyle
 - d. 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 left hand side of a production (such as S in this case) also appears on the right hand 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 (24). However, the grammar in (27) illustrates both kinds of recursive production:

- (27)
- ```

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

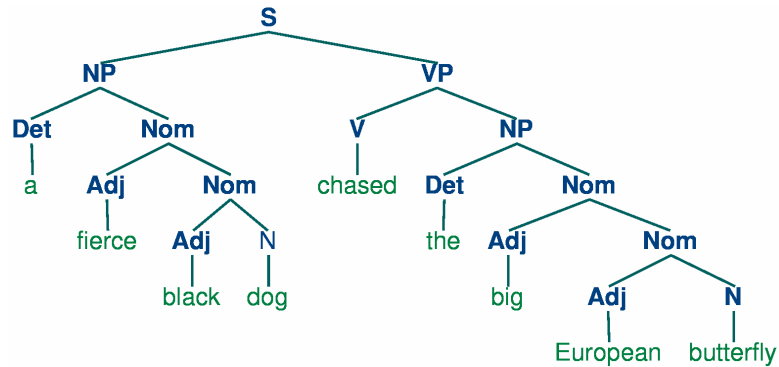
PropN → 'John' | 'Mary'
Det → 'the' | 'a'
N → 'man' | 'woman' | 'park' | 'dog' | 'lead' | 'telescope' | 'butterfly'
Adj → 'fierce' | 'black' | 'big' | 'European'
V → 'saw' | 'chased' | 'barked' | 'disappeared' | 'said' | 'reported'
P → 'in' | 'with'

```

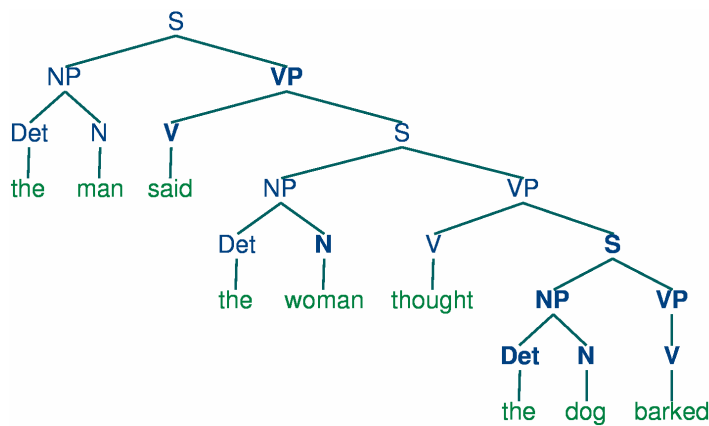
Notice that the production  $NOM \rightarrow ADJ\ NOM$  (where  $NOM$  is the category of nominals) involves direct recursion on the category  $NOM$ , whereas indirect recursion on  $S$  arises from the combination of two productions, namely  $S \rightarrow NP\ VP$  and  $VP \rightarrow V\ S$ .

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.

(28) a.

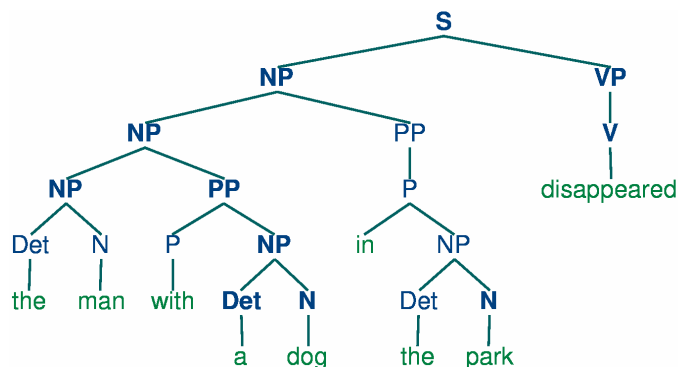


b.



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:  $NP \rightarrow NP\ PP$ . From a linguistic point of view, this production is perfectly respectable, and will allow us to derive trees like this:

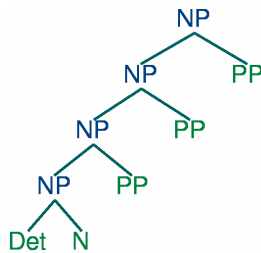
(29)



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



(30)



The structure in (30) 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 (27) correctly generates examples like (31d), corresponding to the four productions with VP on the left hand side:

- (31) a. The woman gave the telescope to the dog  
 b. The woman saw a man  
 c. A man said that the woman disappeared  
 d. 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<sup>1</sup>:

- (32) a. \*The woman disappeared the telescope to the dog  
 b. \*The dog barked a man  
 c. \*A man gave that the woman disappeared  
 d. \*A man said

How can we ensure that our grammar correctly excludes the ungrammatical examples in (32d)? We need some way of constraining grammar productions which expand VP so that verbs *only* co-occur 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

<sup>1</sup>It should be borne in mind that it is possible to create examples that 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.

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:

- (33)  $VP \rightarrow TV\ NP$   
 $TV \rightarrow 'saw' \mid 'kissed' \mid 'hit'$

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 | <i>barked</i>                 |
| TV     | transitive verb   | <i>saw a man</i>              |
| DatV   | dative verb       | <i>gave a dog to a man</i>    |
| SV     | sentential verb   | <i>said that a dog barked</i> |

Table 7.2: Verb Subcategories

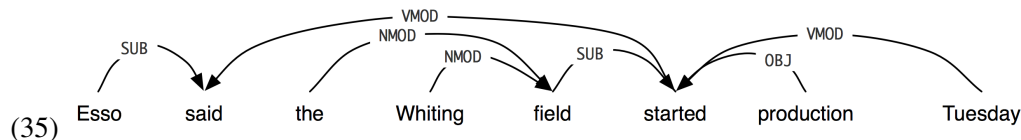
The revised grammar for VP will now look like this:

- (34)  $VP \rightarrow DATV\ NP\ PP$   
 $VP \rightarrow TV\ NP$   
 $VP \rightarrow SV\ S$   
 $VP \rightarrow IV$
- $DATV \rightarrow 'gave' \mid 'donated' \mid 'presented'$   
 $TV \rightarrow 'saw' \mid 'kissed' \mid 'hit' \mid 'sang'$   
 $SV \rightarrow 'said' \mid 'knew' \mid 'alleged'$   
 $IV \rightarrow 'barked' \mid 'disappeared' \mid 'elapsed' \mid 'sang'$

Notice that according to (34), a given lexical item can belong to more than one subcategory. For example, *sang* can occur both with and without a following NP complement.

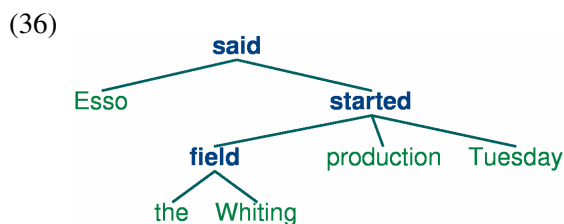
#### 7.4.4 Dependency Grammar

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 root of a sentence is usually taken to be the main verb, and every other word is either dependent on the root, or connects to it through a path of dependencies. Figure (35) illustrates a dependency graph, where the head of the arrow points to the head of a dependency.



As you will see, the arcs in Figure (35) 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 \(36\)](#), where dependents are shown as daughters of their heads.



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, Hall, & Nilsson, 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_contrib.dependency import DepGraph
>>> dg = DepGraph().read("""Eso 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 `deptrree()` method of a `DepGraph()` object to build an NLTK tree like that illustrated earlier in [\(36\)](#).

```

>>> tree = dg.deptrree()
>>> tree.draw()

```

### 7.4.5 Formalizing Context Free Grammars

We have seen that a CFG contains terminal and nonterminal symbols, and productions that 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:

- $\Sigma$  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
  - $\alpha$  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, [\(37\)](#) is a derivation of *the dog with a telescope* for the grammar in [\(24\)](#).

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

(38) NP  
 Det N PP  
 Det N P NP  
 Det N with NP  
 ...

We can also write derivation (37) as:

(39) NP  $\Rightarrow$  DET N PP  $\Rightarrow$  *the* N PP  $\Rightarrow$  *the dog* PP  $\Rightarrow$  *the dog* P NP  $\Rightarrow$  *the dog with* NP  $\Rightarrow$  *the dog with a* N  $\Rightarrow$  *the dog with a telescope*

where  $\Rightarrow$  means “derives in one step”. We use  $\Rightarrow^*$  to mean “derives in zero or more steps”:

- $\alpha \Rightarrow^* \alpha$  for any string  $\alpha$ , and
- if  $\alpha \Rightarrow^* \beta$  and  $\beta \Rightarrow \gamma$ , then  $\alpha \Rightarrow^* \gamma$ .

We write  $A \Rightarrow^* \alpha$  to indicate that  $\alpha$  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 (24) be used to describe sentences that 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  $\rightarrow$  Det N PP, to NP  $\rightarrow$  NP PP. Using the *Step* button, try to build a parse tree. What happens?
4. ● Extend the grammar in (27) with productions that 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 7.2** Context Free Grammars in NLTK

---

```

grammar = nltk.parse_cfg("""
 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"
 """)

>>> sent = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar)
>>> for p in rd_parser.nbest_parse(sent):
... print p
(S (NP Mary) (VP (V saw) (NP Bob)))

```

---

5. ① Pick some common verbs and complete the following tasks:
  - a) Write a program to find those verbs in the Prepositional Phrase Attachment Corpus `nltk.corpus.ppattach`. 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 [Section \(9\)](#)).
  - b) Devise CFG grammar productions to cover some of these cases.
6. ★ Write a function that takes a grammar (such as the one defined in [Listing 7.2](#)) and returns a random sentence generated by the grammar. (Use `grammar.start()` to find the start symbol of the grammar; `grammar.productions(lhs)` to get the list of productions from the grammar that have the specified left-hand side; and `production.rhs()` to get the right-hand side of a production.)
7. ★ **Lexical Acquisition:** As we saw in [Chapter 6](#), 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 that 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.


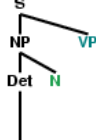
Parsing is important in both linguistics and natural language processing. A parser permits a grammar to be evaluated against a potentially large collection of test sentences, helping linguists to find any problems in their grammatical analysis. A parser can serve as a model of psycholinguistic processing, 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  $S \rightarrow NP VP$  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.draw.rdparser.demo()`. Six stages of the execution of this parser are shown in [Table 7.3](#).

|                                                                                                                                                                                                       |                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p style="text-align: center;">S</p> <hr style="border-top: 1px dashed black;"/> <p style="text-align: center;">the dog saw a man in the park</p> <p style="text-align: center;">a. Initial stage</p> |  <hr style="border-top: 1px dashed black;"/> <p style="text-align: center;">the dog saw a man in the park</p> <p style="text-align: center;">b. 2nd production</p> |  <hr style="border-top: 1px dashed black;"/> <p style="text-align: center;">the dog saw a man in the park</p> <p style="text-align: center;">c. Matching <i>the</i></p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

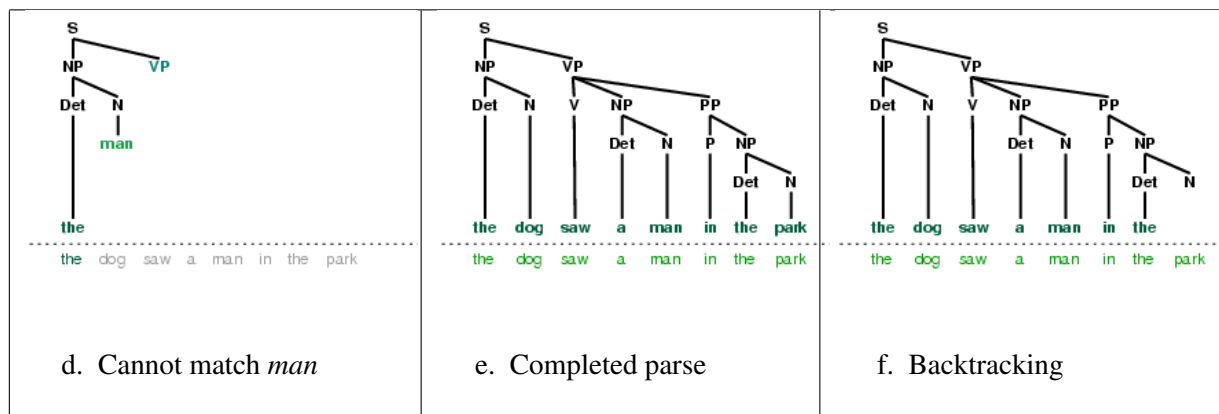


Table 7.3: Six Stages of a Recursive Descent Parser

During this process, the parser is often forced to choose between several possible productions. For example, in going from step 3 to step 4, it tries to find productions with N on the left-hand side. The first of these is  $N \rightarrow \textit{man}$ . When this does not work it *backtracks*, and tries other N productions in order, under it gets to  $N \rightarrow \textit{dog}$ , which matches the next word in the input sentence. Much later, as shown in step 5, it finds a complete parse. This is a tree that 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:

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar)
>>> sent = 'Mary saw a dog'.split()
>>> for t in rd_parser.nbest_parse(sent):
... print t
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

### Note

`RecursiveDescentParser()` 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  $NP \rightarrow NP PP$  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  $VP \rightarrow V NP$  will discard the subtree created for the NP. If the parser then proceeds with  $VP \rightarrow V NP PP$ , then the NP subtree must be created all over again.

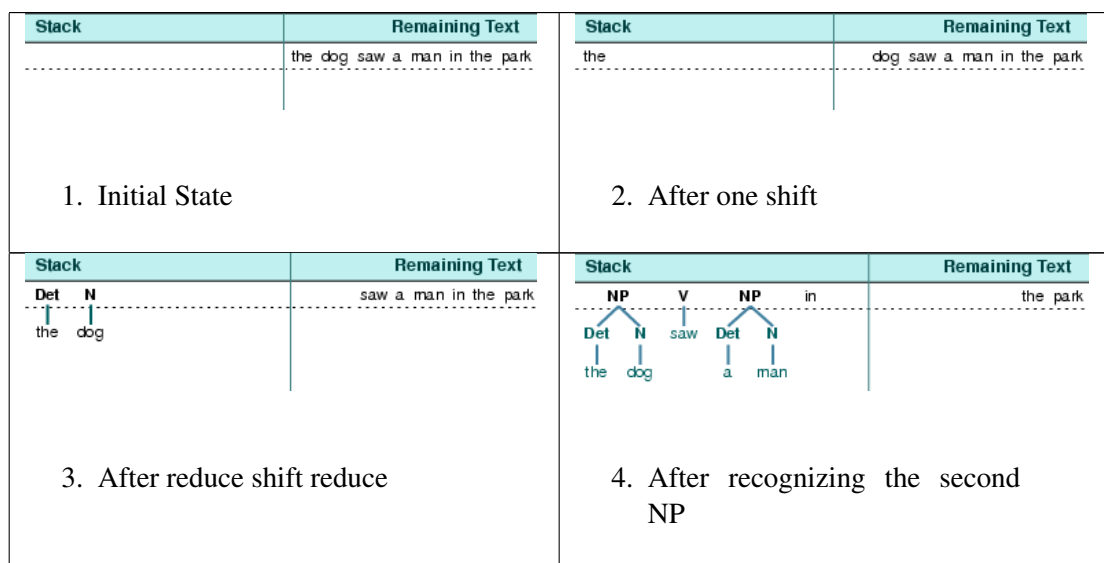
Recursive descent parsing is a kind of **top-down parsing**. Top-down parsers use a grammar to *predict* what the input will be, before inspecting the input! However, since the input is available to the parser all along, it would be more sensible to consider the input sentence from the very beginning. 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 5.2.4); 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  $N \rightarrow \text{dog}$ , 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  $N(\text{dog})$ . At a later stage, if the top of the stack holds two items  $\text{Det}(\text{the})$   $N(\text{dog})$  and if the grammar has a production  $\text{NP} \rightarrow \text{DET } N$  then the reduce operation causes these two items to be replaced with  $\text{NP}(\text{Det}(\text{the}), N(\text{dog}))$ . 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.draw.srparsers.demo()`. Six stages of the execution of this parser are shown in Figure 7.4.





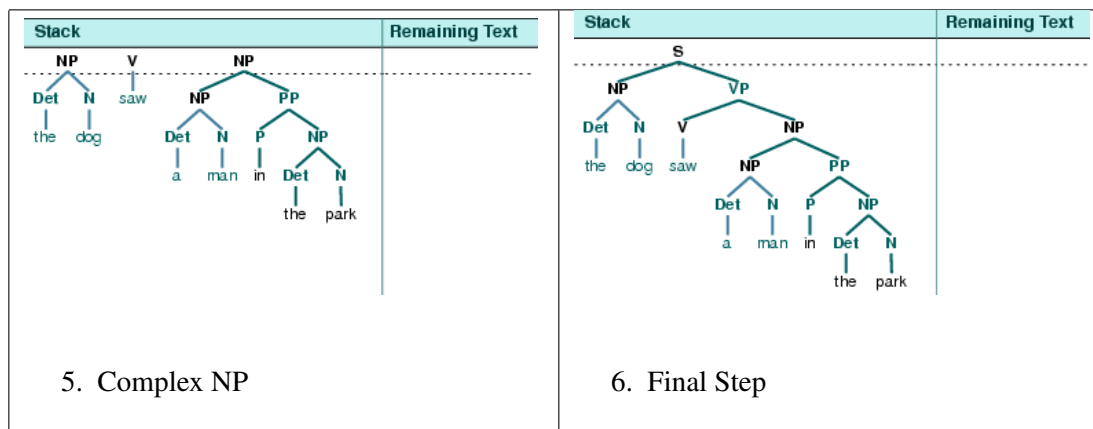


Table 7.4: Six Stages of a Shift-Reduce Parser

NLTK provides `ShiftReduceParser()`, 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 that controls how verbosely the parser reports the steps that it takes as it parses a text:

```
>>> sr_parse = nltk.ShiftReduceParser(grammar, trace=2)
>>> sent = 'Mary saw a dog'.split()
>>> print 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 7.51](#). The parser entered this blind alley at an earlier stage shown in [Table 7.52](#), when it reduced instead of shifted. This situation is called a **shift-reduce conflict**. At another possible stage of processing shown in [Table 7.53](#), the parser must choose between two possible reductions, both matching the top items on the stack:  $VP \rightarrow VP NP PP$  or  $NP \rightarrow NP PP$ . This situation is called a **reduce-reduce conflict**.

| Stack                            | Remaining Text |
|----------------------------------|----------------|
| <p>1. Dead end</p>               |                |
| <p>2. Shift-reduce conflict</p>  | in the park    |
| <p>3. Reduce-reduce conflict</p> |                |

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 favoring the reduction operation that removes the most items from the stack. No such policies are failsafe however.

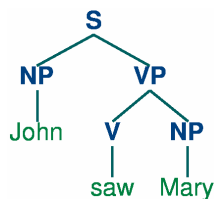
The advantages of shift-reduce parsers over recursive descent parsers is that they only build structure that corresponds to the words in the input. Furthermore, they only build each sub-structure once, e.g.  $\text{NP}(\text{Det}(\text{the}), \text{N}(\text{man}))$  is only built and pushed onto the stack a single time, regardless of whether it will later be used by the  $\text{VP} \rightarrow \text{V NP PP}$  reduction or the  $\text{NP} \rightarrow \text{NP PP}$  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 (27) allows us to produce the following parse of *John saw Mary*:

(40)



Recall that the grammar in (27) has the following productions for expanding NP:

- (41)
- $NP \rightarrow DT\ NOM$
  - $NP \rightarrow DT\ NOM\ PP$
  - $NP \rightarrow PROP\ N$

Suppose we ask you to first look at tree (40), and then decide which of the NP productions you'd want a recursive descent parser to apply first — obviously, (41c) is the right choice! How do you know that it would be pointless to apply (41a) or (41b) 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*  $\alpha$ . More generally, we say that a category  $B$  is a **left-corner** of a tree rooted in  $A$  if  $A \Rightarrow^* B\ \alpha$ .

(42)



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 (27).

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

Table 7.6: Left-Corners in (27)

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

- ☼ 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 5.5.4). Write a function that 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](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 `ParserI`. (Note, this exercise requires knowledge of Python classes, covered in Chapter 9.)
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 that 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.delphin.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, that 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

For more examples of parsing with NLTK, please see the guide at <http://nltk.org/doc/guides/parse.html>.

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 analyze constituency in English, with extensive exemplification and exercises. [Huddleston & Pullum, 2002] provides an up-to-date and comprehensive analysis of syntactic phenomena in English.

Chapter 12 of [Jurafsky & Martin, 2008] covers formal grammars of English; Sections 13.1-3 cover simple parsing algorithms and techniques for dealing with ambiguity; Chapter 16 covers the Chomsky hierarchy and the formal complexity of natural language.

- 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](#)

### About this document...

This chapter is a draft from *Natural Language Processing* [<http://nltk.org/book.html>], by Steven Bird, Ewan Klein and Edward Loper, Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://nltk.org/>], Version 0.9.5, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is